UNIVERSITÀ DEGLI STUDI DI CAGLIARI
Facoltà di Scienze Matematiche Fisiche Naturali
Dipartimento di Matematica ed Informatica
Corso di Laurea in Informatica

Dispense del corso di

Sistemi Operativi

Prof. Francesco Aymerich

A.A. 2003-2004

Testi di riferimento:

A. M. Lister
Fondamenti dei sistemi operativi
Ed. Tecniche Nuove - Collana Università+Ricerca;

A. S. Tanenbaum - A. S. Woodhull.
Sistemi operativi
Prentice Hall - UTET Seconda Ed. 1998

Formattazione a cura di Nicola Asuni (2003-03-15 - 2004-09-14)

INDICE

INDICE **2**

1. INTRODUZIONE **5**

1.1.1 Condivisione delle risorse *5*
1.1.2 Disponibilità di una macchina virtuale *5*
1.1.3 I livelli dell'architettura di un elaboratore *5*
1.2 TIPOLOGIA DEI SISTEMI OPERATIVI 6
1.2.1 Sistemi monoutente *6*
1.2.2 Controllo dei processi *7*
1.2.3 Sistemi a Data Base *7*
1.2.4 Trattamento delle transazioni *7*
1.2.5 I sistemi universali *7*
1.3 STRUTTURA DEI SISTEMI OPERATIVI 8
1.3.1 Sistemi monolitici *8*
1.3.2 Sistemi a strati *9*
1.3.3 Macchine virtuali *9*
1.3.4 UNIX *11*
1.3.5 Modello client-server *12*
1.4 IL SISTEMA OPERATIVO IPOTETICO 13
1.5 RIEPILOGO 13

2 FUNZIONI E CARATTERISTICHE DI UN SISTEMA OPERATIVO **13**

2.1 LE FUNZIONI DEL SISTEMA OPERATIVO 14
2.1.1 Concorrenza *15*
2.1.2 Condivisione *15*
2.1.3 Memorizzazione a lungo termine *15*
2.1.4 Determinatezza *15*
2.2 REQUISITI AUSPICABILI 16
2.2.1 Efficienza *16*
2.2.2 Affidabilità *16*
2.2.3 Facilità di manutenzione *16*
2.2.4 Dimensioni ridotte *16*
2.3 RIEPILOGO 16

3 PROCESSI CONCORRENTI **16**

3.1 PROGRAMMI, PROCESSI E PROCESSORI 16
3.2 LA COMUNICAZIONE TRA I PROCESSI 19
3.2.1 Mutua esclusione *19*
3.2.2 Situazioni di stallo *19*
3.3 I SEMAFORI 20
3.3.1 Mutua esclusione *20*
3.3.2 Sincronizzazione *21*
3.4 I MONITOR 23
3.5 I MESSAGGI 25
3.5.1 Send e receive *25*
3.5.2 Problemi progettuali per i sistemi a scambio di messaggi *26*
3.5.3 Il problema del produttore-consumatore risolto con lo scambio di messaggi *26*
3.6 PROBLEMI CLASSICI DI IPC 27
3.6.1 Problema dei filosofi affamati *27*
3.6.2 Problema dei lettori e degli scrittori *28*
3.6.3 Problema del barbiere addormentato *29*
3.7 RIEPILOGO 30

4 IL NUCLEO DEL SISTEMA **30**

4.1 RISORSE HARDWARE ESSENZIALI 30
4.1.1 Meccanismo degli interrupt *30*
4.1.2 Protezione della memoria *30*
4.1.3 Insieme di istruzioni privilegiate *30*
4.1.4 Orologio in tempo reale *31*
4.2 DESCRIZIONE DEL NUCLEO 31

4.3 LA RAPPRESENTAZIONE: DEI PROCESSI 31
4.3.1 I thread *33*
4.4 FLIH 34
4.4.1 Skip chain *34*
4.4.2 Gli interrupt su Pentium *37*
4.5 LA ROUTINE DI SMISTAMENTO 38
4.6 L'IMPLEMENTAZIONE DI WAIT E SIGNAL 39
4.6.1 Raggruppamento dei processi *39*
4.6.2 Inserimento e prelievo nella coda d'attesa *39*
4.6.3 Allocazione dei processi *40*
4.6.4 Indivisibilità *40*

5 GESTIONE DELLA MEMORIA 42

5.1 OBIETTIVI 42
5.1.1 Rilocazione *42*
5.1.2 Protezione *42*
5.1.3 Condivisione *42*
5.1.4 Organizzazione logica *42*
5.1.5 Organizzazione fisica *43*
5.2 LA MEMORIA VIRTUALE 43
5.3 L'IMPLEMENTAZIONE DELLA MEMORIA VIRTUALE 44
5.3.1 Registri base e registri limite *44*
5.3.2. Swapping *45*
5.3.3. Paginazione *45*
5.3.4 Segmentazione *48*
5.3.5 Segmentazione con paginazione *49*
5.3.6 Segmentazione con paginazione: il Pentium Intel *50*
5.4 METODI DI ALLOCAZIONE DELLA MEMORIA 52
5.4.1 Metodi di posizionamento per sistemi segmentati *52*
5.4.2 Metodi di posizionamento per sistemi paginati *53*
5.4.3 Metodi di sostituzione per sistemi paginati *53*
5.4.4 Metodi di sostituzione per sistemi non paginati *53*
5.4.5 Metodi di ricerca e prelievo (fetch) *54*
5.5 IL MODELLO DELL'AREA DI LAVORO 54
5.6 IMPLEMENTAZIONE NEL NOSTRO SISTEMA IPOTETICO 55

6 INPUT E OUTPUT 56

6.1 CONSIDERAZIONI GENERALI 56
6.1.1 Dispositivi di I/O *56*
6.1.2 Controllori di dispositivo *57*
6.1.3 Accesso diretto in memoria (DMA) *58*
6.1.4 Hardware dei dischi *59*
6.1.5 Software del disco *60*
6.1.6 Gestione degli errori *61*
6.2 OBIETTIVI DEL PROGETTO ED IMPLICAZIONI RELATIVE 61
6.2.1 Indipendenza dal codice dei caratteri *61*
6.2.2 Indipendenza dal tipo di periferica *61*
6.2.3 Efficienza *61*
6.2.4 Uniformità di gestione delle periferiche *61*
6.2.5 Gestione degli errori *61*
6.2.6 Sincronizzazione *62*
6.2.7 Condivisione e non *62*
6.2.8 Organizzare il software a livelli *62*
6.2.9 Gestori di periferica *62*
6.2.10 Software di I/O indipendente dal dispositivo *63*
6.2.11 Software di I/O nello spazio utente *64*
6.3 LE PROCEDURE DI I/O 65
6.4 GESTIONE DEL BUFFER 68
6.5 LE UNITÀ DI SUPPORTO AI FILE 68
6.6 GESTIONE DELLO SPOOL 70

7 IL FILE SYSTEM 71

7.1 OBIETTIVI 71
7.2 LE DIRECTORY 72
7.3 CONDIVISIONE E SICUREZZA 73

7.4 L'ORGANIZZAZIONE DELLA MEMORIA SECONDARIA 74
7.5 L'INTEGRITÀ DEL FILE SYSTEM 76
7.6 APERTURA E CHIUSURA DEI FILE 77
7.7 CONCLUSIONI 78

8 SCHEDULAZIONE E ALLOCAZIONE DELLE RISORSE 78

8.1 OSSERVAZIONI DI CARATTERE GENERALE 78
8.2 I MECCANISMI DI ALLOCAZIONE 79
8.3 LE SITUAZIONI DI STALLO 80
8.4 IL GESTORE DEI PROCESSI 82
8.6 LE GERARCHIE DEI PROCESSI 85
8.7 CONTROLLO E CONTABILITÀ DELLE RISORSE 87
8.8 RIASSUNTO 88

1. INTRODUZIONE

In questo capitolo e nel successivo tenteremo di dare risposta alle seguenti domande:

- che cos'è un sistema operativo? che cosa fa?
- perché è necessario?

Soddisfacendo tali interrogativi, speriamo di dare al lettore un'idea degli obiettivi che tenteremo di raggiungere quando, nei capitoli successivi, si parlerà del modo in cui sono costruiti i sistemi operativi.
Cominceremo con l'esaminare gli scopi, in senso lato, che si prefigge il sistema operativo e con il classificare i vari tipi di sistemi operativi attualmente esistenti.
Generalizzando, un sistema operativo svolge le due funzioni principali di seguito esposte.

1.1.1 Condivisione delle risorse

Un sistema operativo deve suddividere le risorse del sistema di calcolo tra le persone che ne fanno uso simultaneamente. Lo scopo, perciò, è quello di aumentare la disponibilità del sistema di calcolo per gli utenti, rendendo massimo al contempo l'utilizzo di risorse quali le unità centrali, la memoria ed i dispositivi di input/output. L'importanza dell'utilizzazione delle risorse è funzione del costo delle risorse interessate, anche se la continua diminuzione dei costi dell'hardware ha decrementato l'importanza dell'utilizzazione delle risorse, fino al punto che molti microcomputer sono ora dedicati ad un'unica funzione e non vengono mai condivisi. Tuttavia, i sistemi di calcolo di maggiori dimensioni restano pur sempre costosi, tanto da garantire il compimento di sforzi notevoli per la condivisione delle loro risorse.

1.1.2 Disponibilità di una macchina virtuale

La seconda funzione essenziale del sistema operativo è quella di trasformare l'hardware, quale si presenta fisicamente, in una macchina di più semplice impiego: è come se si presentasse all'utente una macchina virtuale con caratteristiche diverse da quelle della macchina fisica di base, ma più duttili. In alcuni componenti, la macchina virtuale presenta spesso divergenze dalla macchina reale su cui si basa: di seguito ne diamo un elenco.

(a) Input/Output (I/O)
Le proprietà di I/O dell'hardware di base possono essere estremamente complesse e possono richiedere una programmazione anche molto sofisticata per divenire utilizzabili. Grazie al sistema operativo, l'utente sarà sollevato dall'oneroso compito di comprendere tale complessità e gli si presenterà una macchina virtuale con proprietà di I/O molto più semplici da usare, pur restando invariate per quanto riguarda la loro potenza.

(b) Memoria
Molti sistemi operativi presentano una macchina virtuale con una memoria diversa per dimensione da quella della macchina reale di base. Ad esempio, un sistema operativo può impiegare la memoria secondaria (sotto forma di dischi magnetici) per dare l'illusione di una memoria centrale più ampia; in alternativa, può suddividere quest'ultima tra gli utenti, cosicché ciascuno di essi si troverà di fronte ad una macchina virtuale con memoria inferiore a quella della macchina reale.

(c) File system
La maggior parte delle macchine virtuali comprende un file system per la memorizzazione di dati e programmi permanenti. Il file system si basa sui supporti di informazione (disco o nastro) della macchina reale, ma il sistema operativo permette all'utente di avere accesso alle informazioni memorizzate per mezzo di un nome simbolico, anziché dover ricorrere alla loro posizione fisica sul supporto di informazione.

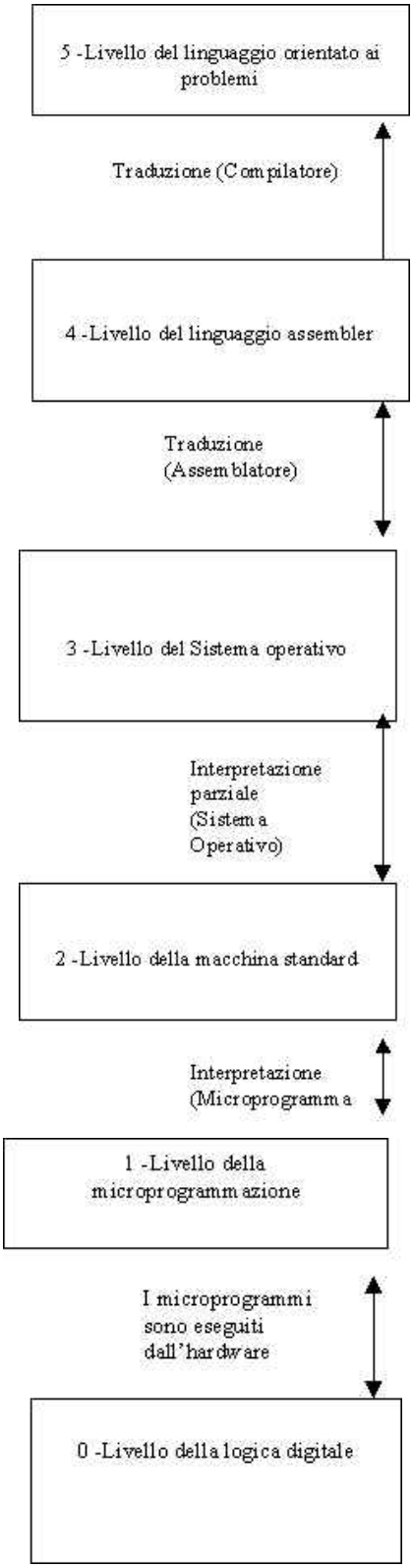
(d) Protezione e gestione degli errori
Dal momento che, nella maggior parte dei casi, i sistemi di calcolo di vaste dimensioni vengono condivisi da svariati utenti, è essenziale che ciascuno di essi sia protetto dalle conseguenze degli errori (intenzionali o meno) compiuti da altri. I computer variano notevolmente per quanto concerne il grado di protezione fornita dall'hardware di base ed il sistema operativo deve avvalersi di questa caratteristica per fornire una macchina virtuale nel cui ambito gli utenti non possano influenzarsi negativamente in modo reciproco.

(e) Interazione dei programmi
Una macchina virtuale può fornire facilitazioni per i programmi utente, così da farli interagire mentre girano in macchina; ad esempio, l'output di un programma viene utilizzato quale input per un altro programma.

(f) Controllo dei programmi
Una macchina virtuale fornisce all'utente uno strumento per manipolare i programmi ed i dati che si trovano dentro al sistema di calcolo. All'utente viene presentato un linguaggio di comando che gli consente di esprimere ciò che vuole far eseguire alla macchina - per esempio, la compilazione e l'esecuzione di un determinato programma, la fusione di due gruppi di dati contenuti nel file system, e così via. Il linguaggio di comando è ad un livello molto più alto (ed è molto più facile da usare) delle istruzioni in linguaggio macchina, eseguibili dalla macchina fisica.
La natura precisa della macchina virtuale dipenderà dall'applicazione per la quale se ne prevede l'uso. Ad esempio, le caratteristiche di una macchina necessaria per un sistema di prenotazione dei posti per compagnie aeree saranno diverse da quelle richieste per il controllo di esperimenti scientifici. Chiaramente, la progettazione de] sistema operativo deve essere nettamente determinata dal tipo di impiego cui è destinata la macchina. Purtroppo, con le "macchine universali" si dà spesso il caso che non si riesca ad individuare facilmente il tipo di applicazione prevista; una critica comunemente sollevata a proposito di parecchi sistemi è che, nel tentativo di voler essere tutto per tutti, finiscono per non essere completamente soddisfacenti per nessuno. Nel paragrafo successivo esamineremo vari tipi di sistemi ed individueremo alcune delle loro caratteristiche.

1.1.3 I livelli dell'architettura di un elaboratore

Ricordiamo dall'Architettura degli Elaboratori i sei livelli presenti nella maggior parte dei sistemi moderni. Il metodo con cui viene supportato ogni livello è indicato sotto ogni riquadro, con il nome del programma che lo supporta, indicato tra parentesi. Una differenza tra i primi tre livelli e gli altri è la natura del linguaggio fornito. I linguaggi di macchina dei livelli 1, 2 e 3 sono numerici. I loro programmi sono costituiti da una lunga serie di numeri, ottimi per le macchine, pessimi per le persone. Cominciando dal livello 4 i linguaggi contengono parole ed abbreviazioni significative per le persone.



Il livello 5 è costituito da un linguaggio progettato per essere utilizzato dai programmatori di applicazioni con problemi da risolvere. Tali linguaggi sono chiamati linguaggi di alto livello. I programmi scritti in questi linguaggi sono generalmente tradotti nel livello 3 o 4 da traduttori conosciuti come compilatori, sebbene occasionalmente siano anche interpretati.

Il livello 4, il livello del linguaggio assembler, è una forma simbolica per uno dei linguaggi sottostanti. Questo livello fornisce un metodo per scrivere programmi per i livelli 1, 2 e 3 in forma non così ostica come i linguaggi delle macchine virtuali stesse. Un altro cambiamento che interviene a livello 4 è il metodo con cui i livelli superiori sono supportati. I livelli 2 e 3 sono sempre interpretati. I livelli 4, 5 e superiori sono di solito, anche se non sempre, supportati da traduzione.

I programmi nel linguaggio assembler sono prima tradotti nei linguaggi di livello 1, 2, 3 e poi interpretati da un'appropriata macchina virtuale o effettiva. I programmi che eseguono la traduzione sono chiamati assemblatori.
Le nuove capacità aggiunte a livello 3 sono eseguite da un interprete che fa operare il livello 2.

Il terzo livello è solitamente un livello ibrido. La maggior parte delle istruzioni nel suo linguaggio appaiono anche nel linguaggio di livello 2. Il terzo livello fornisce un insieme di nuove istruzioni, un'organizzazione della memoria differente, la capacità di eseguire due o più programmi in parallelo e altre caratteristiche varie.
Le istruzioni di livello 3 identiche a quelle di livello 2 vengono eseguite direttamente dal microprogramma e non dal sistema operativo. In altre parole alcune istruzioni di livello 3 sono interpretate dal sistema operativo, altre direttamente dal microprogramma. Ecco cosa intendiamo per ibrido. Chiameremo questo livello: livello del sistema operativo.

Le macchine di livello 2 hanno molto in comune fra loro. Perfino macchine di livello 2 di differenti aziende hanno più similitudini che differenze. In questo libro chiameremo questo livello, livello della macchina standard, per mancanza di un nome universalmente usato.
Ogni produttore di calcolatori pubblica un manuale per ogni calcolatore. Questi manuali riguardano in realtà la macchina virtuale dì livello 2 e non la macchina reale di livello 1, uno per ogni interprete. Bisogna dire che alcuni calcolatori non hanno un livello di microprogrammazione. Su queste macchine le istruzioni del livello di macchina standard sono eseguite direttamente dai circuiti elettronici (livello 0) senza che nessun livello 1 intervenga come interprete. Ne risulta che il livello 1 e non il livello 2 è il livello della macchina standard.

Il livello successivo è il livello 1, che è il livello del vero linguaggio macchina. In contrasto con il livello 0, dove non c'è il concetto di programma come una sequenza di istruzioni da eseguire, nel livello 1 c'è un programma, chiamato microprogramma, il cui lavoro è di interpretare le istruzioni del livello 2.. La maggior parte di queste istruzioni si occupano dello spostamento di dati da una parte della macchina ad un'altra o dell'esecuzione di semplici test.

Al livello più basso, il livello della logica digitale, gli oggetti che ci interessano sono chiamati gate (porte). Ogni gate ha uno o più input digitali (segnali rappresentanti 0 o 1) e fornisce come output alcune semplici funzioni di questi input, quali AND o OR.

Figura 1 - I 6 livelli di un sistema di elaborazione

1.2 TIPOLOGIA DEI SISTEMI OPERATIVI

I sistemi operativi hanno una lunga storia, a partire dai giorni in cui essi sostituivano l'operatore, fino ai moderni sistemi multiprogrammati.

1.2.1 Sistemi monoutente

I sistemi monoutente, come indica il nome stesso, forniscono una macchina virtuale per un solo utente alla volta. Sono idonei a computer dedicati ad un'unica funzione oppure tanto economici da non valere la pena condividerli. La maggior parte dei sistemi operativi per microcomputer (ad esempio, Macintosh e Windows) corrisponde al tipo per monoutenza. I sistemi monoutente, in linea generale, presentano una macchina virtuale semplice che facilita l'applicazione di un vasto numero di pacchetti software, permettendo nel contempo all'utente di sviluppare e mettere in esecuzione programmi per

proprio conto. In altre parole, si dà maggior rilievo alla disponibilità di un linguaggio di comando facilmente utilizzabile, di un file system semplice e di risorse di I/O per dischi e dispositivi vari.

1.2.2 Controllo dei processi

Il termine controllo dei processi indica solitamente il controllo, effettuato tramite calcolatore, di un processo industriale, quale la raffinazione del petrolio greggio o la produzione di macchine utensili. Tale concetto può essere allargato fino a comprendere settori quali il controllo ambientale in una capsula aerospaziale, o il monitoraggio delle condizioni di un degente in ospedale. La caratteristica comune di tutte queste applicazioni è il feedback (retroazione): il computer, cioè, riceve un input proveniente dal processo sotto controllo, elabora una risposta per mantenerlo in condizioni di stabilità ed avvia il meccanismo per l'emissione della risposta stessa. Se, ad esempio, i dati di input denotassero un pericoloso aumento di temperatura, l'intervento di risposta potrebbe benissimo constare nell'apertura di una valvola per aumentare la portata del liquido di raffreddamento. È chiaro che, se il processo deve rimanere in equilibrio (per non parlare della relativa sicurezza), esiste un intervallo di tempo critico entro il quale deve essere fornita la risposta da parte del calcolatore. La funzione principale di un sistema operativo adibito al controllo di un processo è quella di fornire il massimo di affidabilità con il minimo indispensabile di interventi da parte dell'operatore, nonché quella di garantire condizioni di sicurezza anche al verificarsi di eventuali disfunzioni dell'hardware.

1.2.3 Sistemi a Data Base

Il carattere distintivo di tali sistemi è il vasto insieme di dati, o data base, che può essere consultato allo scopo di ottenere informazioni. La risposta alle richieste di informazione deve avere luogo entro un periodo di tempo breve (sono caratteristici tempi inferiori al minuto) e deve essere possibile modificare il data base mano a mano che si aggiornano le informazioni. A tale proposito, citiamo a titolo di esempi i sistemi informativi gestionali (dove il data base è costituito da informazioni sull'andamento aziendale) ed i sistemi informativi sanitari, per i quali il data base consta di un insieme di anamnesi. L'utente (nei casi esemplificati, il direttore della società o il medico) si aspetta di essere in grado di accedere alle informazioni senza dover disporre di alcuna conoscenza relativa all'organizzazione del data base, né per quel che riguarda la struttura del software, né per i dispositivi hardware di memorizzazione. Perciò, il sistema operativo deve mettere a disposizione i servizi di consultazione senza coinvolgere l'utente nella loro realizzazione pratica.

1.2.4 Trattamento delle transazioni

I sistemi per il trattamento delle transazioni sono caratterizzati da un data base che viene modificato con frequenza, finanche parecchie volte al secondo. Se ne possono trovare tipiche applicazioni nel settore delle prenotazioni aeree e delle attività bancarie. Nel primo caso il data base contiene informazioni sulla disponibilità di posti ed è soggetto a modifica ogniqualvolta ne viene prenotato uno; nel secondo caso, consta degli estremi dei conti bancari che vengono aggiornati per ogni accredito e addebito. La principale limitazione nell'elaborazione delle transazioni è data dalla necessità di mantenere aggiornato il data base: è chiaro che il sistema è inutile se le transazioni vengono effettuate sulla base di dati inesatti. Problemi ulteriori sorgono poi quando si trattano transazioni effettuate contemporaneamente sugli stessi dati (ad esempio due agenti di viaggio che tentano di prenotare lo stesso posto). Il singolo utente, naturalmente, non dovrebbe essere consapevole di questi problemi, che vanno risolti dal sistema operativo in modo tale che egli abbia l'impressione di essere l'unico utente del sistema.

1.2.5 I sistemi universali

I sistemi operativi universali sono impiegati per calcolatori caratterizzati da un vasto numero di utenti che svolgono una grande serie di compiti. Tali sistemi vengono progettati in modo da gestire un flusso continuativo di lavoro sotto forma di diversi job (lavori) da eseguirsi sul calcolatore. Ciascun job svolge un compito specifico per un utente determinato (per esempio, analisi dei dati di un'indagine, soluzione equazioni differenziali, calcolo mensile delle paghe) e di solito consta nella messa in macchina di uno o più programmi. Per un caso semplice, poniamo quello de l'analisi dei dati relativi ad un'indagine, il job può limitarsi ad eseguire un programma già compilato assieme ad una serie particolare di dati. Job più complessi potrebbero, invece, comportare la modifica di un programma tramite l'uso di un editor la sua compilazione, per concludere con la sua esecuzione. A causa della varietà job che si possono sottoporre alla macchina, il sistema deve essere in grado di supportare un vasto numero di programmi di utilità, quali i compilatori per svariati linguaggi, gli assemblatori, gli editor, i pacchetti di debug, i word processor ed un file system per la memorizzazione di informazioni permanenti. Deve essere in grado inoltre, di gestire l'ampia tipologia di unità periferiche eventualmente necessari (per esempio: terminali, stampanti, dischi e nastri magnetici, plotter per i grafici, periferiche multimediali.....). La messa a disposizione ed il controllo queste risorse, unitamente all'organizzazione del flusso di lavoro, sono le funzioni in senso lato, di un sistema operativo universale.
Un breve passo indietro nella storia dell'informatica ci aiuta a capire l'evoluzione dei sistemi operativi. I sistemi universali ammettono in genere due categorie di attività: (1) di tipo batch ed (2) interattivo. La caratteristica principale di un sistema di tipo batch è che una volta entrato nel calcolatore, il job non viene più a contatto con l'utente fino suo completamento. Questa modalità era molto diffusa fino agli anni '90, anche se meno rilevante, esiste ancora in tutti i sistemi operativi universali. Il job caratteristico di questo sistema era, agli albori dell'era informatica perforato su schede (il primo calcolatore su cui ho lavorato nei primi anni '60 usava come unità di input un lettore di banda di carta perforata mediante una telescrivente) oppure memorizzato su nastro magnetico e veniva passato ad un operatore per la sottomissione alla macchina. Quando il job era stato completato, l'operatore restituiva l'output all'utente. Per lo più, il job veniva sottoposto al calcolatore tramite dispositivi di input ed output situati all'interno della stessa sala che ospitava il calcolatore; ciò nonostante, alcuni sistemi di tipo batch, chiamati sistemi remote job entry (RJE, elaborazione con accesso a distanza), consentivano la sottomissione dei lavori tramite dispositivi di I/O che potevano anche trovarsi ad una certa distanza dall'impianto centrale ed essere ad esso collegati per mezzo di una linea di trasmissione dei dati. Un sistemi RJE poteva supportare anche parecchie stazioni esterne di I/O e, naturalmente, deve restituire l'output alla stazione dove era stata effettuata la sottomissione del lavoro. Né in un normale sistema di tipo batch, né in un sistema RJE, esiste per l'utente possibilità di interagire con il proprio lavoro mentre è in macchina.
D'altro canto, in un sistema interattivo (accesso multiplo o on demand) l'utente può introdurre un lavoro tramite la tastiera di un terminale e può utilizzare il terminale stesso per sorvegliare controllare il job mentre viene eseguito. Per esempio, l'utente potrebbe correggere gli errori di sintassi riportati dal compilatore, oppure fornire dati che sono funzione dei risultati ottenuti fino a quel momento. Il sistema operativo provvede a suddividere le risorse di calcolo tra i diversi lavori, in modo che a ciascun utente sembri avere per sé tutta la macchina.

Per via dei vantaggi derivanti dall'interazione, il modo ad accesso multiplo tende ad essere usato per attività quali lo sviluppo di programmi e la redazione di documenti, mentre il tipo batch viene ora utilizzato esclusivamente per compiti di routine non interattivi, quali gli statini paga o l'inventario delle scorte di magazzino.
Molti dei sistemi descritti in precedenza possono essere implementati sia su un elaboratore singolo che su svariati computer collegati fra loro. In quest'ultimo caso, i diversi elaboratori possono suddividersi equamente il carico di lavoro oppure, in alternativa, ciascuna macchina può essere dedicata alla gestione di un solo aspetto particolare del lavoro complessivo. Ad esempio, un computer potrebbe essere dedicato alla gestione di tutti gli I/O da stazioni remote, mentre un altro si occupa di tutto il lavoro che richiede un'elaborazione numerica intensiva, I calcolatori possono essere situati nella stessa sala, eventualmente condividendo la memoria (centrale e dei supporti ausiliari), oppure possono essere ubicati ad una certa distanza l'uno dall'altro e comunicare tramite linee di trasmissione dei dati. In tali sistemi distribuiti il sistema operativo deve coordinare le attività di ogni calcolatore e deve garantire un flusso di informazioni adeguate tra di essi, in base alle esigenze.

1.3 STRUTTURA DEI SISTEMI OPERATIVI

L'interfaccia tra il sistema operativo e i programmi utente è definita dal set di "istruzioni estese" fornite dal sistema operativo. Queste istruzioni estese sono tradizionalmente note con il nome di chiamate di sistema (system calls), benché esse possano essere oggi implementate in diversi modi. Per capire veramente cosa fanno i sistemi operativi dobbiamo esaminare con attenzione questa interfaccia. Le chiamate disponibili a questo livello variano da sistema operativo a sistema operativo (benché i concetti sottostanti tendano a rimanere gli stessi). Esse indicano ciò che realmente fa il sistema operativo.
I sistemi operativi possono essere strutturati in diversi modi. I più comuni sono come sistema monolitico, come gerarchia di strati, come sistema a macchine virtuali e infine usando il modello cliente-servitore.

1.3.1 Sistemi monolitici

Questo approccio, che corrisponde di gran lunga all'organizzazione più comune, potrebbe essere sottotitolato "La grande confusione". La struttura dei sistemi monolitici consiste infatti nell'assenza di ogni strutturazione. Il sistema operativo è scritto come un insieme di procedure, ognuna delle quali può indifferentemente chiamare tutte le altre in qualsiasi momento ne abbia bisogno. Quando si usa questa tecnica ogni procedura all'interno del sistema ha una ben definita interfaccia in termini di parametri e risultati, e ognuna è libera di chiamare tutte le altre, purché siano in grado di svolgere le operazioni ad essa necessarie.
Quando si usa questo approccio, per costruire il programma oggetto del sistema operativo, si devono innanzitutto compilare tutte le singole procedure, o i file contenenti le procedure, e quindi si collegano insieme le prime in un unico file oggetto utilizzando il linker di sistema. Di fatto, non c'è nessuna astrazione — ogni procedura è visibile da tutte le altre (in contrasto con una strutturazione a moduli o package, nella quale la maggior parte delle informazioni vengono incapsulate all'interno dei moduli, e solo le funzioni visibili dall'esterno, denominate anche entry point, possono essere utilizzate).

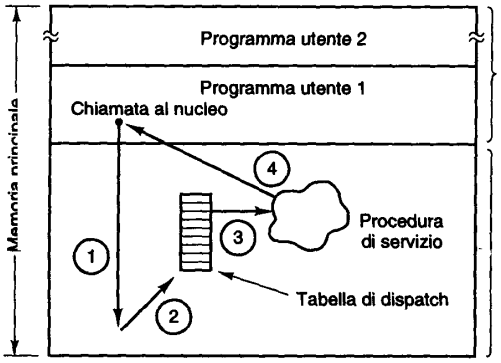
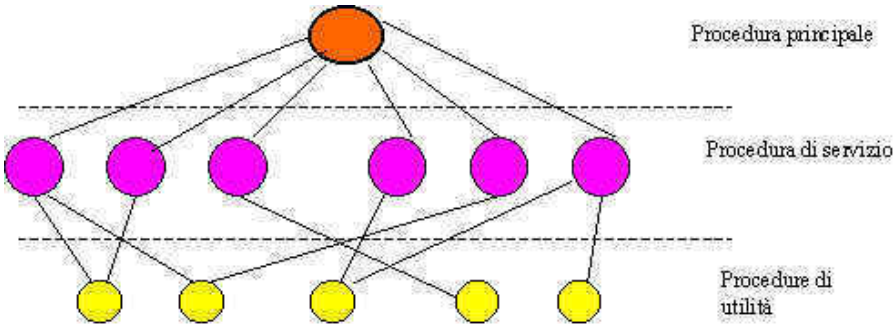


Figura 2 - Come si può effettuare una chiamata di sistema

1. I1 programma utente invoca una trap e passa il controllo al nucleo.
2. Il sistema operativo determina l'indice del servizio richiesto.
3. I1 sistema operativo chiama la procedura di servizio.
4. I1 controllo ritorna al programma utente

Anche nei sistemi monolitici, comunque, è possibile avere un certo grado di strutturazione. I servizi (chiamate di sistema) forniti dal sistema operativo vengono invocati immettendo i parametri in locazioni ben definite, come i registri o lo stack, ed eseguendo quindi una speciale istruzione di trap denominata kernel call (chiamata di nucleo) o supervisor call.
Questa istruzione cambia la modalità della macchina da modo utente a modalità di nucleo (kernel mode) e trasferisce il controllo al sistema operativo, come mostrato dall'evento (1) in figura 2 (La maggior parte delle CPU hanno due modalità: modalità di nucleo, per il sistema operativo, nella quale sono consentite tutte le istruzioni; e modalità utente, per i programmi utente, nella quale l'I/O e alcune altre istruzioni non sono permesse).
Il sistema operativo esamina quindi i parametri della chiamata per determinare quale chiamata di sistema debba essere eseguita, come evidenziato in figura 2.
In seguito il sistema operativo accede a una tabella (tabella di dispatch) che contiene nel k-esimo elemento un puntatore alla procedura che effettua quanto richiesto dalla k-esima chiamata di sistema. Questa operazione, mostrata con (3) in figura 2, identifica la procedura di servizio, che viene quindi chiamata. Quando l'operazione termina, e la chiamata di sistema ha fine, il controllo viene restituito al programma utente (4), in modo da consentirgli di proseguire con l'istruzione immediatamente successiva alla chiamata di sistema.

Figura 3 - Un semplice modello strutturale per un sistema monolitico



Questa organizzazione suggerisce una struttura di base per il sistema operativo:

1. Un programma principale che richiama la procedura di servizio richiesta.
2. Un insieme di procedure di servizio che eseguono le chiamate di sistema.
3. Un insieme di procedure di utilità che forniscono il supporto alle procedure di servizio.

In questo modello ad ogni chiamata di sistema corrisponde una procedura di servizio che si prende cura di essa. Le procedure di utilità svolgono compiti richiesti da diverse procedure di servizio, come ad esempio estrarre i dati dai programmi utente. Questa divisione delle procedure in tre strati viene mostrata in figura 3.

1.3.2 Sistemi a strati

Una generalizzazione dell'approccio di figura 3 è quella di organizzare il sistema operativo come una gerarchia di strati, ognuno costruito al di sopra del precedente. Il primo sistema progettato in questo modo fu il sistema THE, realizzato alla Technische Hogeschool Eindhoven in Olanda da E.W. Dijkstra (1968) e dai suoi studenti. Il sistema THE era un semplice sistema batch per un computer olandese, l'Electrologica X8, che aveva 32 K di parole a 27 bit (a quei tempi i bit erano costosi).

Il sistema aveva 6 livelli come mostrato in figura 4. Il livello 0 si occupava di allocazione del processore, avvicendando i processi in corrispondenza delle interruzioni o della scadenza del timer. Sopra il livello 0 il sistema era composto da processi sequenziali, ognuno dei quali poteva essere programmato senza doversi preoccupare del fatto che più di un processo fosse in esecuzione su di un solo processore. In altre parole, il livello 0 forniva la multi-programmazione della CPU.

Figura 4 - Struttura del sistema operativo THE

Strato	Funzionalità
5	Operatore
4	Programmi utente
3	Gestione dell'input/output
2	Comunicazione processo-console
1	Gestione della memoria e del tamburo
0	Allocazione della CPU e multi-programmazione

Il livello 1 effettuava la gestione della memoria. Esso allocava spazio per i processi nella memoria principale e su un tamburo (o drum, antenato dei moderni hard disk) di 512K parole usato per trattenere parti di processi (pagine) per le quali non vi era spazio in memoria principale. Al di sopra del livello 1 i processi non dovevano preoccuparsi del fatto di essere nella memoria principale piuttosto che nel tamburo; il software del livello 1 si preoccupava di far sì che le pagine dei processi fossero presenti nella memoria principale nel momento in cui venivano richieste.

Il livello 2 gestiva la comunicazione tra ogni processo e la console dell'operatore. Al di sopra di questo livello ogni processo di fatto aveva la sua console di operatore.

Il livello 3 si prendeva cura di gestire i dispositivi di I/O e la memorizzazione nei buffer del flusso di informazioni da e per essi. Al di sopra del livello 3 ogni processo poteva accedere a dispositivi di I/O astratti, più semplici e gradevoli rispetto ai dispositivi reali caratterizzati da svariate peculiarità.

Il livello 4 ospitava i programmi utente. Tali programmi non dovevano preoccuparsi dei processi, della memoria, della console o della gestione dell'I/O. Il processo operatore di sistema era collocato al livello 5.

Un'ulteriore generalizzazione del concetto di stratificazione era presente nel sistema MULTICS, progenitore di UNIX. Invece che su livelli, il sistema MULTICS era organizzato come una serie di anelli concentrici, in cui gli anelli più interni avevano privilegi superiori rispetto agli esterni. Quando una procedura in un anello esterno desiderava chiamare una procedura in un anello interno, doveva effettuare l'analogo di una chiamata di sistema, cioè un'istruzione TRAP i cui parametri venivano accuratamente controllati prima che la chiamata potesse procedere. Benché l'intero sistema operativo facesse parte dello spazio di indirizzamento di ogni processo MULTICS, l'hardware rendeva possibile la definizione di alcune procedure (in realtà segmenti di memoria) come protette in lettura, scrittura o esecuzione.

Mentre lo schema a strati del THE era in realtà solo una convenzione progettuale, poiché tutte le parti del sistema venivano alla fine collegate insieme in un unico programma oggetto, in MULTICS il meccanismo ad anelli agiva realmente a tempo di esecuzione, essendo imposto dall'hardware. Il vantaggio del meccanismo ad anelli consiste nel fatto che esso può essere facilmente esteso ai sottosistemi utente. Ad esempio, un professore potrebbe scrivere un programma per controllare e valutare i programmi degli studenti ed eseguirlo nell'anello n, lasciando che i programmi degli studenti vengano eseguiti al livello n+1 in modo che gli studenti stessi non possano modificare i loro voti.

1.3.3 Macchine virtuali

Le versioni iniziali di os/36O erano sistemi batch puri. Ciononostante, diversi utenti del 360 volevano fare uso del timesharing, così che svariati gruppi, sia all'interno che al di fuori dell'IBM, decisero di scrivere sistemi timesharing per la serie 360. Il sistema timesharing ufficiale dell'IBM, TSS/360, fu consegnato in ritardo e quando finalmente arrivò era così grande e lento che ben poche installazioni lo adottarono. Il sistema TSS/360 fu infine abbandonato dopo che il suo sviluppo era costato 50 milioni di dollari (Graham, 1970), ma un gruppo allo Scientific Center a Cambridge,

Massachusetts, produsse un sistema radicalmente diverso che l'IBM alla fine accettò come prodotto, e che viene ancora oggi utilizzato sui rimanenti mainframe di quel tipo.
Il sistema, originariamente denominato CP/CMS, e in seguito rinominato vM/37O (Seawright e MacKinnon, 1979), era basato su di un'acuta osservazione: un sistema timesharing fornisce (1) multiprogrammazione e (2) una macchina estesa con un'interfaccia più comoda di quella prevista dal semplice hardware. L'essenza del vM/37O consiste nel separare completamente queste due funzioni.
Il cuore del sistema, noto come virtual machine monitor, gira direttamente sull'hardware e realizza la multiprogrammazione, fornendo al livello superiore non una ma svariate macchine virtuali, come mostrato in figura 5. Diversamente da tutti gli altri sistemi operativi, queste macchine virtuali non sono macchine estese, con i file e altre piacevoli astrazioni. Esse sono invece repliche esatte dell'hardware, compresi il modo utente/nucleo, l'I/O, le interruzioni, e quant'altro esiste su di una macchina reale. Poiché ogni macchina virtuale è una replica esatta dell'hardware, ognuna di esse può eseguire un qualsiasi sistema operativo in grado di girare direttamente sull'hardware. Macchine virtuali diverse possono quindi eseguire sistemi operativi diversi, cosa che accade di frequente. Alcune eseguono uno dei discendenti dell'os/360 per le elaborazioni batch o l'elaborazione di transazioni, mentre altre eseguono un sistema interattivo monoutente chiamato CMS (Conversational Monitor System) per utenti in timesharing.
Quando un programma CMS esegue una chiamata di sistema, questa viene tradotta in una trap del sistema operativo in esecuzione sulla macchina virtuale e non in una trap del sistema vM/370, esattamente come se l'istruzione venisse eseguita su di una macchina reale e non virtuale. Il sistema CMS richiama quindi le normali istruzioni hardware per leggere il suo disco virtuale o quant'altro sia necessario per effettuare la chiamata. Queste istruzioni di 1/0 sono tradotte in trap di vM/37O, che le esegue quindi come parte della sua simulazione dell'hardware reale. Separando completamente le funzioni di multiprogrammazione e fornendo una macchina estesa, ognuno dei componenti può essere più semplice, più flessibile e facile da mantenere.

L'idea di una macchina virtuale viene largamente usata ai nostri giorni in un contesto differente: l'esecuzione di vecchi programmi MS-DOS su di un Pentium (o altre CPU a 32 bit Intel). Quando il Pentium e il suo software vennero progettati, sia Intel che Microsoft compresero che vi sarebbe stata una fortissima domanda per poter eseguire il vecchio software sui nuovi hardware. Per questo motivo, Intel fornì una modalità virtuale 8086 sul Pentium. In questa modalità, la macchina si comporta come un 8086 (che è identico all'8088 da un punto di vista dei software), compreso l'indirizzamento a 16 bit e il conseguente limite di memoria di 1 MB.

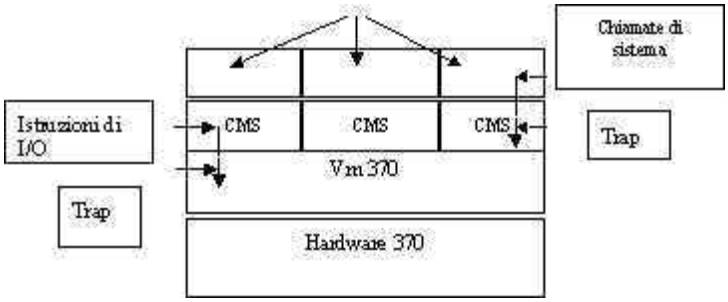


Figura 5 - La struttura di vM/370 con CMS 370 Virtual

Questa modalità viene utilizzata da WINDOWS, os/2 e da altri sistemi operativi per eseguire i programmi MS-DOS. Questi programmi vengono lanciati in modalità virtuale 8086. Finché eseguono istruzioni normali, essi girano direttamente sull'hardware. Viceversa, quando un programma cerca di effettuare una trap nel sistema operativo per richiamare una chiamata di sistema, o cerca di fare direttamente l'I/O in modo protetto, si scatena una trap nei virtual machine monitor.
A questo punto sono possibili due varianti di questo schema: nella prima, MS-DOS stesso viene caricato nello spazio virtuale 8086, in modo che il virtual machine monitor rimandi semplicemente la trap a MS-DOS, esattamente come accadrebbe su un 8086 reale. Quando in seguito MS-DOS stesso cerca di effettuare l'I/O, l'operazione viene intercettata ed eseguita dai virtual machine monitor.
Nell'altra variante, il virtual machine monitor intercetta semplicemente la prima trap e porta esso stesso a compimento l'I/O, essendo a conoscenza di ciò che fanno le chiamate di sistema MS-DOS e quindi ciò che ci si aspetta faccia ogni trap. Questa variante è meno pulita rispetto alla precedente, poiché corrisponde solo all'emulazione di MS-DOS e non di altri sistemi operativi, come potrebbe invece accadere con la precedente. D'altra parte, essa è molto più veloce, poiché risparmia il fastidio di richiamare MS-DOS per effettuare l'I/O. Un ulteriore svantaggio nell'eseguire realmente MS-DOS in modo virtuale 8086 è che MS-DOS fa spesso uso del bit di abilitazione/disabilitazione delle interruzioni usando operazioni che devono essere emulate con un costo considerevole.
È il caso di notare che nessuno di questi approcci è realmente simile a vM/37O, poiché la macchina emulata non è un Pentium completo, ma solo un 8086. Con VM/370 è possibile eseguire vM/370 stesso su di una macchina virtuale. Con il Pentium non è possibile eseguire, ad esempio, WINDOWS sull'8086 virtuale, poiché nessuna versione di WINDOWS gira su 8086; un 286 è il minimo anche per le versioni più vecchie, e l'emulazione del 286 non viene fornita (per non parlare dell'emulazione del Pentium).
In vM/37O ogni processo utente ottiene una copia esatta del computer reale. Con la modalità 8086 virtuale sul Pentium, ogni processo utente ottiene una copia esatta di un computer differente. Spingendosi un po' più in là, i ricercatori del M.I.T. hanno costruito un sistema che fornisce ad ogni utente un clone del computer reale, ma con un sottoinsieme di risorse (Engler et al., 1995). Pertanto una macchina virtuale potrebbe ottenere i blocchi disco da O a 1023, la successiva i blocchi da 1024 a 2047 e così via. Al livello più basso c'è un programma chiamato exokernel, in esecuzione in modalità di nucleo. Il suo compito è allocare risorse alle macchine virtuali, e controllare quindi i tentativi d'utilizzo su di esse per esser sicuro che nessuna macchina cerchi di usare le risorse di qualcun altro. Ogni macchina virtuale può eseguire il suo sistema operativo a livello utente, come in vM/370 e l'8086 virtuale del Pentium, eccetto che ognuna è limitata a usare le risorse richieste che le sono state allocate.
Il vantaggio dello schema exokernel è che risparmia uno strato di corrispondenze. Negli altri schemi, ogni macchina virtuale pensa di possedere un proprio disco, con blocchi che vanno da 0 a un massimo, così che il virtual machine monitor deve gestire delle tabelle per effettuare la corrispondenza degli indirizzi su disco (e di tutte le altre risorse). Nell'exokernel questa operazione di corrispondenza non è necessaria. L'exokernel deve solo sapere quale risorsa sia stata assegnata a quale macchina virtuale. Questo metodo conserva il vantaggio di separare la multiprogrammazione

(nell'exokernel) dal codice di sistema operativo dell'utente (nello spazio utente), ma ad un costo minore, poiché tutto ciò che l'exokernel deve fare è evitare che le macchine virtuali si pestino i piedi a vicenda.

1.3.4 UNIX

Il kernel di UNIX è il cuore del sistema operativo stesso: ogni processo interagisce, infatti, con il kernel direttamente, per mezzo di una chiamata di sistema, o indirettamente, semplicemente quando riceve il permesso di entrare in esecuzione.
Il compito principale del kernel consiste nel controllare e risolvere i conflitti tra i processi riguardo all'uso della CPU. Il kernel seleziona il processo con la priorità più alta che sia pronto ad essere eseguito e gli assegna la CPU. La PRIORITÀ è una misura dell'importanza del processo e viene spesso utilizzata per temporizzare i processi. Il sistema UNIX modifica dinamicamente la priorità dei processi in modo che sia abbassata la priorità dei processi che utilizzano intensivamente la CPU e sia alzata la priorità dei processi a cui da un certo intervallo di tempo non è assegnata la CPU.
Il processo selezionato dal kernel viene quindi eseguito finché non si verifica una delle seguenti condizioni:
• Il processo esaurisce il suo intervallo di tempo. Un intervallo è un breve periodo di tempo che il kernel assegna al processo, per dare a tutti i processi l'opportunità di essere eseguiti.
• Il processo non può proseguire finché non si verifica un certo evento. Ad esempio, quando un processo legge o scrive dati su un file, è necessario attendere finché i dati non siano stati trasferiti.
• Un processo con una priorità più elevata diviene un candidato ad entrare in esecuzione perché l'evento di cui era in attesa si è verificato. Il processo corrente viene "scavalcato" dal processo il cui evento si è verificato.
Quando si verifica una delle precedenti condizioni, il kernel interrompe il processo e assegna la CPU ad un altro processo.
L'algoritmo di selezione del processo da mettere correntemente in esecuzione, detto anche algoritmo di schedulazione dei processi, è strutturato in modo tale da garantire che il kernel non lasci passare troppo tempo tra un intervallo di esecuzione e l'intervallo successivo di ogni processo, che i processi scaricati dalla memoria centrale non restino nella memoria di massa troppo a lungo e soprattutto che ogni processo abbia l'opportunità, prima o poi, di avanzare nell'esecuzione.

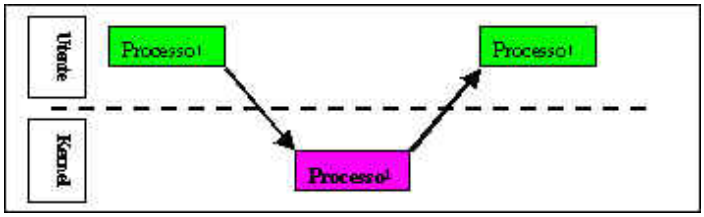


Figura 6 - Commutazione di contesto in UNIX

Uno degli altri compiti del kernel è quello di soddisfare le richieste inoltrate dai processi tramite le chiamate di sistema. Un processo effettua una chiamata di sistema eseguendo una speciale istruzione macchina che forza la CPU a modificare il suo CONTESTO da un modo meno privilegiato, chiamato MODO UTENTE, ad un modo più privilegiato, chiamato MODO KERNEL. La Figura 6 mostra la sequenza di cambiamenti di contesto, chiamata COMMUTAZIONE DI CONTESTO.
Nella Figura, il processo i, in esecuzione in modo utente, effettua una chiamata di sistema che forza una commutazione di contesto dal modo utente al modo kernel. Anche se la CPU modifica il contesto, il sistema UNIX non cambia il processo. Il processo 1 viene eseguito in modo kernel per servire la chiamata di sistema.
Quando il processo 1 in esecuzione in modo kernel, termina le operazioni relative alla chiamata di sistema, il kernel forza un'altra commutazione di contesto ritornando all'esecuzione in modo utente del processo 1. Il processo 0, che rappresenta il kernel, non ha alcun compito nel soddisfare la chiamata di sistema.
La Figura successiva mostra invece il processo i mentre effettua una chiamata di sistema che forza una commutazione di contesto che risulta in una COMMUTAZIONE DI PROCESSO a favore del processo 0, cioè il kernel. Questa situazione usualmente si verifica quando un processo effettua una chiamata di sistema che lo forza ad attendere un evento.
Il processo i, in esecuzione nel modo kernel, si sospende e l'esecuzione passa al processo 0. Il processo 0 seleziona un processo che sia pronto ad entrare in esecuzione. Il processo 0 si sospende a sua volta e viene attivato il processo k, in esecuzione nel modo kernel. Il processo k commuta infine contesto e diventa processo k in esecuzione in modo utente.
L'ultimo compito fondamentale del kernel è l'accesso ai file presenti nei diversi file system. In sintesi, il kernel del sistema UNIX è un unico programma i cui compiti sono l'allocazione della CPU tra processi in conflitto e l'accesso ai file. La CPU viene assegnata ai processi in modo time-sharing provvedendo alla gestione delle interruzioni inviate da parte dei processi con priorità più elevata. Le chiamate di sistema effettuate da un processo vengono servite dal processo stesso eseguito in modo kernel. Il processo 0 che rappresenta il kernel, viene eseguito solo quando la CPU deve essere assegnata ad un altro processo. Infine i processi accedono al file system in risposta a varie chiamate di sistema.

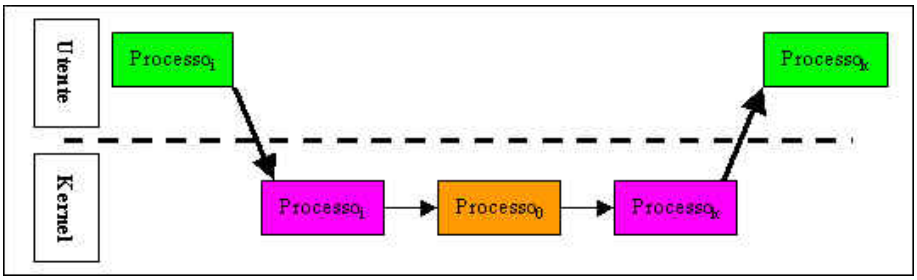
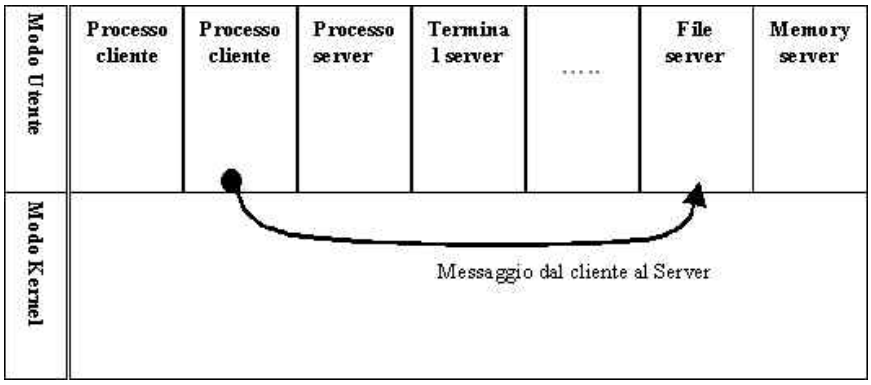


Figura 7 - Commutazione di processo in UNIX

1.3.5 Modello client-server

Nei sistemi operativi moderni c'è la tendenza a spostare ancor più il codice verso i livelli superiori, e rimuovere quanto possibile dal sistema operativo, lasciando un nucleo (kernel) minimale. L'approccio consueto consiste nell'implementare la maggior parte delle funzioni del sistema operativo all'interno di processi utente. Per richiedere un servizio, come la lettura di un blocco di un file, un processo utente (ora chiamato processo cliente o client) invia la richiesta a un processo servitore (server), che effettua il lavoro e invia indietro la risposta.
In questo modello, mostrato in figura 8, il nucleo deve solo gestire la comunicazione tra clienti e servitori. Dividendo il sistema operativo in parti, ognuna delle quali gestisce solo un aspetto del sistema, come il file service, il process service, il terminal service o il memory service, ogni parte diviene piccola e maneggevole. Inoltre, poiché tutti i servitori vengono eseguiti come processi utente, e non in modalità di nucleo, essi non hanno accesso diretto all'hardware.
Di conseguenza, se un difetto nel file server causa un guasto, il servizi relativi ai file possono interrompersi, ma in genere questo non provocherà l'arresto dell'intera macchina.
Un altro vantaggio del modello cliente-servitore è la sua adattabilità all'utilizzo in sistemi distribuiti (si guardi la figura 9). Se un cliente comunica con un servitore inviandogli messaggi, il cliente non ha bisogno di sapere se il suo messaggio venga gestito localmente, sulla sua macchina, o se esso sia inviato attraverso la rete a un servitore su di una macchina remota.



Per quanto concerne il cliente, in entrambi i casi accade lo stesso: una richiesta viene inviata e una risposta restituita.

Figura 8 - Modello Client - Server

Il quadro fornito in precedenza di un nucleo che gestisce solo il traffico di messaggi dai clienti ai server e viceversa non è realistico. Alcune funzioni del sistema operativo (come il caricamento dei comandi sui registri dell'interfaccia fisica di I/O) sono difficili, se non impossibili da effettuare per i programmi nello spazio utente. Ci sono due modi di affrontare questo problema. Un modo è far sì che alcuni processi servitori critici (come i device driver di I/O) vengano eseguiti in modalità di nucleo, cioè con la possibilità di accedere direttamente a tutto l'hardware, e possano comunicare con gli altri processi mediante i normali meccanismi a messaggi.

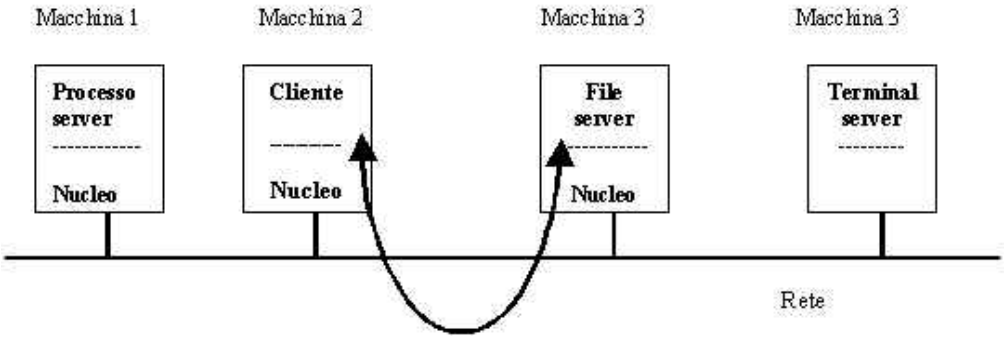


Figura 9 - Sistema distribuito

L'altro metodo consiste nel programmare nel nucleo una quantità minima di meccanismi, lasciando le politiche di gestione ai server nello spazio utente. Ad esempio, il nucleo potrebbe riconoscere che un messaggio inviato a un particolare indirizzo implica prendere il contenuto di quel messaggio e caricarlo nei registri dell'interfaccia di I/O di un disco, per iniziare una lettura dal disco stesso. In questo esempio, il nucleo non andrebbe neanche a guardare i byte contenuti nel messaggio per controllare che essi siano validi o significativi; esso dovrebbe copiarli direttamente nei registri di interfaccia del disco (ovviamente bisogna usare qualche schema per limitare l'uso di tali messaggi ai processi autorizzati). La separazione tra meccanismo e politiche è un concetto importante; esso ricorre più e più volte e in contesti diversi nei sistemi operativi.

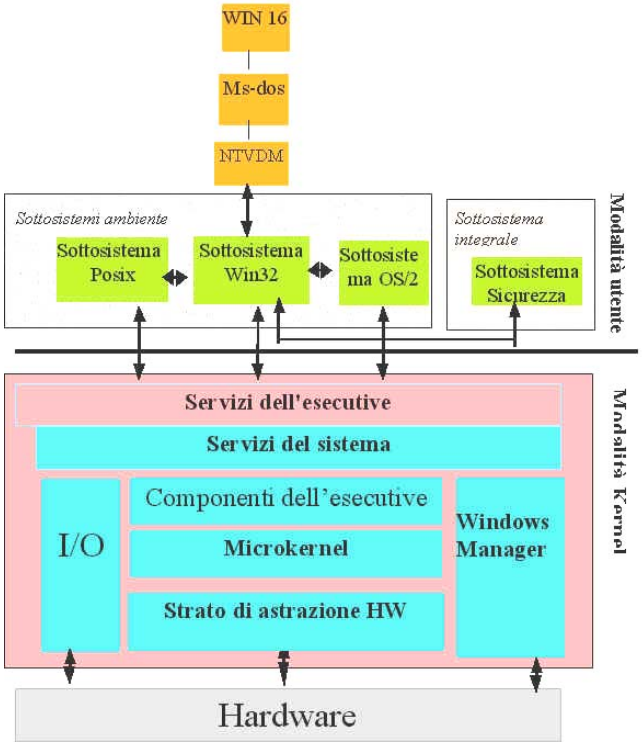


Figura 10 - L'architettura di Windows NT

1.4 IL SISTEMA OPERATIVO IPOTETICO

La maggior parte del presente corso sarà dedicata ai sistemi universali. Tuttavia, parecchi dei problemi esaminati si verificano anche in altri sistemi e, perciò, a seconda del caso, forniremo indicazioni circa la pertinenza della nostra trattazione anche ad altri sistemi.
Al fine di studiare il metodo secondo il quale è costruito un sistema universale, realizzeremo un sistema operativo "su carta"; svilupperemo, cioè, su carta un sistema ipotetico, nella speranza di illustrare così i principi coinvolti. Sarà nostro obiettivo, pure, sviluppare tale sistema in modo logico, in quanto ogni stadio del nostro sviluppo risulterà naturale conseguenza del precedente. La struttura finale in un certo qual modo assomiglierà ad una cipolla, nella quale ogni strato ingloba un insieme di funzioni che dipendono solo dagli strati più interni. La direzione del nostro processo costruttivo andrà dall'interno verso l'esterno, ovverosia sarà di tipo "bottom-up", cosicché, partendo da un nucleo interno, aggiungeremo strati successivi per la gestione della memoria, del I/O, per l'accesso ai file, e così via.
Procedendo nello sviluppo del nostro sistema operativo ipotetico, lo raffronteremo con i sistemi esistenti e porremo in relazione le tecniche da noi usate con quelle impiegate altrove. In tal modo manterremo saldamente radicato nella prassi corrente il nostro sistema operativo, usandolo al contempo come veicolo per trasmettere i principi costruttivi dei sistemi operativi.

1.5 RIEPILOGO

I sistemi operativi possono essere esaminati secondo due punti di vista: come gestori di risorse o come macchine virtuali. Secondo il primo, il compito di un sistema operativo è di gestire in modo efficiente le diverse parti del sistema. Come macchina estesa, il compito del sistema è viceversa quello di fornire agli utenti una macchina virtuale che sia più comoda da usarsi rispetto alla macchina reale.
I sistemi operativi hanno una lunga storia, a partire dai giorni in cui essi sostituivano l'operatore, fino ai moderni sistemi multi programmati.
Il cuore di ogni sistema operativo è l'insieme di chiamate di sistema che esso riesce a gestire. Esse indicano ciò che realmente fa il sistema operativo
I sistemi operativi possono essere strutturati in diversi modi. I più comuni sono come sistema monolitico, come gerarchia di strati, come sistema a macchine virtuali e infine usando il modello cliente-servente.

2 FUNZIONI E CARATTERISTICHE DI UN SISTEMA OPERATIVO

Nel presente capitolo prenderemo in esame più dettagliatamente le funzioni che ci attendiamo di vedere svolte da un sistema operativo ed enucleeremo alcune delle caratteristiche che esso deve possedere.

2.1 LE FUNZIONI DEL SISTEMA OPERATIVO

Come punto di partenza prendiamo in considerazione la macchina base, cioè l'hardware fondamentale di qualsiasi computer, formata da unità centrale di elaborazione, memoria e varie unità periferiche (rimandiamo più avanti la trattazione di computer con più di un'unità centrale di elaborazione). In assenza di qualsiasi risorsa software di assistenza, l'operazione di caricamento ed esecuzione di un programma applicativo deve essere eseguita interamente tramite un operatore. Supponendo di avere a che fare con un funzionamento di tipo batch, l'operatore deve eseguire una noiosa sequenza di compiti, pressappoco simile alla seguente:

- sistemare il programma sorgente nel dispositivo di input;
- avviare un programma per la lettura delle istruzioni;
- avviare un compilatore per compilare il programma sorgente;
- avviare l'esecuzione del programma compilato;
- introdurre i dati nell'unità di input;
- ritirare i risultati ottenuti dalla stampante di linea.

La velocità della macchina, naturalmente, è condizionata dalla rapidità con cui l'operatore è in grado di premere i tasti necessari e di alimentare le unità periferiche e, pertanto, il sistema incontra enormi limitazioni dovute all'operatore.

Un primo passo avanti lo si ottiene chiaramente facendo sì che la macchina svolga automaticamente, senza l'interessamento dell'operatore, la sequenza di interventi necessari alla lettura, compilazione, caricamento ed esecuzione dei programmi. In tal modo il ruolo svolto dall'operatore viene limitato, ad un capo della macchina, all'avvio di un programma di lettura ed, all'altro, a staccare fogli di carta, aumentando di conseguenza la produttività del computer. Naturalmente, senza sacrifici raramente si ottiene qualcosa ed il prezzo che si paga in questo caso per ottenere una velocità maggiore consiste nel dover dedicare parte della macchina ad un programma in grado di portare a termine le operazioni necessarie per ordinare i lavori in sequenza. Dal momento che non tutti i lavori hanno bisogno delle stesse operazioni di ordinamento sequenziale (ad esempio, per alcuni lavori può essere superflua la compilazione) il programma di controllo deve essere in grado di dedurre da una o più istruzioni di controllo esattamente quali sono le operazioni da svolgere per ogni caso specifico. In altri termini, il programma di controllo deve essere in grado di interpretare un job control language. Inoltre, non si può lasciare che gli errori, inevitabili in certi lavori, influiscano su lavori successivi e quindi il programma di controllo deve assumersi il compito di gestire gli errori razionalmente. Un programma di controllo con queste caratteristiche in realtà è un sistema operativo in nuce.

Un sistema di tal genere trova chiaramente limitazioni nella velocità con cui riesce ad eseguire le operazioni di I/O ed il passo successivo per migliorarne l'efficienza consiste nel ridurre la sua dipendenza dall'I/O. Da un punto di vista storico, il primo metodo con cui si ottenne questo risultato fu l'off-lining (fuori linea) tramite il quale tutte le operazioni di input ed output avvenivano per mezzo di nastri magnetici. Il compito di trasferire su nastro magnetico l'input contenuto nelle schede (o nel nastro di carta) e quello di trasformare l'output registrato su nastro magnetico in informazioni per la stampante di linea o per il perforatore venivano relegati ad un calcolatore ausiliario di minor costo e potenza. Il trasferimento dei nastri magnetici tra il computer centrale e l'ausiliario avveniva manualmente. I sistemi basati sul principio dell'off-lining (tra i quali segnaliamo l'IBM Fortran Monitor System) funzionarono egregiamente a partire dagli ultimi anni '50 fino alla metà degli anni '60.

L'off-lining diminuiva la dipendenza dall'I/O, assicurando che tutte le operazioni relative venissero eseguite su un nastro magnetico che, all'epoca, rappresentava il mezzo più veloce reperibile al minor costo. Il prezzo che si pagava era rappresentato dall'aggiunta al sistema operativo in nuce di una serie di routine per codificare ed immagazzinare su nastro magnetico dei dati che, in ultima istanza, erano destinati ad una qualche altra unità periferica (e, per contro, sotto forma di input). Si noti anche che, siccome il nastro magnetico essenzialmente è un mezzo di tipo seriale, non si pensava affatto all'eventualità di poter disporre i lavori da eseguire in una successione diversa da quella in cui si presentavano.

Nell'intento di eliminare la dipendenza dall'I/O, anziché ridurla semplicemente, si devono impiegare procedimenti che permettano di sovrapporre le operazioni di I/O all'elaborazione, il che risulta possibile con l'ausilio di due componenti hardware: il canale e l'interrupt. Un canale è un dispositivo che controlla una o più unità periferiche e che esegue i trasferimenti di dati tra queste ultime e la memoria in modo del tutto autonomo rispetto all'unità centrale di elaborazione. Un interrupt è un segnale che trasferisce il controllo dell'unità centrale d'elaborazione ad una locazione fissa, memorizzando al contempo il valore precedente relativo al contatore di programma. In tal modo, il programma in corso d'esecuzione al momento dell'interrupt viene abbandonato temporaneamente, ma può essere ripreso in seguito (è, cioè, interrotto). Un interrupt proveniente da un canale funge da segnale indicante che è stato completato un trasferimento di dati. È quindi possibile per l'unità centrale di elaborazione iniziare un trasferimento tra unità periferiche, continuare l'elaborazione mentre il canale controlla il trasferimento e ricevere un segnale d'avviso, tramite l'interrupt, quando il trasferimento è completo. (Si rimanda lo studente, desideroso di maggiori conoscenze sull'hardware relativo ai canali ed all'interrupt, ad un qualsiasi testo sulla struttura dei computer, quali Foster, 1970, oppure Stone, 1975).

A questo punto diventa possibile memorizzare i lavori su un qualche supporto idoneo di informazione, solitamente un disco, ed eseguirli uno ad uno contemporaneamente alla memorizzazione di altri lavori. Per quanto riguarda gli ampliamenti del nostro sistema operativo in rapida crescita, si tratta di introdurre una routine che gestisca gli interrupt ed un'altra routine che decida qual è il prossimo lavoro da eseguire tra i tanti presenti sul disco. Quest'ultima funzione, quella di schedulazione, è resa possibile dall'impiego del disco quale mezzo di input, poiché in questo caso si tratta di un dispositivo ad accesso casuale, contrariamente al nastro magnetico che è di tipo seriale. Un sistema operante in questo modo viene definito single stream batch monitor, dove "single stream" (flusso singolo) sta ad indicare che viene eseguito un solo lavoro per volta. Tali sistemi erano preponderanti verso la metà degli anni '60.

Lo svantaggio principale di un sistema a flusso singolo è che l'intera macchina viene dedicata all'esecuzione di un solo lavoro, non importa di quale entità. Si riesce a superare tale svantaggio con la multiprogrammazione, la messa in esecuzione di parecchi programmi nella stessa macchina ed allo stesso tempo. L'idea di fondo consiste nel tenere in memoria svariati programmi simultaneamente, mentre il processore centrale suddivide tra di essi il proprio tempo disponibile in base alle risorse (quali, ad esempio, i canali o le periferiche) necessarie a ciascuno di essi in ogni dato istante. In tal modo è possibile, disponendo sempre in macchina di una varietà idonea di lavori, ottenere un'utilizzazione ottimale delle risorse, ivi compresa l'utilizzazione dell'unità centrale d'elaborazione, in quanto, ogniqualvolta un lavoro si trova in attesa del completamento di un trasferimento di I/O, l'elaboratore può passare all'esecuzione di qualche altro lavoro presente in macchina. L'onere aggiuntivo che grava sul sistema operativo è il controllo delle risorse e la protezione di un lavoro dalle attività relative ad un altro. Un sistema operativo di questo tipo viene definito multi-stream batch monitor, in alcune versioni ampiamente presente in tutti i grossi elaboratori dei primi anni settanta (per esempio il GEORGE3 della serie 1CL 1900 e l'OS/360 della serie IBM 360 e 370). La realizzazione di questi sistemi deve molto al lavoro d'avanguardia svolto all'inizio degli anni sessanta sul calcolatore Atlas presso l'Università di Manchester.

A questo punto della nostra progressione, partita dalla macchina pura e semplice, ci ritroviamo con un sistema piuttosto sofisticato che fa buon uso dell'hardware disponibile. Il suo svantaggio principale, dal punto di vista dell'utente, è la totale mancanza di interazione con il lavoro mentre questo gira sull'elaboratore. L'interazione è preziosa in molte situazioni ed, in particolare, durante la realizzazione ed il debugging di un programma. A titolo d'esempio, il debugging viene facilitato se il programmatore è in grado, da subito, di provare le modifiche introdotte e vederne l'effetto, apportando, se necessario, ulteriori cambiamenti. Analogamente, molti programmi mirati alla risoluzione dei problemi richiedono un "instradamento" da parte dell'utente, ovverosia l'utente determina l'intervento successivo del programma dopo aver preso in esame i risultati ottenuti fino a quel momento. Affinché la strada dell'interazione diventi praticabile, il sistema di tipo batch a flusso multiplo deve essere modificato in modo da accettare gli input provenienti da utenti al lavoro sulle tastiere dei terminali remoti; in altre parole, deve diventare un sistema ad accesso multiplo. Nei primi anni settanta, la maggior parte delle industrie di questo settore produceva sistemi operativi ad accesso multiplo per i calcolatori di più vaste dimensioni: citiamo ad esempio il MINIMOP (1CL 1900), il TSS/360 (IBM 360 e 370) ed il TOPS-IO (DEC System-10). Verso la metà degli anni settanta furono combinati assieme il funzionamento di tipo batch e quello ad accesso multiplo per sistemi quali l'MVS (IBM 370), l'MCP (Burroughs 6700 e 7700) ed il VME (1CL 2900).
La discussione precedente ci ha portati verso un sistema operativo che deve almeno eseguire le seguenti funzioni:

1. pianificazione del job;
2. interpretazione del job control language;
3. gestione degli errori;
4. gestione dell'I/O;
5. gestione degli interrupt;
6. schedulazione;
7. controllo delle risorse;
8. protezione;
9. accesso multiplo.

Inoltre, il sistema operativo dovrebbe risultare semplice da mandare in esecuzione, dal punto di vista dell'utente, e semplice da controllare, dal punto di vista dell'addetto all'installazione. Pertanto, possiamo aggiungere queste altre funzioni:

10. disponibilità di una buona interfaccia con l'operatore;
11. contabilizzazione delle risorse di calcolo.

Da quanto è stato detto sopra, risulterà chiaro che i sistemi operativi decisamente non sono "fenomeni" irrilevanti e non è, pertanto, sorprendente che ci si sia prodigati in sforzi enormi per scriverli. Come passo ulteriore verso la strutturazione coerente del sistema operativo, dall'elenco summenzionato di funzioni possiamo ricavare certe caratteristiche che il sistema deve presentare. Queste, unitamente ai problemi che si accompagnano alla loro evidenziazione, vengono elencate nel paragrafo successivo.

2.1.1 Concorrenza

La concorrenza dipende dall'esistenza di parecchie attività simultanee, o parallele. Esempi sono la sovrapposizione delle operazioni di I/O con quelle di calcolo e la copresenza in memoria di svariati programmi applicativi. La concorrenza porta con sé il problema di passare dall'una all'altra attività, di proteggere l'una dagli effetti dell'altra e di sincronizzare le attività che sono in rapporto di dipendenza reciproca.

2.1.2 Condivisione

Può risultare necessario che le attività concorrenti condividano risorse o informazioni, essenzialmente per quattro ordini di motivi.

a. Il costo: risulta eccessivamente oneroso fornire separatamente risorse sufficienti per tutti gli utenti.
b. Valorizzazione del lavoro altrui: è utile poter impiegare i programmi e le routine di altri.
c. Condivisione dei dati: può risultare necessario utilizzare lo stesso data base per svariati programmi differenti, se possibile operativi per svariati utenti distinti.
d. Eliminazione del superfluo: è economicamente conveniente condividere un'unica copia di un programma (per esempio, un compilatore) tra parecchi utenti, anziché dover fornire una copia distinta per ciascun utente.

I problemi connessi con la condivisione sono l'allocazione delle risorse, l'accesso simultaneo ai dati, l'esecuzione simultanea di programmi e la protezione dalla contaminazione degli errori altrui.

2.1.3 Memorizzazione a lungo termine

L'esigenza di condividere programmi e dati porta con sé anche la necessità di memorizzare le informazioni a lungo termine. Ciò consente, inoltre, all'utente di sfruttare la comodità di tenere i propri programmi o i propri dati nell'elaboratore, anziché dover ricorrere ad un qualche supporto esterno (per esempio, schede). I problemi che sorgono riguardano l'accesso facile, la protezione dalle ingerenze altrui (intenzionali o di altra natura) e la protezione da eventuali gravi malfunzionamenti del sistema.

2.1.4 Determinatezza

Un sistema operativo deve essere determinato nel senso che mettendo in macchina lo stesso programma, oggi o domani, a parità di dati si dovrebbero ottenere i medesimi risultati. D'altro canto è indeterminato, in quanto deve far fronte ad avvenimenti che si succederanno in un ordine imprevedibile. Si tratterà di avvenimenti quali richieste di assegnazione di risorse, errori nella fase esecutiva dei programmi ed interrupt provenienti da unità periferiche. A causa dell'enorme numero di evenienze fortuite che possono aver luogo, non è ragionevole pensare di poter scrivere il sistema operativo in modo tale che provveda a tutte singolarmente; al contrario, si deve scrivere il sistema così da gestire qualsiasi successione di eventi.
Vale la pena mettere in rilievo che nessuna di queste caratteristiche è specifica dei soli sistemi universali. Ad esempio, la capacità di mantenere in memoria dati a lungo termine è chiaramente necessaria anche per i sistemi a consultazione di file, mentre la concorrenza è una delle caratteristiche primarie dei sistemi per il trattamento delle transazioni.
In conclusione del presente capitolo, faremo un breve accenno ad alcune caratteristiche che è consigliabile siano presenti nei sistemi operativi universali.

2.2 REQUISITI AUSPICABILI

2.2.1 Efficienza

Si è già fatto cenno alle esigenze di efficienza. Purtroppo, non è semplice illustrare un criterio univoco in base al quale poter valutare l'efficienza di un sistema operativo, I vari criteri possibili sono elencati qui di seguito:

a. tempo medio intercorrente tra i job;
b. tempo di CPU non utilizzato;
c. tempo d'esecuzione per i job di tipo batch;
d. tempo di risposta (nei sistemi ad accesso multiplo);
e. utilizzazione delle risorse;
f. produttività (numero di job all'ora).

Non si possono soddisfare tutti questi criteri contemporaneamente; avremo modo di ritornare sull'argomento nei capitoli successivi a proposito di quelle decisioni di carattere progettuale che danno effettivamente maggior peso ad alcuni di questi criteri che non ad altri.

2.2.2 Affidabilità

In linea di principio, un sistema operativo dovrebbe essere totalmente privo d'errori e dovrebbe essere in grado di gestire tutti gli eventi. Nella pratica questo non avviene mai, ma vedremo in seguito in che modo sia possibile fare progressi notevoli in questa direzione.

2.2.3 Facilità di manutenzione

Dovrebbe risultare possibile effettuare interventi di manutenzione su un sistema operativo (sia migliorandolo, che correggendone gli errori) senza dover impiegare un esercito di sistemisti. Da qui deriva la necessità che il sistema presenti una costruzione modulare, con interfacce tra i moduli chiaramente definite, e che sia dotato di una documentazione esauriente.

2.2.4 Dimensioni ridotte

Lo spazio utilizzato per contenere il sistema operativo, sia esso in memoria centrale che su memoria di massa, è sprecato per quanto riguarda le attività di calcolo produttive. Inoltre, rispetto ad un sistema di dimensioni ridotte, in uno più voluminoso tendono maggiormente a verificarsi situazioni d'errore ed, ovviamente, ci vuole più tempo per scriverlo.

2.3 RIEPILOGO

Per riassumere gli argomenti affrontati nel presente capitolo, ricordiamo che sono state introdotte le funzioni generiche che prevediamo verranno svolte dal sistema operativo, di averne messo in evidenza alcune caratteristiche fondamentali e di aver passato in rassegna alcuni requisiti auspicabili per un sistema. Nel prossimo capitolo esamineremo alcuni strumenti basilari che ci saranno d'aiuto nella costruzione del nostro sistema.

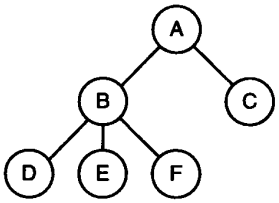
3 PROCESSI CONCORRENTI

Prima di poter iniziare lo studio dettagliato dei sistemi operativi, è necessario introdurre alcuni concetti basilari e sviluppare alcuni strumenti. Questo sarà lo scopo del presente capitolo.

3.1 PROGRAMMI, PROCESSI E PROCESSORI

Cominciamo col considerare un sistema operativo come un insieme di attività, ciascuna delle quali fornisce una delle funzioni descritte nel capitolo 2, come, ad esempio, la schedulazione o la gestione dell'I/O. Ogni attività consiste nell'esecuzione di uno o più programmi e la si richiama ogniqualvolta si dovrà fornire la funzione corrispondente. Usiamo il termine processo per descrivere un'attività di tal genere (altre denominazioni presenti nella letteratura specializzata sono task e computazione). Si può così pensare ad un processo come ad una sequenza di azioni, svolte tramite l'esecuzione di una sequenza di istruzioni (un programma) il cui risultato netto è quello di fornire una delle funzioni di sistema. Possiamo allargare il concetto fino ad abbracciare anche le funzioni di utenza oltre a quelle di sistema, cosicché anche l'esecuzione di un programma applicativo verrà definita processo.
Un processo può comportare l'esecuzione di più di un programma; per contro, un singolo programma o routine può essere coinvolto in più di un processo. Per esempio, una routine idonea ad aggiungere singole voci ad un elenco può essere usata da qualsiasi processo impegnato nella manipolazione delle code d'attesa. Pertanto, il fatto di sapere che un programma specifico è in corso d'esecuzione non ci dice molto sul tipo di attività che viene portata avanti, né su quale funzione sia in corso di implementazione. È per questo motivo, in larga misura, che il concetto di processo, parlando dei sistemi operativi, è più utile di quello di programma.
Un processo è in grado di proseguire in virtù di un agente che segue il programma relativo. Tale agente è noto sotto il nome di processore. Diciamo che un processore esegue un processo o che un processo è in esecuzione su un processore. Quest'ultimo è un qualcosa che segue istruzioni; a seconda della natura delle istruzioni, il processore può essere realizzato tramite il solo hardware oppure con una combinazione di hardware e software. Ad esempio, un'unità centrale d'elaborazione (CPU) è un processore atto all'esecuzione delle istruzioni in linguaggio macchina, mentre una CPU assieme ad un interprete BASIC possono formare un processore per l'esecuzione di istruzioni in BASIC. La trasformazione (trattata nel capitolo 1) di un elaboratore puro e semplice in una macchina virtuale in realtà può essere vista come la combinazione dell'hardware proprio dell'elaboratore con il sistema operativo, alfine di mettere a disposizione dell'utente un processore capace di eseguire i processi applicativi (ovvero, un processore capace di eseguire le istruzioni dategli dall'utente).
Il programma o i programmi associati ad un processo non necessitano di essere implementati sotto forma di software. L'intervento di un canale, ad esempio, nello svolgimento di un trasferimento di dati, può essere considerato come un processo nel quale il programma relativo viene cablato ed inserito nell'hardware del canale stesso. Visto in tal senso, il canale, ma a dire il vero anche una qualsiasi periferica, rappresenta un processore in grado di eseguire soltanto un unico processo.

Si può facilmente fare uso dei concetti di processo e processore per interpretare sia la concorrenza che la non determinatezza, due delle caratteristiche del sistema operativo esposte nel capitolo precedente.
La concorrenza può essere considerata come l'attivazione di svariati processi (e cioè l'esecuzione di svariati programmi) in contemporanea. A condizione che vi siano tanti processori quanti sono i processi, ciò non presenta alcuna difficoltà da un punto di vista logico. Se, però, come accade più spesso, vi sono meno processori che processi, si può ottenere una concorrenza virtuale dirottando i processori da un processo all'altro. Se tale commutazione viene effettuata ad intervalli di tempo adeguatamente brevi, il sistema darà l'illusione della concorrenza, quando lo si prenda in considerazione sulla base di tempi macroscopici. La concorrenza virtuale si raggiunge all'atto pratico tramite la tecnica dell'interfogliazione dei processi su un unico processore.
Può risultare utile, a questo punto, fare un'analogia con le attività di una segretaria in un ufficio generico. Ciascuno dei lavori che la segretaria deve svolgere, ad esempio la battitura di lettere, l'archiviazione delle fatture o il prendere appunti sotto dettatura, può essere associato ad un processo di un sistema operativo. Il processore è rappresentato dalla segretaria stessa e la sequenza di istruzioni che definiscono ciascun lavoro è analoga ad un programma. Se nell'ufficio c'è molto da fare, può darsi che la segretaria debba sospendere un lavoro per svolgerne un altro e, in tale situazione, probabilmente si lamenterà poiché "deve fare troppi lavori contemporaneamente". Naturalmente, si troverà a fare un solo lavoro alla volta, ma l'impressione complessiva data dalla frequenza con cui passa da un'attività ad un'altra sarà quella della concorrenza. Possiamo spingere oltre l'analogia osservando che, prima che la segretaria possa dedicarsi ad un'altra attività, deve prendere nota in modo adeguato di ciò che sta facendo in quel momento, per essere in grado di non perdere poi il filo. Similmente, un processore in un sistema di calcolo deve registrare informazioni sul processo che sta eseguendo in un dato momento, prima di poter passare ad un qualche altro processo. La natura esatta dell'informazione da memorizzare sarà oggetto di trattazione nel capitolo successivo; per il momento ci limitiamo ad osservare che deve essere sufficiente per consentire al processo di essere ripreso in un tempo successivo.
Portando l'analogia ancora oltre, possiamo inserire nell'ambito dell'ufficio altre segretarie (processori). Ora può esistere veramente la concorrenza tra le varie segretarie che svolgono compiti diversi, sebbene possano permanere ancora situazioni di concorrenza virtuale, in quanto ogni segretaria passa da un compito all'altro, per tornare poi al compito di partenza. Solo quando il numero delle segretarie è pari a quello dei lavori da svolgere è possibile effettuarli tutti in maniera veramente concorrente. L'unità analoga ad un processore, quale un dispositivo di I/O, che riesce a svolgere soltanto un unico processo, rappresenta la segretaria più giovane che è capace solo di fare il caffè.
Associato ad ogni processo vi è il suo spazio di indirizzamento, cioè una lista di locazioni di memoria comprese tra un minimo (solitamente 0) e un massimo, che il processo può usare per letture e scritture. Lo spazio di indirizzamento contiene il programma eseguibile, i dati del programma e il suo stack (o pila). Associato a ogni processo vi è inoltre un insieme di registri che comprende il program counter, il puntatore allo stack, e altri registri hardware, nonché tutte le altre informazioni necessarie all'esecuzione del programma.
Periodicamente il sistema operativo decide di fermare l'esecuzione di un processo e di iniziarne un altro, ad esempio perché il primo ha consumato, nel periodo trascorso, il tempo di CPU a sua disposizione. Se un processo viene temporaneamente sospeso, esso deve poter essere ripreso esattamente nelle stesse condizioni nelle quali si trovava al momento della sua sospensione. Questo significa che tutte le informazioni relative al processo devono essere esplicitamente salvate da qualche parte durante la sua sospensione. Ad esempio, il processo potrebbe avere diversi file aperti in lettura. A ognuno di questi file viene associato un puntatore contenente la posizione corrente nel file (cioè l'indice del byte o record che deve essere letto in sequenza). Quando un processo viene temporaneamente sospeso tutti questi puntatori devono essere salvati, in modo che una chiamata ad un'apposita routine, eseguita dopo il ripristino del processo, legga i dati corretti. In molti sistemi operativi tutte le informazioni su ogni processo, ad eccezione del contenuto del suo spazio di indirizzamento, vengono immagazzinate in una tabella di sistema chiamata la tabella dei processi (process-table), che è un array (o una lista) di strutture, una per ogni processo esistente.
Di conseguenza, un processo (sospeso) consta del suo spazio di indirizzamento, solitamente chiamato core image (o immagine di nucleo, in onore delle memorie a nucleo magnetico usate un tempo), e della sua voce nella tabella dei processi, contenente, tra le altre cose, i suoi registri.
Le chiamate di sistema fondamentali per la gestione dei processi sono quelle che si occupano della creazione e terminazione dei processi. Si consideri un esempio tipico. Un processo chiamato interprete dei comandi o shell legge comandi da un terminale. L'utente ha appena inserito un comando per richiedere la compilazione di un programma. La shell deve quindi creare un nuovo processo che eseguirà il compilatore. Quando la compilazione ha termine il processo eseguirà una chiamata di sistema per terminare.
Se un processo può creare uno o più altri processi (chiamati processi figli o child) e questi processi possono a loro volta creare processi figli, si arriva rapidamente alla struttura ad albero Processi correlati, che devono cooperare per conseguire il loro scopo, necessitano spesso di comunicare l'uno con l'altro e di sincronizzare le loro attività. Questa comunicazione viene chiamata comunicazione interprocesso, e sarà trattata in dettaglio più avanti.



Processi padre e figli

Altre chiamate di sistema possono essere usate per richiedere più memoria (o rilasciare memoria inutilizzata), attendere la terminazione di un processo figlio, e sovrascrivere il programma corrente con uno diverso. A volte c'è bisogno di comunicare informazioni ad un processo in esecuzione che non è in attesa di riceverle. Ad esempio, un processo che comunica con un altro processo su di un computer diverso può inviare messaggi attraverso la rete di comunicazione. Per proteggersi dalla possibilità che un messaggio o la risposta ad esso possa andare perduta, il mittente può richiedere al suo sistema operativo di essere avvertito dopo un determinato numero di secondi, in modo da poter ritrasmettere il messaggio s nessuna conferma di ricezione (acknowledgement) è pervenuta nel frattempo. Dopo aver impostato questo timer, il programma può continuare tranquillamente le sue elaborazioni.
Quando il numero specificato di secondi è trascorso il sistema operativo invia un segnale al processo. Il segnale fa sì che il processo sospenda temporaneamente qualunque cosa stesse facendo al momento, salvi i suoi registri nello stack, e inizi ad eseguire una speciale procedura di gestione dei segnali, ad esempio per ritrasmettere il messaggio presumibilmente perduto. Quando il gestore del segnale termina, il processo in esecuzione viene ripristinato nello stato in cui si trovava immediatamente prima del segnale.
I segnali sono l'analogo software delle interruzioni hardware, e possono essere generati da un certo numero di cause, oltre che per la scadenza di un timer. Molte eccezioni (trap) intercettate dall'hardware, come l'esecuzione di un'istruzione illegale o l'uso di un indirizzo non valido, sono convertite in segnali diretti al processo responsabile.
Tenendo presente quanto esposto in antecedenza, definiamo l'elaborazione concorrente (o elaborazione parallela) osservando che, se si scatta un'istantanea del sistema nel suo complesso, si possono trovare svariati processi in diversi

stati compresi tra il punto d'inizio e quello d'arrivo. Questa definizione chiaramente abbraccia sia la concorrenza vera e propria che quella virtuale.
La non determinatezza è la seconda caratteristica del sistema operativo che si può descrivere facilmente in termini di processi. Se consideriamo i processi come sequenze di azioni interrompibili tra un passo e l'altro, la non determinatezza si riflette nell'ordine imprevedibile secondo il quale procedono le sequenze. Ritornando alla nostra analogia riferita alle segretarie, possiamo paragonare questi eventi imprevedibili che si verificano entro un sistema operativo ai telefoni di un ufficio che squillano. Non è possibile sapere in anticipo quando suonerà un certo telefono, quanto ci vorrà per sbrigare la chiamata o che tipo di effetto avrà la telefonata sui vari lavori in corso di svolgimento nell'ufficio. Tuttavia, possiamo osservare che la segretaria che va a rispondere al telefono deve ricordarsi di quel che sta facendo in quel momento, per potervi poi ritornare sopra in seguito. Allo stesso modo, l'interruzione di un processo deve essere accompagnata dalla registrazione di informazioni tali da consentire successivamente la riattivazione del processo stesso. Le informazioni registrate in questo caso sono uguali a quelle necessarie quando i processori vengono commutati tra i vari processi col fine di creare situazioni di concorrenza. A dire il vero, l'interruzione di un processo può essere considerata come la commutazione, causata da qualche imprevisto, di un processore che viene distolto da un processo.
Possiamo riassumere nel modo seguente ciò che si è detto finora. Un processo è una sequenza di azioni e perciò è dinamico, mentre un programma è una sequenza di istruzioni e perciò è statico. Un processore è quell'agente che serve a mettere in esecuzione un processo. La non determinatezza e la concorrenza possono essere descritte, rispettivamente, in termini di interruzioni di processi tra due azioni consecutive e come commutazione di processori tra diversi processi. Alfine di effettuare tali interruzioni e commutazioni, si devono memorizzare informazioni sul processo in questione sufficienti perché possa essere ripreso successivamente.
Per fornire un esempio di multiprocesso consideriamo un sistema di acquisizione dati, che ripetutamente esegue queste attività.

```
Program acquisizione_dati_sequenziale
.................
begin
  while true do [loop infinito]
    begin
      Acquisisci; [acquisizione dal convertitore A/D]
      Registra; [memorizza i dati su disco]
      Statistica; [elaborazione statistica]
      Print; [stampa i risultati]
    end [while]
end [acquisizione_dati_sequenziale]
```

Le attività previste da questo programma hanno orientamenti diversi:
A1 – Input
R1 – I/O
S1 – Calcola
P1 -- Output

Quindi sono candidate per essere eseguite in modo concorrente. Possiamo iniziare una fase di acquisizione mentre stampiamo i risultati dell'elaborazione precedente, oppure possiamo registrare i dati su disco mentre elaboriamo i dati dell'acquisizione precedente.
Sulla base di quanto detto si possono codificare le quattro fasi di attività sotto forma di processi separati.
Per completare l'esempio si può fare l'ipotesi che esistano due buffer globali (accessibili a tutti i processi, IB1 per l'input ed OB1 per l'output. Su IB1 "Acquisisce" deposita i dati e "Registra" e "Statistica" leggono, mentre su OB1 "Statistica" deposita i dati che "Print" dovrà inviare alla stampa. Abbiano dovuto introdurre nei vari processi un meccanismo di segnalazione che consenta di sincronizzare i processi.

```
Program/modulo acquisizione_dati_multiprocesso
........
process acquisisci;
begin
  while true do [loop infinito]
    begin
      aspetta-segnale_da(registra, statistica)
      Acquisisci; [acquisizione dal convertitore A/D]
      Invia-segnale-a(registra, statistica)
    end [while]
end [acquisisci]

process registra;
begin
  while true do [loop infinito]
    begin
      aspetta-segnale_da(acquisisci)
      Registra; [memorizza i dati su disco]
      Invia-segnale-a(acquisisci)
    end [while]
end [registra]

process statistica;
begin
  while true do [loop infinito]
    begin
      aspetta-segnale_da(acquisisci, print)
      Statistica; [elaborazione statistica]
```

```
        Invia-segnale-a(print, acquisisci)
      end [while]
end [statistica]

process print;
begin
   while true do [loop infinito]
      begin
         aspetta-segnale_da(statistica)
         Print; [stampa i risultati]
         Invia-segnale-a(statistica)
      end [while]
end [print]

[processo padre]
begin [acquisizione_dati_multiprocesso]
   Inizializza_ambiente;
   Invia-segnale-a(acquisisci, acquisisci, statistica); [prima attivazione
   dei processi figli]
   Inizializza acquisisci, registra, statistica, print
end [acquisizione_dati_multiprocesso]
```

Schematizziamo con un grafo l'esecuzione sequenziale e quella concorrente.

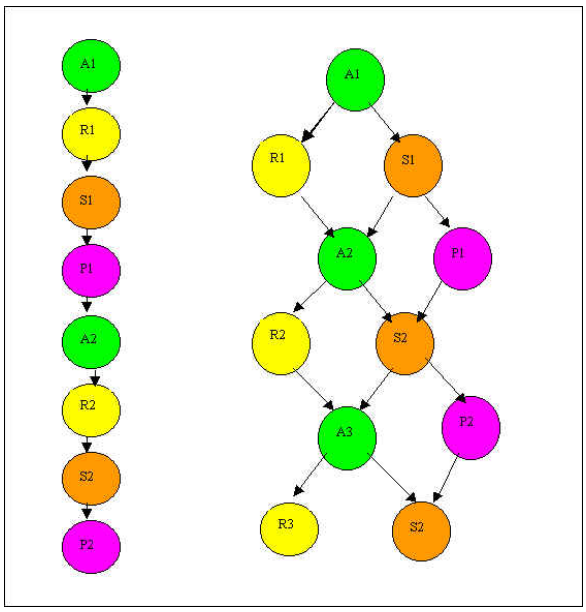


Figura 11 - Esempio di multiprocesso

3.2 LA COMUNICAZIONE TRA I PROCESSI

I processi all'interno di un sistema di calcolo, quindi, non agiscono separatamente l'uno dall'altro. Se da un lato devono collaborare per raggiungere l'obiettivo voluto, e cioè quello di eseguire i job introdotti dall'utente, dall'altro si trovano in competizione per l'uso di risorse limitate, quali processori, memoria o file. Questi due elementi di collaborazione e competizione comportano la necessità di una qualche forma di comunicazione tra i processi.
I campi in cui la comunicazione è essenziale possono essere classificati come segue.

3.2.1 Mutua esclusione

Le risorse di un sistema possono essere classificate come condivisibili, intendendo che possono essere usate da svariati processi in situazioni di concorrenza (nel senso definito nel paragrafo precedente), o come non condivisibili, intendendo che il loro utilizzo è limitato ad un processo per volta. La non condivisibilità di una risorsa deriva da uno dei due motivi esposti di seguito.
La natura fisica della risorsa non rende praticabile la via della condivisione. Esempio tipico ne è un lettore di nastro perforato, nel quale è impossibile commutare i nastri tra caratteri consecutivi.
Il tipo di risorsa è tale che, se fosse utilizzata da più processi in modo concorrente, le azioni di uno di questi potrebbero interferire con quelle di un altro. Un esempio molto comune consiste in una locazione di memoria contenente una variabile cui può aver accesso più di un processo: se un processo esamina la variabile in questione proprio mentre un altro la modifica, il risultato è imprevedibile e solitamente disastroso. A titolo d'esempio, supponiamo che in un sistema per la prenotazione dei passaggi aerei la disponibilità di un determinato posto sia rappresentata da quanto è contenuto in una certa locazione di memoria. Se quella locazione è accessibile a più processi contemporaneamente, è possibile per due agenti di viaggio prenotare simultaneamente il posto preso in considerazione svolgendo l'infelice sequenza di azioni riportate qui sotto:

1. L'agente A vede che il posto è libero e consulta il cliente
2. L'agente B vede che il posto è libero e consulta il cliente
3. L'agente A prenota il posto
4. L'agente B prenota lo stesso posto

Tra le risorse non condivisibili, pertanto, sono comprese per lo più le periferiche, i file modificabili e le aree dati che sono soggette a modifica; tra le risorse condivisibili si annoverano le CPU, i file a sola lettura e le zone di memoria contenenti procedure pure o dati protetti contro l'eventuale modifica.
La tecnica della mutua esclusione consiste nell'assicurare che possa accedere solamente un processo per volta alle risorse non condivisibili.
In linea generale, la velocità di un processo relativamente ad un altro è imprevedibile, poiché dipende dalla frequenza con cui si verificano le interruzioni in ciascun processo e anche dalla frequenza e dal tempo di assegnazione del processore. Diciamo che i processi si svolgono in modo asincrono l'uno rispetto all'altro. Ciò nonostante, per raggiungere una buona collaborazione tra essi, vi sono determinati punti in cui i processi devono sincronizzare le proprie attività. Si tratta, cioè, di quei punti oltre i quali un processo non riesce ad andare fino a che un altro non abbia portato a termine una certa attività. Per esempio, un processo di schedulazione dei job degli utenti non può proseguire fino a che un processo di input non abbia caricato nella memoria della macchina almeno un job. Analogamente, può accadere che un processo di calcolo che produce un qualche output non possa proseguire fino a che il suo output precedente non sia stato stampato. Fa parte dei compiti del sistema operativo mettere a disposizione meccanismi tali da attivare la sincronizzazione.

3.2.2 Situazioni di stallo

Quando svariati processi si trovano a competere per l'utilizzo delle risorse, è possibile il verificarsi di una situazione in cui nessun processo riesce a continuare perché le risorse necessarie a ciascuno di essi sono tenute sotto controllo da un altro. Tale situazione viene denominata stallo (deadlock o deadly embrace) ed è analoga all'intasamento che si verifica allorché due correnti opposte di traffico, tentando di svoltare tagliandosi la strada reciprocamente, si immobilizzano

completamente, poiché ciascuna corrente occupa lo spazio stradale necessario all'altra. È chiaro che la prevenzione delle situazioni di stallo, o per lo meno la limitazione dei relativi effetti, è una delle funzioni del sistema operativo.

3.3 I SEMAFORI

Il più importante contributo verso la comunicazione tra processi fu dato dall'introduzione (Dijkstra, 1965) del concetto di semaforo e delle operazioni primitive di wait e signal (attesa ed avvio) che agiscono su di essi. Spesso ci si riferisce a wait e signal con le denominazioni originali di Dijkstra, rappresentate dalle iniziali delle parole corrispondenti in olandese, P e V. Noi comunque adotteremo la terminologia più recente e descrittiva.
Un semaforo è un numero intero non negativo sul quale, a parte l'inizializzazione del suo valore, possono agire solo le operazioni di wait e signal. Queste agiscono esclusivamente sui semafori ed il loro effetto risponde alle seguenti definizioni.
signal(s)
Il suo effetto consiste nell'aumentare di un'unità il valore del semaforo s, dove l'aumento viene considerato un'operazione indivisibile. L'indivisibilità comporta che signal(s) non è equivalente alla fase di assegnamento "s := s + 1". Per fare un esempio, supponiamo che due processi, A e B, vogliano eseguire l'operazione di "signal (s)" con un valore di s uguale a 3. Il valore di s, quindi, al completamento di entrambe le operazioni, sarà 5. Supponiamo d'altro canto che, in circostanze analoghe, entrambi i processi vogliano eseguire "s := s +1". A potrebbe quindi valutare l'espressione s + 1 e riscontrare che è uguale a 4, ma prima che A possa assegnare ad s il risultato, anche B valuterebbe s + I con risultato identico. Ciascun processo, quindi, assegnerebbe il valore 4 ad s ed uno degli incrementi desiderati andrebbe perso.
wait(s)
Il suo effetto consiste nel diminuire di un'unità il valore del semaforo s non appena si prefigura la possibilità di un risultato non negativo. Anche qui si tratta di un'operazione indivisibile. L'operazione di wait comporta un ritardo potenziale, dato che, quando agisce su un semaforo il cui valore è 0, il processo che esegue l'operazione può proseguire solo quando un altro processo abbia aumentato di 1 il valore del semaforo tramite un'operazione di signal. L'indivisibilità dell'operazione sta a significare che, qualora vengano ritardati svariati processi, solo uno di essi può mandare a buon fine l'operazione allorché il semaforo diventa positivo. Non si premette, però, di quale processo si tratti.
Gli effetti delle operazioni di wait e signal possono essere riassunti come segue:

- wait(s) : when s>0 do decremento di s
- signal(s) : incremento di s

dove s è un qualsiasi semaforo.
Si può vedere da queste definizioni che ogni operazione di signal su un semaforo ne aumenta il valore di 1 ed ogni operazione riuscita (cioè completata) di wait ne diminuisce il valore sempre di 1. Quindi, il valore di un semaforo si trova in rapporto con il numero di operazioni di wait e signal eseguite su di esso, secondo la relazione

```
val(sem)=C(sem) + ns(sem) - nw(sem)                    (3.1)
```

dove
val(sem) è il valore del semaforo sem, C(sem) è il suo valore iniziale
ns(sem) è il numero di operazioni di signal eseguite su di esso
nw(sem) è il numero di operazioni di wait andata a buon fine su di esso.
Ma per definizione

```
val(sem)>=0
nw(sem) <= ns(sem) + C(sem)                            (3.2)
```

nella quale l'eguaglianza risulta vera se e solo se val(sem) = 0.
La relazione (3.2) è invariante rispetto alle operazioni di wait e di signal, è, cioè, vera qualsiasi sia il numero delle operazioni eseguite.
L'implementazione dei semafori e delle operazioni di wait e signal sarà esaminata nel capitolo successivo. Per adesso la daremo per scontata e ce ne serviremo per risolvere i problemi elencati in precedenza.

3.3.1 Mutua esclusione

Le risorse non condivisibili, sia che si tratti di periferiche, di file o di dati in memoria, possono essere protette dall'accesso simultaneo ad opera di svariati processi, evitando che questi eseguano in maniera concorrente i segmenti di programma tramite cui avviene tale accesso. Questi segmenti di programma sono definiti sezioni critiche e la mutua esclusione nell'utilizzo delle risorse può essere vista come mutua esclusione nell'esecuzione delle sezioni critiche. Per esempio, si possono escludere reciprocamente programmi, evitando che abbiano accesso ad una tavola di dati, se tutte le routine che leggono o modificano quella tavola sono scritte sotto forma di sezioni critiche, così da renderne eseguibile solo una alla volta.
Si può ottenere la mutua esclusione attraverso il semplice espediente di racchiudere ogni sezione critica tra le operazioni wait e signal su uno stesso semaforo il cui valore iniziale sia 1. Pertanto, ciascuna sezione critica è così programmata:

```
wait(mutex);
sezione critica
signal(mutex);
```

dove mutex è il nome di un semaforo.
Ad un attento esame, il lettore noterà che, se il valore iniziale di mutex è uguale ad 1, la mutua esclusione è veramente garantita, poiché al massimo un solo processo può eseguire wait(mutex) prima che un altro esegua signal(mutex). In maniera più formale, applicando la relazione 3.2 si ottiene

```
nw (mutex) <= ns (mutex) + 1
```

che implica che al massimo un processo può trovarsi all'interno della sua sezione critica in un dato momento.
Inoltre, in questo modo non si impedisce mai, senza che ve ne sia il bisogno, l'ingresso di un processo nella sua sezione critica; ciò avviene solamente se qualche altro processo si trova già entro la propria sezione critica. Possiamo verificarlo osservando che ad un processo viene negato l'ingresso solo se il valore di mutex è uguale a 0. In tal caso la relazione 3.2 indica che

```
nw(mutex) = ns(mutex) + 1
```

In altri termini, il numero di operazioni di wait andate a buon fine sul semaforo mutex supera di un'unità il numero delle operazioni di signal, il che indica che qualche processo si trova già entro la propria sezione critica.
La conclusione che traiamo è che ogniqualvolta un processo desidera avere accesso ad una variabile o ad una risorsa soggetta a condivisione deve accedervi attraverso una sezione critica protetta da un semaforo, come esposto più sopra.
A titolo d'esempio, supponiamo che un sistema operativo contenga due processi, A e B, che rispettivamente aggiungono e tolgono elementi da una coda. Per far sì che non si crei confusione per i puntatori della coda, può risultare necessario

limitare l'accesso alla stessa ad un solo processo per volta. L'aggiunta e la rimozione degli elementi, così, verrebbe codificata come sezione critica, secondo quanto segue:

Programma per il processo A	**Programma per il processo B**
…….	…….
wait(mutex);	wait(mutex);
aggiunta di un elemento in coda	rimozione di un elemento dalla coda
signal(mutex);	signal(mutex);
…….	…….

A questo punto il lettore sospettoso potrebbe chiedersi se si poteva risolvere il problema della mutua esclusione senza l'ulteriore formalità dei semafori e delle operazioni ad essi connesse. A giudicare dalle apparenze, tale sospetto potrebbe sembrare ben fondato, in quanto pare possibile ottenere una soluzione valida proteggendo ciascuna sezione critica per mezzo di un'unica, semplice variabile (che poniamo sia denominata porta). Quando porta viene settata ad aperta (che poniamo sia rappresentato dal valore 1) è concesso entrare nella sezione critica; quando è settata a chiusa (rappresentato da 0) non è concessa l'entrata. Pertanto, una sezione critica verrebbe codificata così:

```
while porta = chiusa do nessuna operazione;
porta: = chiusa;
sezione critica
porta: =aperta;
```

Qui ogni processo entrante nella sezione critica verifica che la porta sia aperta (e continua il ciclo fino a che non lo diviene); quindi setta porta a chiusa, evitando che altri processi entrino nella sezione. All'uscita il processo ripristina porta ad aperta, liberando la sezione e rendendo possibile l'esecuzione ad opera di un altro processo.

1. Purtroppo questa semplice soluzione non funzionerebbe. Il motivo sta nel fatto che la verifica che la porta è aperta (in riga è separata dalla chiusura della porta (in riga 2). Di conseguenza, due eventuali processi in esecuzione parallela potrebbero entrambi trovare una porta aperta (in riga 1), prima che uno dei due abbia la possibilità di chiuderla (in riga 11 risultato sarebbe che ambedue entrano assieme nella sezione critica.
2. I semafori evitano simili difficoltà proprio grazie al fatto che le operazioni di wait e signal sono indivisibili; non esiste la possibilità che due processi agiscano contemporaneamente sullo stesso semaforo. L'implementazione dei semafori, oggetto di trattazione nel capitolo 4, deve naturalmente garantire che ciò accada.

3.3.2 Sincronizzazione

La forma più semplice di sincronizzazione può essere rappresentata da un processo A che non dovrebbe andare oltre un dato punto L1 fino a che qualche altro processo B non abbia raggiunto il punto L2. Si verificano esempi di questa situazione ogni qual volta A richiede, al livello L1 informazioni che vengono fornite da B solo dopo aver raggiunto L2. Si può programmare la sincronizzazione come segue:

Programma per il processo A	**Programma per il processo B**
…….	…….
L1: wait(prosecuzione);	L2: signal(prosecuzione);
…….	…….

dove prosecuzione rappresenta un semaforo con valore iniziale 0.
Risulta chiaro dai programmi su riportati che A non può proseguire oltre L1 fino a che B non abbia eseguito l'operazione di signal a livello L2. Ovviamente, se B esegue quest'ultima operazione prima che A abbia raggiunto L1, non si verifica alcun ritardo per A. Possiamo nuovamente far ricorso alla relazione 3.2 per introdurre una dimostrazione più formale. Abbiamo

```
nw(prosecuzione) < = ns(prosecuzione)
```

che implica che A non può oltrepassare L1 prima che B abbia oltrepassato L2.
L'esempio citato è asimmetrico, in quanto l'avanzamento di A è regolato da quello di B, ma non viceversa. Il caso in cui due processi regolano reciprocamente l'avanzamento dell'altro è esemplificato dal problema classico del produttore e del consumatore. Dal momento che tale problema è rappresentativo anche di molti altri che si verificano nella comunicazione tra i processi, ci addentreremo in maggiori dettagli.
Un insieme di processi "produttore" ed uno di processi "consumatore" comunicano per mezzo di un dispositivo di transito (buffer) nel quale i produttori depositano articoli che poi vengono prelevati dai consumatori. I produttori ripetono continuamente il ciclo "produzione di un articolo, deposito dell'articolo nel buffer", mentre i consumatori ripetono un ciclo analogo: "prelievo di un articolo dal buffer, consumo dell'articolo". Un produttore tipo potrebbe essere rappresentato da un processo di calcolo che mette in un buffer delle linee di output, mentre il corrispondente consumatore potrebbe essere il processo che provvede alla stampa di ciascuna linea. Il buffer possiede una capacità limitata, pur essendo abbastanza grande da contenere N articoli di dimensioni identiche. La sincronizzazione richiesta è duplice: in primo luogo, i produttori non possono mettere articoli nel buffer se questo è già pieno e, secondariamente, i consumatori non possono prelevare articoli se il buffer è vuoto. In altri termini, se il numero di articoli depositati è d ed il numero di quelli prelevati è e, dobbiamo assicurare che:

$$0 <= d - e <= N \quad (3.3)$$

Inoltre, il buffer deve essere protetto dall'accesso simultaneo di diversi processi, per il timore che le azioni di uno (ad esempio nella modifica dei puntatori) possano disturbare quelle dell'altro. Sia il deposito che il prelievo di articoli, quindi, devono essere codificati sotto forma di sezioni critiche.
Avanziamo il postulato che la soluzione del problema sia la seguente:

Programma per i produttori	**Programma per i consumatori**
repeat indefinitamente	repeat indefinitamente
begin	begin
produci l'articolo;	wait(articolo disponibile);
wait(spazio disponibile);	wait(manipolazione del buffer);
wait(manipolazione del buffer);	preleva articolo dal buffer;
deposita articolo nel buffer;	signal (manipolazione del buffer);
signal (manipolazione del buffer);	signal (spazio disponibile);
signal (articolo disponibile);	consuma l'articolo
end	end

La sincronizzazione viene raggiunta tramite i semafori spazio disponibile ed articolo disponibile in cui i valori iniziali sono N e 0 rispettivamente. La mutua esclusione dall'accesso al buffer, per quanto riguarda i processi, viene messa in atto per mezzo del semaforo manipolazione del buffer avente valore iniziale 1. Ora dobbiamo dimostrare che questa soluzione soddisfa la relazione 3.3. Applicando la relazione di invarianza 3.2 ai due semafori di sincronizzazione, otteniamo:

`nw(spazio disponibile) <= ns(spazio disponibile) + N` (3.4)

e

`nw(articolo disponibile) <= ns (articolo disponibile)` (3.5)

Però, dall'ordine delle operazioni relative al programma per i produttori, osserviamo che

`ns(articolo disponibile) <= d <= nw(spazio disponibile)` (3.6)

e, dall'ordine delle operazioni relative al programma per i consumatori, che

`ns(spazio disponibile) <= e <= nw(articolo disponibile)` (3.7)

Quindi dalla relazione 3.6

```
d <= nw(spazio disponibile)
     <= ns(spazio disponibile)+N per la 3.4
     <= e + N per la 3.7
```

Analogamente, dalla relazione 3.7

```
e <= nw(articolo disponibile)
     <= ns(articolo disponibile) per la 3.5
     <= d per la 3.6
```

Combinando questi due risultati, si ottiene

`e <= d <= e + N`

che dimostra che la relazione 3.3 è soddisfatta come richiesto.

La soluzione del problema dei produttori e dei consumatori può essere usata come traccia per tutte quelle situazioni in cui un processo passa informazioni ad un altro. Più in particolare, i processi che governano le periferiche di input fungono da produttori per quei processi che consumano l'input, così come i processi che governano le periferiche di output fungono da consumatori per quei processi che producono output.

Come sottolineato in precedenza, può verificarsi una situazione di stallo ogni qual volta un gruppo di processi si trova a competere per l'utilizzo delle risorse. Di per sé si tratta di un problema che prenderemo in esame esaurientemente solo nei prossimi capitoli.

In questo paragrafo osserviamo che si possono verificare situazioni di stallo anche a causa di processi in reciproca attesa che altri portino a compimento determinate azioni. A titolo d'esempio, consideriamo due processi, A e B, che agiscono sui semafori X e Y come descritto di seguito.

Processo A	**Processo B**
…….	…….
wait (X);	wait (Y);
…….	…….
wait (Y);	wait (X);
…….	…….

Se i valori iniziali sia di X che di Y sono uguali ad 1, ogni processo riesce a completare la prima operazione di wait abbassando i valori dei semafori a 0. Chiaramente, nessuno dei due processi riesce a quel punto a proseguire oltre la successiva operazione di wait ed ha il sopravvento una situazione di stallo.

Di fatto, la situazione è analoga a quella in cui lo stallo si verifica a causa dell'antagonismo nella richiesta di utilizzo delle risorse. Se si considera un semaforo come una risorsa e se le operazioni di wait e signal vengono viste come miranti all'utilizzo o al rilascio della stessa, la situazione che ne consegue è di stallo, poiché sia A che B detengono la risorsa necessaria all'altro. La "condivisibilità" di un semaforo è determinata dal suo valore iniziale: se esso è n (< I), il semaforo può essere condiviso da n processori; se è 1 (come nel caso preso in esame), il semaforo non è condivisibile.

Un caso di stallo analogo, anche se più complesso, si può ottenere dal problema dei produttori e dei consumatori, ribaltando l'ordine delle operazioni di wait eseguite da questi ultimi. Se il buffer è vuoto, per un consumatore è possibile guadagnare l'accesso al buffer eseguendo "wait(manipolazione del buffer)", per venire poi posto in stato di sospensione sul "wait(articolo disponibile)". L'unico modo in cui si può sbloccare il consumatore consiste nel far depositare un articolo nel buffer dal produttore, facendogli eseguire "signal(articolo disponibile)". Tuttavia, nessun produttore può avere accesso al buffer dato che questo viene occupato dal consumatore in stato di blocco. Ne consegue una situazione di stallo. Ancora una volta la situazione è analoga a quella risultante dall'antagonismo nella richiesta di utilizzo delle risorse.

In questo caso le risorse sono il buffer, occupato dal consumatore, ed il nuovo articolo, detenuto dal produttore.

L'insegnamento da trarsi da questi esempi è che si possono verificare situazioni di stallo conseguentemente a una sequenza di operazioni di wait disposte secondo un ordine sbagliato, anche quando non viene avanzata nessuna richiesta esplicita di utilizzo di risorse. Può giovare rendersi conto del fatto che il pericolo di stallo incombente relativo alla soluzione erroneamente data al problema del produttore e del consumatore non verrebbe rilevato dalla validazione eseguita nel paragrafo precedente. Sia la soluzione che non presenta situazioni di stallo, sia quella con tendenza a crearne, soddisfano la relazione 3.3 e perciò, da quel punto di vista, sembrano parimenti corrette. 1 criteri per il riconoscimento delle possibilità di stallo in relazione all'allocazione delle risorse saranno presi in esame nel capitolo 8. Si può effettuare il riconoscimento delle eventuali situazioni di stallo (o la loro assenza) all'interno di una sequenza di operazioni di wait, svolgendo un'analisi simile a quella esposta in seguito.

Si prenda in considerazione la soluzione data al problema del produttore e del consumatore nel precedente punto (2). Dimostreremo che in realtà non presenta situazioni di stallo. In primo luogo osserviamo che le sezioni interne dei programmi per i produttori e per i consumatori sono semplicemente sezioni critiche protette con il semaforo manipolazione del buffer e che perciò di per sé non possiedono nessuna potenziale situazione di stallo. Quest'ultimo, quindi, può avere la meglio solo se si verificano i seguenti due casi:

a. nessun produttore riesce a superare wait(spazio disponibile) e nessun consumatore è in grado di eseguire signal(spazio disponibile) (ovverosia, nessun consumatore preleva articoli);
b. nessun consumatore riesce a superare wait(articolo disponibile) e nessun produttore è in grado di eseguire signal(articolo disponibile) (ovverosia, nessun produttore deposita articoli).

La condizione (a), applicando la relazione 3.2, implica che

```
nw(spazio disponibile) = ns(spazio disponibile) + N
        e
nw(articolo disponibile) = ns(spazio disponibile)
```

La condizione (b), applicando la relazione 3.2, implica che

```
nw(articolo disponibile) = ns(articolo disponibile)
     e
nw(spazio disponibile) = nw(articolo disponibile)
```

Mettendo assieme queste relazioni otteniamo

```
nw(spazio disponibile) = nw(spazio disponibile) + N
```

che è una contraddizione, poiché N>0.

Un'analisi simile nel caso in cui venga invertito l'ordine delle operazioni di wait nel programma dei consumatori non è possibile, dato che la sezione interna di quest'ultimo non è più una semplice sezione critica: a questo punto conterrebbe un'operazione di wait. È necessaria un'analisi più complessa (per esempio Habermann, 1972; Lister, 1974) per dimostrare che in realtà possono aver luogo situazioni di stallo.

3.4 I MONITOR

Nel paragrafo precedente abbiamo visto come sia possibile fare uso dei semafori per attuare la sincronizzazione e la comunicazione tra processi. L'impiego indisciplinato dei semafori tuttavia è piuttosto incline a dar luogo a situazioni d'errore: è facile per un programmatore mettere nel posto sbagliato operazioni di wait e signal oppure, addirittura, non metterne affatto. Ad esempio, se un programmatore non si accorge che una particolare struttura dati va condivisa tra parecchi processi non provvederà ad includere le sezioni critiche dei programmi che vi hanno accesso tra operazioni di wait e signal. La struttura dati risulterà quindi non protetta dalla manipolazione contemporanea da parte di processi diversi e probabilmente si creeranno situazioni contraddittorie all'interno del suo contenuto.

Alfine di evitare l'insorgere di problemi di questo genere sono state avanzate parecchie ipotesi relative ad accorgimenti per i linguaggi di programmazione che costringono il programmatore a dichiarare esplicitamente le risorse ed i dati da condividere, mettendo in atto la mutua esclusione per quanto concerne l'accesso a tali oggetti di condivisione. Uno degli accorgimenti più largamente adottati e di maggiore importanza è il (programma) monitor (Hoare, 1974) del quale daremo una breve descrizione qui di seguito.

Un monitor consiste di:

1. i dati comprendenti un oggetto in condivisione;
2. un insieme di procedure che possono essere richiamate per accedere all'oggetto;
3. un (segmento di) programma che inizializzi l'oggetto (questo programma viene eseguito una sola volta, quando l'oggetto viene creato).

Per esempio, un buffer (come quello descritto nel precedente paragrafo) per passare i dati tra i processi dei produttori e quelli dei consumatori potrebbe essere rappresentato da un monitor comprendente:

lo spazio ed i puntatori del buffer (per esempio, un array e gli indici relativi);

due procedure, deposito e prelievo, che possono essere richiamate dai processi per mettere e togliere, rispettivamente, un articolo dal buffer;

un (segmento di) programma che inizializzi i puntatori del buffer sulla sua locazione iniziale.

Il compilatore di un linguaggio comprendente dei monitor deve assicurare che l'accesso ad un oggetto condiviso sia possibile solamente richiamando una procedura del monitor corrispondente; si può raggiungere piuttosto agevolmente questo obiettivo tramite il meccanismo di "scope" della maggior parte dei linguaggi con struttura a blocchi. Il compilatore deve anche garantire che le procedure di ciascun monitor siano implementate sotto forma di sezioni critiche che si escludono reciprocamente, e può riuscirvi facilmente generando operazioni di wait e signal sugli idonei semafori esistenti nel programma compilato. Ritornando all'esempio relativo al buffer, il compilatore assicura così che l'accesso al buffer sia limitato alle procedure di deposito e prelievo e che queste ultime si escludano reciprocamente.

Dovrebbe risultare ovvio che i monitor eliminano una fonte potenzialmente ricca di situazioni d'errore, trasferendo dal programmatore al compilatore la responsabilità della mutua esclusione. Chi si assume la responsabilità per altre forme di sincronizzazione, tuttavia, è sempre il programmatore, il quale per realizzarla deve servirsi dei semafori (o di altri metodi equivalenti). Per esempio, la rappresentazione sotto forma di monitor di un buffer (come indicato più sopra) garantisce che, tramite la mutua esclusione del deposito e del prelievo, non si possano inserire e togliere contemporaneamente gli articoli. Ma, così, non si impedisce che i processi continuino a depositare articoli nel buffer quando è pieno, né che tentino di prelevarli quando è vuoto. Si può invece prevenire questo tipo di fenomeno disastroso introducendo adeguate operazioni di sincronizzazione all'interno delle procedure di deposito e di prelievo.

L'accorgimento del monitor è stato implementato in svariati linguaggi di programmazione, quali il Concurrent Pascal (Hansen, 1975), il Pascal-plus (Welsh e Bustard, 1979) e il Mesa (Lampson e Redell, 1980). Per chi si occupa dell'implementazione dei sistemi operativi presenta il vantaggio di limitare ad un insieme di procedure ben definite tutte le operazioni svolte su un oggetto condiviso, nonché quello di garantire che tali operazioni si escludano reciprocamente. La sincronizzazione per scopi diversi da quello della mutua esclusione resta di pertinenza del programmatore.

La struttura generica di un monitor (Hoare, 1974) espressa in una sintassi derivata dal Pascal, è la seguente:

```
var m:monitor;
begin
```

dichiarazione delle variabili locall che rappresentano l'oggetto; dichiarazione delle procedure che accedono alle variabili locall; inizializzazione delle variabili locall

```
end
```

Alle variabili locall non si può avere accesso fuori dall'ambito del monitor; in seguito all'inizializzazione, su di esse possono agire solo le stesse procedure del monitor. Per impedirne la manipolazione contemporanea da parte di più processi, le procedure del monitor sono implementate sotto forma di sezioni critiche dotate di mutua esclusione. Quest'ultima è assicurata dal compilatore, il quale genera le opportune operazioni di sincronizzazione (quali le operazioni di wait e signal sui semafori) nel programma compilato. Per i lettori che hanno familiarità con i tipi astratti di dati, un monitor può essere considerato come un oggetto di dati astratto condivisibile tranquillamente tra svariati processi.

Nonostante la mutua esclusione delle procedure monitor sia garantita dal compilatore, l'onere di altre forme di sincronizzazione resta di pertinenza del programmatore. Le operazioni di sincronizzazione proposte da Hoare, ed adottate nella maggior parte dei monitor implementati, non consistono in operazioni di wait e signal sui semafori, ma in operazioni simili sugli oggetti denominate condizioni. Analogamente ad un semaforo, una condizione ha due operazioni che possono

essere applicate ad essa: una serve a ritardare un processo, mentre l'altra a riprenderlo. Chiameremo queste operazioni rispettivamente cwait e csignal, per distinguerle dalle operazioni di wait e signal eseguite sui semafori.
Le definizioni di cwait e csignal sono le seguenti:
cwait (condizione): sospendi l'esecuzione del processo chiamante
csignal (condizione): riprendi l'esecuzione di qualche processo sospeso in seguito ad un cwait sulla stessa condizione. Se ve ne sono svariati, selezionane uno; se non ve ne sono, non intervenire.
L'operazione di cwait annulla l'esclusione del monitor, altrimenti nessun altro processo sarebbe in grado di entrare nel monitor per eseguire un csignal. Analogamente, csignal annulla l'esclusione, affidandola al processo ripreso. A quest'ultimo, pertanto, deve essere concesso di venire eseguito prima che sia possibile entrare nel monitor per qualsiasi altro processo.
Benché condizioni e semafori servano a scopi simili, le differenze che seguono sono significative.

a. Una condizione, contrariamente ad un semaforo, non possiede alcun valore. Si può pensare ad essa come ad una coda cui viene aggiunto un processo in seguito all'esecuzione di un cwait e dalla quale viene rimosso quando qualche altro processo effettua un csignal.
b. Un'operazione di cwait fa sempre sì che il processo in esecuzione venga ritardato, a differenza di quanto avviene con l'operazione di wait, che provoca un ritardo solo se il valore del semaforo è zero.
c. Un'operazione di csignal produce effetti solo se un processo è sospeso sulla stessa condizione. Un'operazione di signal, invece, produce sempre un effetto: l'incremento del valore del semaforo. Le operazioni di signal vengono "ricordate" (nel valore del semaforo), mentre ciò non succede per le operazioni di csignal.

A titolo d'esempio sull'impiego delle condizioni per scopi di sincronizzazione, forniamo i particolari del monitor per il buffer di cui abbiamo parlato nel paragrafo precedente:

Figura 12 - Un monitor

```
monitor example
integer i;
condition c;
procedure producer(x);
end;
procedure consumer(x);
end;
end monitor;
var buffer: monitor;
begin
var B: array[0….. .N—1]of item (spazio per N articoli)
nextin, nextout: 0. . . N - 1; (puntatori del buffer)
nonfull, nonempty, condition; (per la sincronizzazione)
count: 0 ... N; (numero di articoli nel buffer)
procedure deposit (x: item);
begin
     if count = N then cwait (nonfull); (evita overflow)
     B [nextin] : = x
     nextin: = (nextin + 1) mod N;
     count: = count + 1; (un dato in più nel buffer)
     csignal (nonempty) (riprendi qualsiasi processo consumatore in attesa)
end;
procedure extract (var x: item);
begin
     if count = O then cwait (nonempty); (evita overflow)
     x: = B[nextout];
     nextout: = (nextout + 1) mod N;
     count: = count — 1; (un articolo in meno nel buffer)
     csignal (nonfull) (riprendi qualsiasi processo produttore in attesa)
end;
nextin: = 0; nextout: = 0; (buffer inizialmente vuoto)
count: = 0
end
```

I monitor hanno una proprietà importante che li rende utili per ottenere la mutua esclusione: un solo processo alla volta può essere attivo al loro interno. I monitor sono un costrutto di un linguaggio di programmazione; come tali, il compilatore è a conoscenza delle loro peculiarità e può gestire le chiamate alle procedure di monitor in modo differente dalle altre. In genere, quando un processo chiama una procedura di monitor, le prime istruzioni controllano se un altro processo sta in quel momento utilizzando il monitor. Se è così, il processo viene sospeso finché l'altro processo non abbandona il monitor. Se nessun altro processo sta usando il monitor, il processo chiamante può entrare. Il compilatore ha il compito di implementare la mutua esclusione negli accessi al monitor, solitamente usando un semaforo binario. Dato che è il compilatore, e non il programmatore, a gestire la mutua esclusione, è molto meno probabile che qualcosa vada storto. La persona che programma il monitor non si deve comunque preoccupare di come il compilatore otterrà la mutua esclusione, gli basta sapere che scrivendo tutte le sezioni critiche come procedure di monitor, non accadrà mai che due processi eseguano le loro sezioni critiche nello stesso istante.
Vediamo un'altra soluzione al problema del produttore-consumatore con l'introduzione di due variabili di condizione, insieme a due operazioni, WAIT e SIGNAL, che operano su di esse. Quando una procedura di monitor scopre di non poter continuare (ad esempio quando il produttore trova il buffer pieno), effettua una WAIT sulla variabile di condizione fuil. Questa chiamata provoca la sospensione del processo chiamante e consente l'ingresso nel monitor a un altro processo a cui era stato in precedenza proibito l'accesso. Quest'altro processo, per esempio il consumatore, può svegliare il suo partner effettuando una SIGNAL sulla variabile di condizione su cui è in attesa. Per evitare due processi attivi all'interno del monitor nello stesso istante, abbiamo bisogno di una regola che stabilisca cosa accade dopo una SIGNAL. Hoare propose di lasciare in esecuzione il processo appena risvegliato, sospendendo l'altro. Brinch Hansen propose di risolvere il problema richiedendo che il processo che esegue la SIGNAL abbandoni immediatamente il monitor. In altre parole, un costrutto SIGNAL può apparire solo come l'ultimo costrutto di una procedura di monitor. In questo testo useremo la proposta di Brinch Hansen, poiché è concettualmente più semplice e anche più facile da realizzare. Se si invoca una

SIGNAL su una variabile di condizione su cui diversi processi sono in attesa, solo uno di essi, scelto dallo scheduler di sistema, viene risvegliato.
Le variabili di condizione non sono contatori e non accumulano segnali per un uso successivo come fanno i semafori. Pertanto, se si invia un segnale a una variabile di condizione su cui nessuno è in attesa, il segnale viene perduto. La WAIT deve cioè essere effettuata prima della SIGNAL. Questa regola rende l'implementazione assai più semplice. In pratica, questa limitazione non crea problemi, dato che, se necessario, è facile usare delle variabili per tener conto dello stato di ogni processo. Un processo che potrebbe altrimenti invocare una SIGNAL è in grado, controllando queste variabili, di accorgersi che l'operazione non è necessaria.
Lo schema di una soluzione al problema produttore-consumatore, programmata in uno pseudo-Pascal che prevede l'uso dei monitor, viene mostrato in figura 13.
La mutua esclusione automatica sulle procedure di monitor garantisce che se, ad esempio, il produttore scopre che il buffer è pieno all'interno di una procedura di monitor, esso sarà in grado di completare l'operazione WAIT senza doversi preoccupare della possibilità che lo scheduler esegua il consumatore immediatamente prima del completamento della WAIT. Il consumatore non potrà entrare nel monitor fino a che la WAIT non sarà terminata e il produttore segnalato come non più eseguibile.
Automatizzando la mutua esclusione in sezione critica, i monitor rendono la programmazione parallela molto meno soggetta a errori di quanto non lo sia con i semafori. Ciononostante, i monitor conservano alcune controindicazioni. Non a caso la figura 13 è stata scritta in pseudo-Pascal piuttosto che in C, come gli altri esempi di questo libro. Come detto in precedenza, i monitor sono un concetto relativo ai linguaggi di programmazione. Il compilatore deve poterli riconoscere e provvedere in qualche modo alla mutua esclusione.
Il C, il Pascal e la maggior parte degli altri linguaggi non hanno i monitor, ed è quindi poco ragionevole aspettarsi che i loro compilatori impongano una qualsiasi regola di mutua esclusione. Di fatto, come potrebbe il compilatore sapere quali procedure fanno parte dei monitor e quali no?.

Figura 13 - Un'alternativa al programma produttore consumatore

```
monitor ProducerConsumer condition full, empty;
integer count;
procedure enter;
    begin
        if count = N then wait(full);
        enter_item;
        count:=count+ 1;
        if count = 1 then signal(empty)
    end;
procedure remove;
    begin
        if count = 0 then wait(empty);
        remove.jtem;
        count := count— 1;
        if count = N – 1 then signal(full)
    end;
count := 0;
end monitor;
procedure producer;
    begin
    while true do
        begin
        produce_item;
        ProducerConsumer. enter
        end
    end;
procedure consumer;
        begin
        while true do
        begin
        ProducerConsumer. remove;
        consume.item
        end
end;
```

Con i monitor, si ha bisogno di un linguaggio che ne preveda l'uso. Alcuni linguaggi, come il Concurrent Euclid (Holt, 1983) comprendono i monitor, ma sono rari.
Un altro problema dei monitor, e anche dei semafori, è che queste astrazioni erano state progettate per risolvere il problema della mutua esclusione su una o più CPU che accedono a una memoria comune. Quando si passa a un sistema distribuito con più CPU, ognuna con la sua memoria privata e connesse da una rete locale, queste primitive diventano inapplicabili. La conclusione è che i semafori sono di livello troppo basso, e i monitor non sono utilizzabili eccetto che in pochi linguaggi di programmazione. Inoltre, nessuna di queste primitive fornisce il supporto per lo scambio di informazioni tra macchine. È pertanto necessario pensare a qualcosa di diverso.

3.5 I MESSAGGI

3.5.1 Send e receive

Quel qualcosa di diverso è lo scambio di messaggi. Questo metodo di comunicazione interprocesso usa due primitive, SEND e RECEIVE che, come i semafori e diversamente dai monitor, sono chiamate di sistema piuttosto che costrutti del linguaggio. Come tali, esse possono essere facilmente inserite in procedure di libreria, come

```
send(destinazione, &messaggio)
    e
receive(sorgente, &messaggio)
```

La prima di queste chiamate invia un messaggio a un dato destinatario mentre la seconda riceve un messaggio da una data sorgente (o da una qualsiasi sorgente, ANY, se al destinatario non importa). Se nessun messaggio è disponibile, il destinatario potrebbe bloccarsi finché non ne arriva uno. In alternativa potrebbe ritornare immediatamente con un codice di errore.

3.5.2 Problemi progettuali per i sistemi a scambio di messaggi

I sistemi a scambio di messaggi pongono molti problemi stimolanti e molte questioni progettuali che non sorgono con i semafori o i monitor, specialmente se i processi Comunicanti risiedono su macchine diverse connesse da una rete. Per esempio, i messaggi possono essere persi dalla rete. Per salvaguardarsi dalla perdita dei messaggi, il mittente e il destinatario possono accordarsi in modo che, non appena un messaggio viene ricevuto, il destinatario invii indietro uno speciale messaggio di acknowledgement (conferma). Se il mittente non riceve l'acknowledgement entro un determinato intervallo di tempo, ritrasmette il messaggio.
Consideriamo ora ciò che accade se il messaggio viene ricevuto correttamente, ma l'acknowledgement viene perso. Il mittente ritrasmetterà il messaggio e il destinatario lo riceverà due volte. È essenziale che il destinatario possa distinguere un nuovo messaggio dalla ritrasmissione di uno vecchio. Di solito, questo problema viene risolto aggiungendo a ogni messaggio numeri di sequenza consecutivi. Se il destinatario riceve un messaggio che porta lo stesso numero di sequenza del messaggio precedente, si accorge che il messaggio è un duplicato che può essere ignorato.
I sistemi a messaggi devono anche trattare il problema di come assegnare i nomi ai processi in modo che il processo specificato in una chiamata SEND o RECEIVE venga identificato senza ambiguità. L'autenticazione è un ulteriore problema dei sistemi a messaggi: come può il cliente sapere che sta comunicando con il file server reale, e non con un impostore?
All'altro estremo dello spettro ci sono i problemi progettuali che diventano rilevanti quando sia il mittente che il destinatario sono sulla stessa macchina. Uno di questi è il problema delle prestazioni. La copia dei messaggi da un processo all'altro è sempre più lenta di un'operazione semaforica o dell'accesso a un monitor. Si è fatto molto per rendere efficiente lo scambio di messaggi. Cheriton (1984) ha per esempio suggerito di limitare la dimensione dei messaggi a ciò che trova posto nei registri di macchina, e di realizzare quindi lo scambio di messaggi utilizzando i registri.

3.5.3 Il problema del produttore-consumatore risolto con lo scambio di messaggi

Vediamo ora come si può risolvere il problema del produttore-consumatore mediante lo scambio di messaggi e senza alcuna memoria condivisa. Una soluzione viene mostrata in figura 14.
Assumiamo che tutti i messaggi abbiano la stessa dimensione e che i messaggi inviati, ma non ancora ricevuti, siano automaticamente inseriti dal sistema operativo in un buffer. In questa soluzione usiamo un totale di N messaggi, in qualche modo analoghi alle N posizioni del buffer in memoria condivisa. Il consumatore inizia inviando N messaggi vuoti al produttore. Quando il produttore ha un oggetto da inviare al consumatore, prende un messaggio vuoto e ne rimanda indietro uno pieno. In questo modo, il numero totale dei messaggi nel sistema rimane costante nel tempo e i messaggi possono essere salvati in uno spazio di memoria le cui dimensioni sono note in anticipo.
Se il produttore lavora più velocemente del consumatore, tutti i messaggi finiranno con l'essere pieni e in attesa del consumatore; il produttore sarà bloccato, attendendo che un messaggio vuoto torni indietro. Se il consumatore lavora più velocemente, accade il contrario: tutti i messaggi resteranno vuoti, in attesa che il produttore li riempia; il consumatore sarà bloccato, in attesa di un messaggio pieno.
Lo scambio di messaggi consente di realizzare un gran numero di varianti. Per iniziare, consideriamo il modo in cui vengono indirizzati i messaggi. Un metodo consiste nell'assegnare a ogni processo un indirizzo unico e nel far sì che i messaggi siano indirizzati ai processi. Un metodo differente consiste nell'inventare una nuova struttura dati, chiamata mailbox (casella postale). Una mailbox è un posto in cui immagazzinare un certo numero di messaggi, numero in genere specificato al momento della creazione della mailbox. Quando si usano le mailbox i parametri di indirizzamento nelle chiamate SEND e RECEIVE sono mailbox, non processi. Quando un processo tenta di inviare a una mailbox piena viene sospeso fino a che un messaggio non viene rimosso dalla mailbox, lasciando il posto al nuovo messaggio.
Nel problema del produttore-consumatore, sia il produttore che il consumatore creano mailbox abbastanza grandi da contenere N messaggi. Il produttore invia messaggi contenenti dati alla mailbox del consumatore e il consumatore invia messaggi vuoti alla mailbox del produttore. Quando si usano le mailbox il meccanismo di funzionamento dei buffer è chiaro: la mailbox di destinazione trattiene i messaggi che sono stati inviati al processo destinatario, ma non sono stati ancora accettati.

Figura 14 - Il problema produttore consumatore con N Messaggi

```
# define N 100 /* numero di posizioni nel buffer */
void producer(void) {
     int item;
     message m; /* buffer dei messaggi */
     while (TRUE) {
        produce_item(&item);
        receive(consumer, &m); /* attendi un vuoto */
        build_message(&m); /* costruisci il messaggio da inviare */
        send(consumer, &m); /* invia il messaggio al consumatore */
     }
}

void consumer(void) {
     int item, i;
     message m;
     for (i = 0; i < N; i++)
             send (producer, &m); / * invia N vuoti */
     while (TRUE) {
             receive(producer, &m); /* estrai il messaggio pieno */
             extract_item(&m, &item); /* estrai i dati dal messaggio */
             send (producer, &m); /* restituisce un messaggio vuoto */
             consume _item(item); /* consuma i dati */
     }
}
```

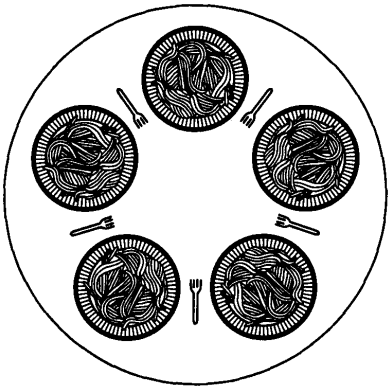
L'alternativa opposta alle mailbox consiste nell'eliminare ogni forma di buffering. Quando si segue quest'approccio, se si chiama la SEND prima della RECEIVE, il processo mittente rimane bloccato fino all'esecuzione della RECEIVE, quando il messaggio può essere copiato direttamente e senza alcun buffer intermedio dal mittente al destinatario. Analogamente, se si esegue prima la RECEIVE, il destinatario resta bloccato fino all'invocazione della SEND. Questa strategia viene spesso chiamata rendezvous. Il rendezvous è più facile da implementare di uno schema a messaggi con buffer, ma è meno flessibile in quanto mittente e destinatario devono per forza essere eseguiti in modo sincronizzato.
I sistemi a messaggi devono anche trattare il problema di come assegnare i nomi ai processi in modo che il processo specificato in una chiamata SEND o RECEIVE venga identificato senza ambiguità. L'autenticazione è un ulteriore problema dei sistemi a messaggi: come può il cliente sapere che sta comunicando con il file server reale, e non con un impostore?
All'altro estremo dello spettro ci sono i problemi progettuali che diventano rilevanti quando sia il mittente che il destinatario sono sulla stessa macchina. Uno di questi è il problema delle prestazioni. La copia dei messaggi da un processo all'altro è sempre più lenta di un'operazione semaforica o dell'accesso a un monitor. Si è fatto molto per rendere efficiente lo scambio di messaggi. Cheriton (1984) ha per esempio suggerito di limitare la dimensione dei messaggi a ciò che trova posto nei registri di macchina, e di realizzare quindi lo scambio di messaggi utilizzando i registri. L'uso dei buffer è chiaro: la mailbox di destinazione trattiene i messaggi che sono stati inviati al processo destinatario, ma non sono stati ancora accettati.
La comunicazione interprocesso tra i processi utente di UNIX avviene mediante le pipe, che sono di fatto delle mailbox. La sola vera differenza tra un sistema a messaggi con mailbox e il meccanismo delle pipe è che queste ultime non preservano i confini tra i messaggi. In altre parole, se un processo scrive su una pipe 10 messaggi di 100 byte e un altro processo legge dalla pipe 1000 byte, il lettore otterrà in un sol colpo tutti e 10 i messaggi. In un vero sistema a messaggi ogni READ dovrebbe restituire un solo messaggio. Ovviamente non ci sono problemi se i processi si accordano per scrivere e leggere sempre dalla pipe messaggi di dimensione fissa, o per terminare ciascun messaggio con un carattere speciale (ad esempio un line-feed).

3.6 Problemi classici di IPC

(IPC = InterProcess Communication)
La letteratura sui sistemi operativi è piena di problemi interessanti che sono stati ampiamente discussi e analizzati. Nelle sezioni seguenti esamineremo tre dei problemi più conosciuti.

3.6.1 Problema dei filosofi affamati

Figura 15 - La tavola dei filosofi affamati



Nel 1965 Dijkstra formulò e risolse un problema di sincronizzazione che chiamò il problema dei filosofi affamati. Da allora, chiunque inventi un'ulteriore primitiva di sincronizzazione si è sentito in obbligo di dimostrare l'efficacia della nuova primitiva, mostrando quanto elegantemente risolva il problema dei filosofi affamati. Il problema può essere formulato in modo molto semplice come indicato di seguito. Cinque filosofi sono seduti intorno a un tavolo circolare. Ognuno di loro ha un piatto di spaghetti, tanto scivolosi che ogni filosofo ha bisogno di due forchette per mangiarli. Tra ogni coppia di piatti c'è una forchetta. Lo schema della tavola è mostrato in figura 15.

La vita di un filosofo consiste di periodi in cui pensa, alternati a periodi in cui mangia (questa è un'astrazione, persino per dei filosofi, ma le altre attività sono in questo contesto irrilevanti). Quando un filosofo ha fame cerca di ottenere la forchetta alla sua sinistra e quella alla sua destra, una alla volta e in qualsiasi ordine. Se riesce a ottenere le due forchette, mangia per un po', quindi posa le forchette e riprende a pensare. La domanda chiave è: si può scrivere un programma per ciascun filosofo che faccia ciò che si suppone debba fare e non si blocchi mai? (È stato fatto notare come il requisito di avere due forchette sia in qualche modo artificioso; forse dovremmo passare dal cibo italiano a quello cinese, sostituendo il riso agli spaghetti e le bacchette alle forchette.)
La figura 16 mostra la soluzione ovvia. La procedura prendi_la_forchetta attende che la forchetta specificata sia disponibile e quindi la afferra. Sfortunatamente, questa soluzione oltre a essere ovvia è anche sbagliata. Supponiamo che tutti e cinque i filosofi prendano simultaneamente la loro forchetta sinistra. Nessuno sarà in grado di prendere la forchetta destra e avremo un deadlock.
Potremmo modificare il programma in modo che, dopo aver preso la forchetta sinistra, controlli che la forchetta destra sia disponibile. Se non lo è, il filosofo posa la forchetta sinistra, aspetta un po', e quindi ripete l'intera procedura. Anche questa proposta fallisce, benché per un motivo differente. Con un po' di sfortuna, tutti i filosofi potrebbero iniziare l'algoritmo simultaneamente, prendendo la loro forchetta sinistra, per poi rendersi conto che la forchetta destra non è disponibile, mettendo quindi a posto la forchetta sinistra e aspettando per prendere nuovamente la forchetta sinistra in modo simultaneo, e così via, per sempre. Una situazione come questa, nella quale tutti i programmi procedono senza termine e senza ottenere alcun progresso, viene chiamata starvation (inedia, o morire di fame; viene chiamata starvation anche quando il problema non ha luogo in un ristorante italiano o cinese).

Figura 16 - Una non-soluzione al problema dei filosofi affamati

```
#define N 5 /* numero di filosofi
void filosofo(int i) /* i: indice del filosofo, da 0 a 4
while (TRUE) {
     pensa ( ); /* il filosofo sta pensando */
     prendi_la_forchetta(i); /* prendi la forchetta sinistra */
     prendi_la_forchetta((i+1) % N); /* prendi la forchetta destra; % è l'operatore modulo
     mangia ( ); /* yum-yum, spaghetti */
     posa_la_forchetta(i); /* rimetti la forchetta sinistra sul tavolo */
     posa_la_forchetta((i+1) % N); /* rimetti la forchetta destra sul tavolo */
}
```

Ora potreste pensare: "Se quando non riescono a ottenere la forchetta destra i filosofi attendessero semplicemente un tempo casuale, anziché lo stesso tempo, la probabilità che tutti continuino a procedere raggruppati per un tempo di circa un'ora è molto piccola, ma vorremmo una soluzione che funzioni sempre e non possa fallire a causa di una sequenza improbabile di numeri casuali (si pensi ai controlli di sicurezza di una centrale nucleare).
Un miglioramento rispetto alla figura 16, che non comporta nessun deadlock o starvation

Figura 17 - Una soluzione al problema dei filosofi affamati

```
#define N 5 /* numero dei filosofi
#define SINISTRA (i-1)%N /* indice del vicino di sinistra di */
#define DESTRA (i+1)%N /* indice del vicino di destra di */
#define PENSANDO 0 /* il filosofo sta pensando*/
#define AFFAMATO 1 /* il filosofo sta cercando di prendere le forchette
#define MANGIANDO 2 /* il filosofo sta mangiando
typedef mt semaphore; /* i semafori sono degli interi particolari*/
int state[N]; /* l'array che memorizza gli stati di tutti */
semaphore mutex = 1; /* mutua esclusione per le sezioni critiche */
semaphore s[N]; /* un semaforo per ogni filosofo */
void filosofi (int i) { /* i: indice del filosofo, da O a N-1 */
     while (TRUE) { /* ripeti all'infinito*/
            pensa ( ); /* il filosofo sta pensando*/
        prendi_le_forchette(i); /* prendi le due forchette o bloccati */
        mangia( ); /* yum-yum, spaghetti */
        posa_le_forchette(i); /* rimetti entrambe le forchette sul tavolo
    }
}
void prendi_la_forchetta(int i) { /* i: indice del filosofo, da O a N-1 */
     wait(&mutex); /* entra in sezione critica */
     state[i] = AFFAMATO; /* registra che il filosofo i è affamato */
     test(i); /* cerca di ottenere le 2 forchette */
     SIGNAL(&mutex); /* esce dalla sezione critica */
     wait(&s[i]); /* si blocca se non ha ottenuto le forchette */
}
void posa_la_forchetta(i) { /* i: indice del filosofo, da O a N-1 */
     wait(&mutex); /* entra in sezione critica */
     state(i) = PENSANDO; /* il filosofo ha finito di mangiare */
     test(SINISTRA); /* controlla se il vicino di sinistra può mangiare */
     test(DESTRA); /* controlla se il vicino di destra può mangiare */
     SIGNAL(&mutex); /* esce dalla sezione critica */
}
void test(i) {/* i: indice del filosofo, da O a N-1 */
     if (state[i] == AFFAMATO
            && state[SINISTRA] != MANGIANDO
            && state[DESTRA] != MANGIANDO) {
     state[i] = MANGIANDO;
     SIGNAL(&s[i]);
     }
}
```

3.6.2 Problema dei lettori e degli scrittori

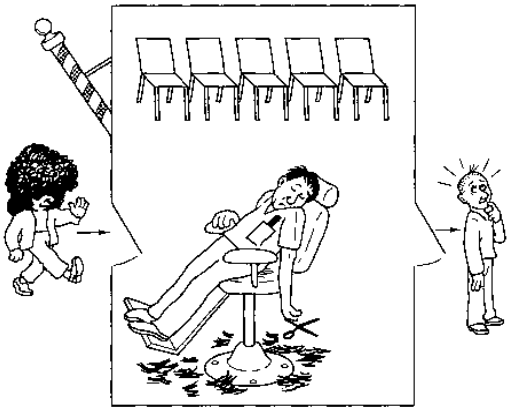
Il problema dei filosofi affamati è utile per modellare il caso di processi che competono per l'accesso esclusivo a un numero limitato di risorse, come i dispositivi di I/O. Un altro problema famoso è il problema dei lettori e degli scrittori (Courtois et al., 1971), che modella l'accesso a una base di dati. Si immagini, ad esempio, un sistema di prenotazioni aeree, in cui molti processi competono per leggere e scrivere i dati. È accettabile che più processi leggano nello stesso istante la base di dati, ma se un processo sta aggiornando (scrivendo) la base di dati, nessun altro processo può accedere alla base di dati, neanche i lettori. Il problema è: come si programmano i lettori e gli scrittori? Una soluzione è mostrata in figura 18.
In questa soluzione, il primo lettore che accede alla base di dati esegue una WAIT sul semaforo db. I lettori seguenti si limitano a incrementare un contatore, rc. I lettori, uscendo, decrementano il contatore e l'ultimo ad uscire esegue una SIGNAL sul semaforo, consentendo l'ingresso di un eventuale scrittore bloccato.
La soluzione qui presentata contiene implicitamente una scelta di non facilissima comprensione e che vale la pena di commentare. Supponiamo che, mentre un lettore sta usando la base di dati, un altro lettore arrivi. Poiché non c'è nessun problema se due lettori usano la risorsa nel medesimo istante, al secondo lettore viene consentito l'accesso. Nello stesso modo si può far entrare un terzo lettore, così come i lettori seguenti, se arrivano insieme.
Supponiamo ora che arrivi uno scrittore. Lo scrittore non può accedere alla base di dati, poiché gli scrittori devono ottenere un accesso esclusivo, di conseguenza lo scrittore viene sospeso.
In seguito arrivano altri lettori. Finché almeno un lettore è attivo, i lettori che seguono vengono fatti accedere alla risorsa. Come conseguenza di questa strategia, tutti i lettori vengono immediatamente ammessi fintanto che si ha un flusso costante di lettori. Lo scrittore resterà sospeso fino a che non rimane nessun lettore. Se, per esempio, ogni 2 secondi si ha l'arrivo di un nuovo lettore, e ogni lettore impiega 5 secondi per svolgere le sue operazioni, lo scrittore non riuscirà mai ad accedere alla base di dati
Per prevenire questa situazione, il programma potrebbe essere scritto in modo leggermente differente: quando un lettore arriva e uno scrittore è in attesa, il lettore viene sospeso in coda allo scrittore invece di essere immediatamente ammesso. In questo modo, lo scrittore deve aspettare che terminino i lettori che erano attivi al momento del suo arrivo, ma non i lettori arrivati in seguito. Lo svantaggio di questa soluzione è che consente una minore concorrenza e quindi prestazioni inferiori. In Courtois et al. (1971) viene mostrata una soluzione che dà priorità agli scrittori. Per i dettagli, si faccia riferimento al loro articolo.

Figura 18 - Una soluzione al problema dei lettori-scrittori

```
typedef int semaphore;
semaphore mutex = 1; /* controlla l'accesso a 'rc' */
semaphore db = 1; /* controlla l'accesso al data base */1
int rc = 0 /* # di processi in lettura o che vogliono leggere */
void lettore(void) {
    while (TRUE) { /* ripeti all'infinito
            wait(&mutex); /* ottieni l'accesso esclusivo a 'rc' */
            rc = rc + 1; /* c'è un lettore in più */
            if (rc == 1) wait(&db); /* se è il primo lettore ...
            SIGNAL(&mutex); /* rilascia l'accesso esclusivo a 'rc' */
            leggi_data_base ( ); /* accedi ai dati */
            wait(&mutex); /* ottieni l'accesso esclusivo a 'rc' */
            rc = rc - 1; /* c'è un lettore in meno */
            if (rc == 0) /* se è l'ultimo lettore ... */
            SIGNAL(&db);
            SIGNAL(&mutex); /* rilascia l'accesso esclusivo a 'rc'*/
            usa_dat_letti ( ); /* sezione non critica */
    }
}
void scrittore(void) {
    while (TRUE) { /* ripeti all'infinito */
    rifletti_sui_dati ( ); /* sezione non critica */
    wait(&db); /* ottieni l'accesso esclusivo */
    scrivi_data_base ( ); /* aggiorna i dati */
    SIGNAL(&db); /* rilascia l'accesso esclusivo */
}
```

3.6.3 Problema del barbiere addormentato

Figura 19 - Il barbiere addormentato



Un altro problema classico di IPC ha luogo nella bottega di un barbiere. Nel salone ci sono un barbiere, una sedia da barbiere e n sedie per far sedere i clienti in attesa, se ve ne sono. Se non ci sono clienti, il barbiere si siede nella sedia e si addormenta, come illustrato in figura 19. Quando un cliente arriva sveglia il barbiere addormentato. Se altri clienti arrivano mentre il barbiere sta tagliando i capelli a un cliente, si siedono (se ci sono sedie vuote) o altrimenti abbandonano il negozio (se le sedie sono occupate). Il problema consiste nel programmare il barbiere e i clienti in modo da evitare corse critiche.

La nostra soluzione usa tre semafori: clienti, che conta i clienti in attesa (escluso il cliente sulla sedia da barbiere, che non è in attesa), barbieri, corrispondente al numero di barbieri fermi e in attesa di clienti (0 o 1), e mutex, utilizzato per la mutua esclusione. Si ha inoltre bisogno di una variabile, waiting, che conti i clienti in attesa. Essa è in sostanza un copia di clienti. Il motivo per cui serve waiting è che non c'è alcun modo di leggere il valore corrente di un semaforo. In questa soluzione, un cliente che entra nel salone deve contare il numero di clienti in attesa. Se tale numero è minore del numero delle sedie rimane, altrimenti esce.
La nostra soluzione è mostrata in figura 20. La mattina, quando il barbiere arriva al lavoro, esegue la procedura barbiere, che lo sospende sul semaforo clienti fino a che non arriva qualcuno. Il barbiere quindi si addormenta, come mostrato in figura 2-20.

Figura 20 - Una soluzione al problema del barbiere addormentato

```
#define CHAIRS 5 /* # di sedie per i clienti in attesa */
typedef int semaphore;
semaphore clienti = 0; /* # di clienti in attesa */
semaphore barbieri = 0; /* # di barbieri in attesa di clienti */
semaphore mutex = 1; /* per la mutua esclusione */
mt waiting = 0; /* ci sono clienti in attesa */
void barbiere(void) {
    while (TRUE) {
       wait(clienti); /* si sospende se il # di clienti è 0 */
       wait(mutex); /* accede a 'waiting' */
       waiting = waiting - 1; /* decrementa il numero di clienti in attesa */
       SIGNAL(barbieri); /* un barbiere è pronto a tagliare */
       SIGNAL(mutex); 1 /* rilascia 'waiting' */
       taglia_capelli( ); /* taglia i capelli (fuori dalla sezione critica) */
    }
}
void cliente(void) {
    wait(mutex); /* entra in sezione critica */
    if (waiting <CHAIRS) { /* se non ci sono sedie libere esci */
            waiting = waiting + 1; /* incrementa il numero di clienti in attesa */
       SIGNAL(clienti); /* sveglia il barbiere se necessario */
       SIGNAL(mutex); /* rilascia l'accesso a 'waiting' */
       wait(barbieri); /* si sospende se non ci sono barbieri liberi */
       viene_servito ( ); /* si siede e viene servito */
   }
```

```
    else {
        SIGNAL(mutex); /* il salone è pieno, esci */
      }
  }
```

Quando un cliente arriva esegue cliente, che inizia cercando di acquisire mutex per entrare in sezione critica. Un altro cliente che entri di lì a poco non potrà far nulla fino a che il primo non abbia rilasciato mutex. Il cliente controlla quindi che il numero dei clienti in attesa sia inferiore al numero di sedie. Se non è così, rilascia mutex ed esce senza taglio.
Se c'è una sedia disponibile, il cliente incrementa la variabile intera waiting ed esegue quindi una SIGNAL sul semaforo clienti, risvegliando il barbiere. A questo punto, il cliente e il barbiere sono entrambi svegli. Quando il cliente rilascia mutex, il barbiere acquisisce il semaforo, sbriga le sue faccende, e inizia quindi il taglio.
Quando il taglio è terminato il cliente abbandona la procedura ed esce dal negozio. Diversamente dai nostri esempi precedenti, il cliente non esegue alcun ciclo, poiché ogni cliente riceve un solo taglio. Il barbiere invece esegue un ciclo, cercando ogni volta di servire il cliente successivo. Se ne trova uno, somministra un altro taglio, altrimenti si addormenta.
A margine vale la pena notare come i problemi dei lettori-scrittori e del barbiere addormentato, benché non comprendano alcun trasferimento di dati, appartengono all'area dell'IPC, in quanto comportano una sincronizzazione tra più processi.

3.7 RIEPILOGO

Nel presente capitolo abbiamo introdotto la nozione di processo, confrontandola con quella di programma, sottolineando che si tratta di un concetto utile per lo studio delle caratteristiche dei sistemi operativi. In particolare, si è visto che sia la concorrenza che la non determinatezza possono essere descritte in termini di commutazione di processori tra processi. Abbiamo anche introdotto i semafori e le operazioni svolte su di essi come meccanismo per effettuare la comunicazione tra processi; si è inoltre mostrato che essi rappresentano uno strumento sufficiente a risolvere i problemi di mutua esclusione e sincronizzazione. Infine, si è visto che in alcuni casi è possibile usare determinate relazioni relative ai valori dei semafori per dimostrare che i processi possiedono le proprietà rispondenti alle nostre aspettative.
Nel capitolo successivo faremo uso di questi strumenti per avviare la costruzione del nostro sistema operativo ipotetico.

4 IL NUCLEO DEL SISTEMA

Nel capitolo che precede abbiamo sviluppato i concetti e gli strumenti necessari per costruire il sistema operativo ipotetico delineato nel capitolo 1. Come ricordato in quella sede, il nostro sistema operativo ipotetico assomiglierà ad una cipolla nella quale ogni strato fornisce un insieme di funzioni che dipendono esclusivamente dagli strati più interni. Al centro della cipolla si trovano le risorse fornite dall'hardware della macchina stessa. Gli altri strati possono essere considerati come se implementassero macchine virtuali successive, cosicché la cipolla nel suo complesso rappresenterà la macchina virtuale richiesta dall'utente.
L'interfaccia principale tra l'hardware di base della macchina ed il sistema operativo è fornita dal nucleo del sistema, che rappresenta lo strato più interno della cipolla. Lo scopo del nucleo è quello di fornire un ambiente nel quale possano esistere i processi; ciò comporta la gestione degli interrupt, la commutazione di processori tra i processi e l'implementazione dei meccanismi per la comunicazione tra i processi. Prima di passare alla descrizione dettagliata di queste funzioni, daremo uno sguardo all'hardware indispensabile per supportare il sistema operativo che tenteremo di costruire.

4.1 RISORSE HARDWARE ESSENZIALI

4.1.1 Meccanismo degli interrupt

Nel capitolo 2 abbiamo detto che, per poter sovrapporre le attività di I/O all'elaborazione centrale, deve essere possibile interrompere il processo in corso quando è completato un trasferimento tra periferiche. Pertanto, chiediamo al nostro elaboratore di fornire un meccanismo di interruzione che per lo meno salvi il valore del contatore di programma per il processo interrotto e trasferisca il controllo ad una locazione fissa di memoria. Quest'ultima verrà utilizzata come punto di partenza per un (segmento di) programma che va sotto il nome di interrupt routine o di interrupt handler (gestore degli interrupt), il cui scopo consiste nell'individuare la fonte dell'interrupt e nel rispondervi in modo idoneo. Descriveremo il gestore degli interrupt nel paragrafo 4.4 ed esamineremo le varie forme che può assumere a seconda dell'esatta natura del meccanismo di interruzione a disposizione.
Per dare compiutezza alla nostra esposizione, ricordiamo che alcuni elaboratori, per esempio la gamma CDC CYBER 170, funzionano senza disporre di un meccanismo esplicito di interrupt. In un elaboratore del genere, uno o più processori devono essere dedicati alla supervisione dello stato dei dispositivi di I/O per rilevare il completamento dei trasferimenti. Nella fattispecie, per il CYBER tale funzione è svolta da un certo numero di cosiddetti "processori periferici", che sgravano così la CPU da tutta la gestione dell'I/O. Esiste comunque, anche in questo caso, una forma ristretta di interrupt, poiché i processori periferici possono costringere la CPU a saltare ad una locazione diversa.

4.1.2 Protezione della memoria

Quando parecchi processi girano contemporaneamente in macchina, è necessario proteggere la parte di memoria utilizzata da un processo contro l'accesso non consentito ad opera di un altro. I meccanismi di protezione che devono essere inclusi nell'hardware di indirizzamento della memoria sono oggetto di descrizione particolareggiata nel prossimo capitolo; per il momento daremo per scontata la loro esistenza.

4.1.3 Insieme di istruzioni privilegiate

Col fine di rendere impossibili le interferenze reciproche di processi concorrenti, parte del repertorio di istruzioni dell'elaboratore deve essere riservato all'uso esclusivo da parte del sistema operativo. Queste istruzioni privilegiate svolgono funzioni quali:

a. abilitazione e disabilitazione degli interrupt;
b. commutazione di un processore tra processi;
c. accesso ai registri utilizzati dall'hardware di protezione della memoria;

d. esecuzione dell'input o dell'output;
e. arresto del processore centrale.

Per distinguere tra i momenti in cui è concesso o meno utilizzare istruzioni privilegiate, la maggioranza degli elaboratori opera in due modi, di solito denominati modo privilegiato (supervisor mode) e modo utente o non privilegiato (user mode). Le istruzioni privilegiate sono permesse solamente nel modo privilegiato. Il passaggio dal modo utente al modo privilegiato ha luogo automaticamente al verificarsi di una delle seguenti situazioni:

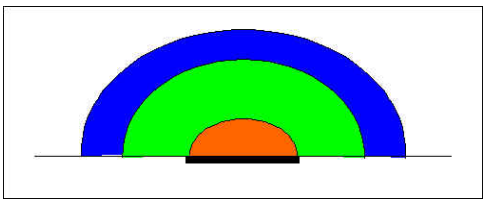
a. un processo utente richiama il sistema operativo per eseguire una qualche funzione che richieda l'impiego di un'istruzione privilegiata; tale richiamo viene definito extracode o supervisor call;
b. si verifica un interrupt;
c. si verifica una condizione d'errore all'interno di un processo utente. La situazione può essere considerata come un "interrupt interno" ed essere gestita in primo luogo da una routine per la gestione degli interrupt;
d. viene fatto un tentativo di eseguire un'istruzione privilegiata mentre ci si trova nel modo non privilegiato. Il tentativo può essere visto come un tipo particolare di errore e gestito come nel precedente punto (c).

Il ritorno dal modo privilegiato a quello non privilegiato viene effettuato tramite un'istruzione, privilegiata essa stessa.
Bisognerebbe forse ricordare che lo schema esposto prima, riguardante due livelli di privilegio, in alcuni elaboratori (per esempio l'ICL 2900, il CYBER 180 e l'Honeywell 645) viene esteso fino ad includere vari livelli. Rimandiamo a più oltre, però, la trattazione delle macchine a livelli multipli.

4.1.4 Orologio in tempo reale

Per mettere in atto i meccanismi di schedulazione e per addebitare le risorse utilizzate dai vari utenti è fondamentale che l'hardware includa un orologio il quale provveda ad eseguire degli interrupt ad intervalli prefissati di tempo reale (di tempo, quindi, misurato nel mondo esterno e distinto da quello richiesto dalle attività di calcolo messe in atto da un qualsiasi processo particolare).
Da questo punto in poi, presupponiamo che la macchina sulla quale è in corso di costruzione il nostro sistema operativo ipotetico possieda le risorse hardware che abbiamo appena delineato. Più precisamente, siccome ammettiamo la possibilità che in una singola configurazione siano compresi svariati processori centrali, supporremo che ognuno di essi possieda le prestazioni di cui al numero (1), (2) e (3), e che esista un unico orologio in grado di intervenire in tempo reale per interrompere i processi stessi. Per di più, ci limiteremo alle configurazioni strettamente accoppiate, nelle quali i processori sono identici e condividono una memoria comune. Con ciò si esclude di prendere in considerazione le reti di calcolo, in cui i processori hanno memorie distinte e comunicano tramite un qualche tipo di trasmissione dei dati, nonché i sistemi dove processori differenti sono organizzati così da suddividersi un certo carico di lavoro nel modo che più si confà alle loro caratteristiche specifiche. Il primo dei due argomenti merita un libro a parte, mentre il secondo è ancora un fertile campo di ricerca nel quale non sono ancora emerse tendenze chiare.

Figura 21 - Il sistema operativo ipotetico



La linea orizzontale alla base dello schema rappresenta l'hardware del calcolatore; il segmento in neretto della linea è l'insieme delle istruzioni privilegiate concernenti la protezione della memoria e l'I/O, che saranno trattate nei capitoli seguenti.

Il nucleo è formato da tre programmi:

1. il first-level interrrupt handler (gestore degli interrupt di primo livello), che svolge la funzione di gestire inizialmente tutti gli interrupt ;
2. il (dispatcher (routine di smistamento) che distribuisce i processori centrali tra i processi;
3. due procedure (routine) che implementano le primitive riguardanti la comunicazione tra processi (wait e signal) già descritte nel capitolo 3. Queste procedure vengono richiamate tramite supervisor call nei processi interessati.

4.2 DESCRIZIONE DEL NUCLEO

Poiché il nucleo è costruito direttamente sull'hardware di base, possiamo facilmente prevedere che si tratti di quella parte del sistema operativo più pesantemente dipendente dalla macchina. Infatti, si tratta probabilmente dell'unica parte del sistema operativo che necessita di essere scritta sostanzialmente in linguaggio Assembler; a parte poche eccezioni di scarso rilievo, prevalentemente riferite all'I/O, per gli altri strati è più conveniente la codifica per mezzo di un linguaggio ad alto livello. I linguaggi più idonei alla programmazione dei sistemi, sui quali eventualmente il lettore potrà documentarsi, sono il RCPI. (Richards, 1969), il BLISS (Wulfetal., 1971) cd il C (Kernighan e Ritchie, 1978). Vi sono altri linguaggi, quali il Concurrent Pascal (Hansen, 1975) ed il Modula (Wirth, 1977 e 1983), che sono intesi specificamente per la stesura dei sistemi operativi, benché impongano certe limitazioni al progettista di sistemi, Il fatto che il linguaggio assembler sia limitato ad una piccola parte (presumibilmente non oltre le 400 istruzioni del sistema operativo, cosa resa possibile dalla struttura gerarchica del sistema, può contribuire in modo determinante alla realizzazione dell'obiettivo di un prodotto senza errori, comprensibile e di facile manutenzione.

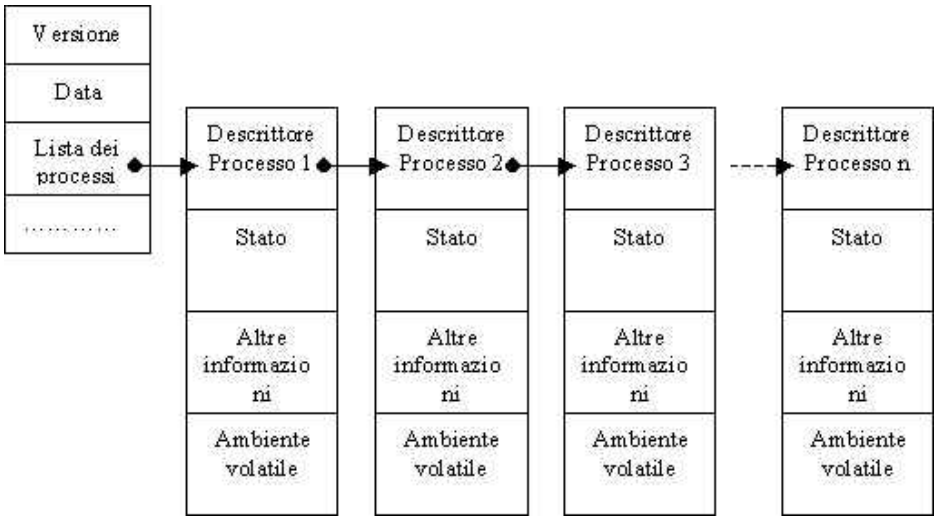
4.3 LA RAPPRESENTAZIONE: DEI PROCESSI

I programmi che si trovano all'interno del nucleo sono addetti al mantenimento dell'ambiente entro cui possono esistere i processi. Di conseguenza, è necessario che operino su una qualche struttura dati che sia la rappresentazione fisica di tutti i processi esistenti entro il sistema. La natura di questa struttura dati è l'argomento che tratteremo nel presente paragrafo.
Ogni processo può essere rappresentato tramite un descrittore di processo (a volte noto come blocco di controllo o vettore di stato) che è una zona di memoria contenente tutte le informazioni importanti riguardanti il processo stesso. La composizione di queste informazioni risulterà più chiara andando avanti; per il momento diciamo che vi è incluso un qualche tipo di identificazione del processo (il nome del processo) ed un'indicazione del suo stato. Un processo può trovarsi in uno dei tre seguenti stati principali: in esecuzione, quando ha a disposizione un processore; in attesa di esecuzione (in stato di pronto), stato da cui potrebbe evolversi se gli venisse dedicato un processore; oppure può essere non eseguibile (in attesa di eventi), e quindi impossibilitato a fare uso di un processore anche se quest'ultimo vi fosse dedicato. Il motivo più comune per cui un processo non è eseguibile è che si trovi in attesa del completamento di un

trasferimento tra periferiche. Lo stato di un processo è un'informazione essenziale per la routine di smistamento, quando quest'ultima arriva al punto di allocare un determinato processore centrale.
Denomineremo un processo in corso di esecuzione su un processore centrale processo corrente per quel processore; naturalmente il numero dei processi correnti all'interno del sistema in ogni dato momento è minore o uguale al numero dei processori a disposizione.
Un'altra parte del descrittore può essere utilizzata per memorizzare tutte quelle informazioni concernenti il processo che bisogna salvare quando quest'ultimo perde il controllo di un processore. Queste informazioni, necessarie per riprendere successivamente il processo, comprendono i valori di tutti i registri di macchina, quali il contatore di programma, gli accumulatori ed i registri indice, che potrebbero essere modificati da un altro processo. Comprendono inoltre i valori di eventuali registri usati per l'indirizzamento della memoria che fa capo al processo (si veda il capitolo 5), il puntatore allo stack, lo stato dei file aperti, il tempo di CPU utilizzato e le informazioni di scheduling, ecc. Tutte queste informazioni nel complesso vengono da noi definite ambiente volatile del processo (altri nomi ricorrenti nella letteratura specializzata sono context block (blocco di contesto) e process state (stato del processo). Per essere più espliciti, si può definire l'ambiente volatile di un processo come quel sottoinsieme di risorse di sistema, condivisibili e modificabili, cui il processo stesso può avere accesso.
Il descrittore di ciascun processo viene inserito in una lista dei processi che contiene una descrizione di tutti i processi esistenti all'interno del sistema. Per il momento adotteremo un modello elementare di lista dei processi, nel quale i descrittori sono collegati tramite una lista semplice. La lista dei processi è la prima struttura dati da noi introdotta nel nostro sistema operativo, ma dal momento che non sarà affatto l'ultima, introduciamo qui anche una tabella centrale il cui scopo è quello di servire come mezzo di accesso a tutte le strutture di sistema. La tabella centrale conterrà un puntatore per ciascuna struttura dati e potrà anche essere impiegata per conservare altre informazioni globali, quali la data, l'ora ed il numero della versione del sistema, che potrebbero essere richieste di tanto in tanto. La tabella centrale e la lista dei processi sono illustrati in figura 22.

Figura 22 - Tabella centrale e lista dei processi

TABELLA CENTRALE



In molti sistemi operativi la gestione dei processi, della memoria e dei file è realizzata separatamente da moduli diversi all'interno del sistema; di conseguenza la tabella dei processi è partizionata, e ogni modulo gestisce i campi di sua pertinenza. La figura 22bis mostra alcuni dei campi più importanti. Solo gli attributi della prima colonna verranno trattati in questo capitolo. Le altre due colonne sono mostrate solo per dare un idea di quali informazioni siano necessarie al resto del sistema.

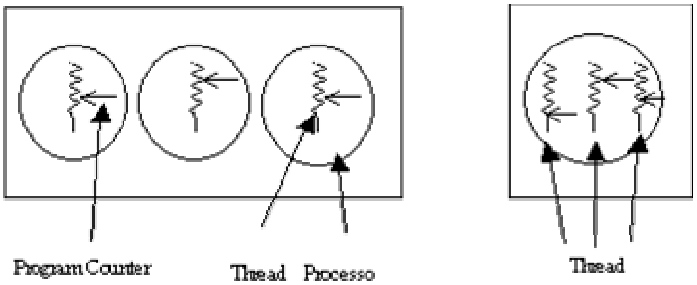
Figura 22bis - esempio di alcuni campi nella tabella dei processi

Gestione del processi	Gestione della memoria	Gestione dei file
Registri	Puntatore al segmento codice	Maschera UMASK
Parola di stato (PSW)	Puntatore al segmento dati	Directory radice
Contatore di istruzione	Puntatore al segmento bss	Directory di lavoro
Puntatore allo stack	Stato di uscita	Descrittori dei file
Stato del processo	Stato dei segnali	Uid effettivo
Istante In cui il processo è partito Tempo di CPU usato	Id del processo	Gid effettivo
Istante del prossimo allarme	Processo padre	Parametri della chiamata
Puntatore alla coda dei messaggi	Id del gruppo	Svariati bit di flag
Bit dei segnali pendenti	Uid reale	
Id del processo	Uid effettivo	
Svariati bit di flag	Gid reale	
	Gid effettivo	
	Bit map dei segnali	
	Svariati bit di flag	

4.3.1 I thread

In un tradizionale modello a processi, del tipo che abbiamo appena studiato, c'è una singola sequenza di controllo e un solo program counter per ogni processo. Ciononostante, in alcuni moderni sistemi operativi, il sistema fornisce il supporto per definire sequenze di controllo multiple all'interno di un processo. Queste sequenze di controllo sono solitamente chiamate thread o, a volte, processi lightweight (leggeri).
In figura 22ter sono mostrati tre processi tradizionali. Ogni processo ha il suo spazio di indirizzamento e una sola sequenza di controllo. Di contro, in figura 22ter(b) si ha un solo processo con tre sequenze di controllo. Benché in entrambi i casi vi siano tre thread, in figura 22ter(a) ognuno di essi opera in uno spazio di indirizzamento distinto, mentre in figura 22ter(b) tutti e tre condividono lo stesso spazio di indirizzamento.
Per mostrare un esempio di come si possano utilizzare i thread consideriamo un processo file server che riceve richieste di lettura e scrittura sui file e rimanda indietro i dati richiesti o accetta i dati aggiornati. Per migliorare le prestazioni, il server gestisce in memoria una cache dei file usati più di recente, leggendo da e scrivendo sulla cache per quanto possibile. Questa situazione si presta a rappresentare il modello di figura 22ter(b). Quando arriva una richiesta viene passata a un thread che la elabora, mentre gli altri thread restano a disposizione, in modo che il server possa continuare a soddisfare le nuove richieste anche mentre sta eseguendo un I/O da disco.

Figura 22ter - (a) Tre processi ognuno di un thread. (b) Un processo con tre thread.



Il modello di figura 22ter(a) non è adatto a questo scopo, perché è essenziale che tutti i thread del file server accedano alla stessa cache, mentre i tre thread di figura 22ter(a) non condividono lo stesso spazio di indirizzamento e non possono pertanto condividere la stessa cache in memoria.
Un altro esempio in cui i thread si dimostrano utili sono i browser per World Wide Web, come Netscape e Mosaic. Molte pagine Web contengono un buon numero di piccole immagini. Per ogni immagine contenuta in una pagina Web il browser deve stabilire una diversa connessione verso il sito relativo alla pagina e richiedere quindi l'immagine. Una gran quantità di tempo viene persa per l'effettuazione e la chiusura di tutte queste connessioni. Se il browser comprende più thread è possibile richiedere molte immagini allo stesso tempo, migliorando sostanzialmente le prestazioni nella maggior parte dei casi, visto che, nel caso di immagini piccole, il fattore dominante diventa il tempo necessario allo stabilirsi della connessione e non la velocità della linea di trasmissione.
Quando vi sono più thread nello stesso spazio di indirizzamento, alcuni dei campi riportati in figura 22 e 22bis non sono relativi al processo, ma al thread, e impongono quindi la presenza di una separata tabella dei thread, avente una voce per ogni thread. Tra le informazioni necessarie a ogni thread vi sono il program counter, i registri e lo stato. Il program counter è necessario in quanto i thread, così come i processi, possono essere sospesi e poi ripresi. I registri sono necessari perché, quando i thread vengono sospesi, i loro registri devono essere salvati. Infine, i thread, come i processi, possono essere in modalità di esecuzione, pronto o bloccato (running, ready o blocked).
In alcuni sistemi, il sistema operativo non si accorge dell'esistenza dei thread. In altre parole, essi vengono completamente gestiti nello spazio utente. Ad esempio, quando un thread sta per bloccarsi sceglie e avvia il suo successore prima di sospendersi. Diversi package che gestiscono i thread a livello utente sono entrati nell'uso comune; tra questi i P-thread POSIX, e i package C-threads di Mach.
In altri sistemi, il sistema operativo è a conoscenza dell'esistenza di più thread per ogni processo, in modo che, quando un thread si blocca, è il sistema operativo a scegliere il thread successivo da mandare in esecuzione, sia esso parte dello stesso processo o di uno diverso. Per effettuare lo scheduling, il nucleo deve avere una tabella che elenca tutti i thread presenti nel sistema, in modo analogo alla tabella dei processi.
Benché queste due alternative possano sembrare equivalenti, differiscono considerevolmente in termini di prestazioni. Una commutazione di thread è molto più veloce quando la loro gestione viene effettuata nello spazio utente piuttosto che nel caso in cui sia necessaria una chiamata al nucleo. Questo argomento depone ampiamente a favore di una gestione dei thread effettuata nello spazio utente. D'altra parte, quando i thread vengono gestiti interamente nello spazio utente e un thread si blocca (ad esempio in attesa di I/O o aspettando che termini la gestione di un'eccezione provocata dalla mancanza di frame di pagina in memoria, detta anche page fault), il nucleo blocca l'intero processo, non essendo a conoscenza dell'esistenza degli altri thread. Questa considerazione depone a favore di una gestione dei thread a livello del nucleo. Di conseguenza, entrambi i sistemi sono utilizzati, e oltre ad essi diversi schemi ibridi sono stati proposti (Anderson et al., 1992).

Sia che i thread vengano gestiti a livello utente che a livello del nucleo, essi introducono numerosi problemi che devono esser risolti e che modificano in modo significativo il modello di programmazione. Per iniziare, si consideri l'effetto della chiamata di sistema FORK. Se il processo genitore ha più di un thread, questo significa che il figlio deve contenere altrettanti thread? Se così non fosse, il processo potrebbe non funzionare correttamente, poiché ognuno di essi potrebbe essere essenziale al funzionamento.
Se invece il processo figlio ottiene tanti thread quanti ne sono contenuti nel genitore, cosa accadrebbe se, in quel momento, uno dei thread fosse, ad esempio, bloccato su di una READ da tastiera? Si avrebbero due thread bloccati in attesa di input da tastiera? Quando dal terminale viene inserita una riga, entrambi i thread ne ottengono una copia? Solo il genitore? O solo il figlio? Lo stesso problema si ha con le connessioni di rete già aperte.

Un altro insieme di problemi nasce dal fatto che i thread condividono un gran numero di strutture dati. Cosa succede se un thread chiude un file mentre un altro thread sta ancora leggendo da esso? Supponiamo che un thread si accorga di aver bisogno di memoria e inizi ad allocarsi più memoria, quindi, a metà strada, un nuovo thread viene mandato in esecuzione. Il nuovo thread si accorge della scarsità di memoria disponibile e inizia ad allocarsi memoria. L' allocazione avviene una o due volte? In quasi tutti i sistemi la cui progettazione è avvenuta senza tener conto dei thread, le librerie (come le procedure di allocazione di memoria) non sono rientranti, e provocano un crash se si effettua una seconda chiamata mentre la prima è ancora attiva.
La segnalazione degli errori può generare un ulteriore problema. In UNIX, dopo una chiamata di sistema, lo stato della chiamata viene inserito in una variabile globale, errno. Cosa succede se un thread effettua una chiamata di sistema e, prima che sia in grado di leggere errno, un altro thread effettua una chiamata di sistema sovrascrivendo il valore originario?
Consideriamo adesso i segnali. Alcuni segnali sono ovviamente specifici per i thread, mentre altri non lo sono. Ad esempio, se un thread chiama una ALARM, la logica vuole che il risultato ritorni al thread che ha effettuato la chiamata. Se il nucleo è a conoscenza dell'esistenza dei thread, esso si preoccupa che il thread giusto riceva il segnale.
Quando il nucleo non è a conoscenza dei thread, è il package dei thread che deve in qualche modo tener conto degli allarmi. Un'ulteriore complicazione per i thread a livello utente si ha quando (come in UNIX) un processo può avere un solo allarme pendente per volta e diversi thread chiamano indipendentemente la ALARM.
Altri segnali, come le interruzioni di tastiera, non fanno esplicito riferimento ad alcun thread. Chi li deve intercettare? Un thread designato? Tutti i thread? Un thread creato appositamente? Tutte queste soluzioni comportano dei problemi. Inoltre, cosa accade se un thread cambia la procedura di gestione dei segnali senza avvisare gli altri thread?
L'ultimo problema introdotto dai thread è la gestione dello stack. In molti sistemi, quando si ha un traboccamento (overflow) dello stack, il nucleo, in modo automatico, fornisce spazio di stack in aggiunta. Quando un processo ha più thread, deve anche avere più stack. Se il nucleo non è a conoscenza di tutti questi stack non li può neanche incrementare automaticamente in seguito a un traboccamento dello stack (stack fault). In effetti, non può neanche accorgersi che un errore nell'assegnazione della memoria (memory fault) è collegato a una crescita dello stack.
Questi problemi non sono certo insormontabili, ma mostrano come la semplice introduzione dei thread in un sistema preesistente non sia qualcosa che possa funzionare senza una sostanziale riprogettazione del sistema. Come minimo, la semantica delle chiamate di sistema deve essere ridefinita e le librerie devono essere riscritte. Tutte queste cose devono essere poi realizzate in modo da mantenere la compatibilità all'indietro con i programmi esistenti e quindi con il caso restrittivo che prevede un thread per ogni processo. Per ulteriori informazioni sui thread si consultino (Hauser et al., 1993 e Marsh et al., 1991).

4.4 FLIH

Il gestore degli interrupt di primo livello (first-level interrupt handler; FLIH) è la parte di sistema operativo responsabile delle risposte da fornire ai segnali provenienti sia dal mondo esterno (interrupt) che dall'interno dello stesso sistema di calcolo (segnalazione di errori e supervisor call). Designeremo collettivamente i due tipi di segnali con il nome di interrupt e faremo uso degli aggettivi "esterno" ed "interno" per distinguerli, quando sarà necessario. La funzione del FLIH è duplice:

1. determina l'origine dell'interrupt;
2. avvia la gestione dell'interrupt.

Come già detto nel paragrafo 4.1, al meccanismo degli interrupt in un elaboratore compete almeno il salvataggio del valore del contatore di programma per il processo interrotto. Si deve assicurare inoltre che vengano salvati adeguatamente gli altri registri richiesti dal FLIH ed eventualmente già usati dal processo interrotto. Se questo compito non venisse svolto dal meccanismo degli interrupt, dovrebbe essere eseguito prima di ogni altra cosa da parte dello stesso FLIH. Poiché quest'ultimo è un programma relativamente semplice che opera entro una zona di memoria riservata, l'insieme dei registri interessati non sarà vasto, magari pari solamente ad un singolo accumulatore. Certamente sarà notevolmente più piccolo dell'ambiente volatile relativo al processo interrotto, che non bisogna salvare nella sua interezza, dal momento che il processo può essere ripreso una volta che sia stato attuato l'intervento di risposta all'interrupt.
Una strategia alternativa al salvataggio dei valori dei registri, adottata, per esempio, sullo Zilog Z8O, sulla Serie 1 IBM e su alcuni modelli della gamma PDP-I1, consiste nel fornire un insieme aggiuntivo di registri da usarsi solamente nel modo privilegiato. Lo FLIH può, quindi, utilizzare questi registri lasciando intatti quelli del processo interrotto.

4.4.1 Skip chain

La determinazione dell'origine dell'interrupt può essere eseguita in maniera più o meno facile, a seconda dell'hardware disponibile. Nel caso dell'hardware più primitivo, nel quale tutti gli interrupt trasferiscono il controllo del processo ad una stessa locazione, si deve svolgere una sequenza di verifiche sui flag di stato relativi a tutte le possibili fonti di interrupt per giungere alla loro individuazione. Tale sequenza, frequentemente denominata skip chain, è illustrata in figura 23. Risulta chiaramente vantaggioso codificare la skip chain in modo che le cause che più frequentemente determinano l'interrupt si trovino vicino alla testa della stessa.
In alcuni calcolatori (per esempio il Pentium) la skip chain risulta superflua in quanto essi racchiudono un hardware particolare che distingue tra le fonti di interruzione, trasferendo il controllo a locazioni differenti a seconda della causa stessa. Ciò riduce il tempo necessario per individuare un interrupt a scapito, però, delle locazioni aggiuntive necessarie per la gestione degli interrupt. Una soluzione di compromesso cui ricorrono svariati calcolatori, ivi inclusa la serie IBM 370, è quella di fornire un numero limitato di locazioni per la gestione degli interrupt, ciascuna delle quali viene condivisa da un gruppo di dispositivi. Ad ogni classe di dispositivi di I/O (ad esempio floppy disk, hard disk, timer e terminali) viene di solito associata una locazione vicino al fondo della memoria, chiamata anche vettore di interruzione. La prima fase dell'individuazione degli interrupt, quindi, si attua tramite l'hardware e, per completarla, è sufficiente una breve skip chain che si diparte da ciascuna locazione. Questo è spesso il modo con cui si distinguono interrupt esterni, error trap e supervisor call. Il meccanismo di interrupt, come sull'IBM 370, può offrire ulteriori ausili per l'individuazione delle cause allocando in una qualche locazione fissa della memoria le informazioni riguardanti l'interrupt.
Gli interrupt inviati ad un processore centrale normalmente vengono inibiti quando questo passa dal modo non privilegiato al modo privilegiato. Ciò garantisce che i valori dei registri, salvati entrando nel FLIH, non vengano sovrascritti (e quindi perduti) ad opera di un successivo interrupt, verificatosi prima di lasciare il FLIH. Un interrupt che si verifichi mentre è disabilitato il meccanismo degli interrupt viene tenuto in sospeso fino a che il meccanismo stesso non viene riabilitato ritornando al modo non privilegiato. Questa soluzione diventa inattuabile in quelle situazioni in cui qualche unità periferica richiede risposte molto più veloci di altre, come quando, per esempio, non si debbono perdere dati. Per questi casi risulta

conveniente introdurre il concetto di priorità tra le varie cause di interrupt e consentire che una routine per la gestione degli interrupt venga interrotta a sua volta da una richiesta di intervento proveniente da un'unità con priorità più alta.
Alcuni calcolatori (per esempio l'IBM 370) ne consentono l'effettuazione attraverso una disabilitazione selettiva degli interrupt all'interno del FLIH: mentre quest'ultimo esegue l'intervento richiesto da un interrupt, provvede, nel contempo, a disabilitare tutti quelli con priorità minore o uguale. Naturalmente si deve prestare attenzione a memorizzare i registri di programma del processo interrotto in locazioni differenti, in base al livello di priorità dell'interrupt pervenuto. In alternativa, lo stesso hardware che supporta gli interrupt può distinguere i vari livelli di priorità, trasferendo il controllo e salvando i registri in locazioni diverse per ciascun livello. Un interrupt di un certo livello inibisce automaticamente quelli che si trovano ad un livello identico o inferiore. Il CDE System IO, il PDP-I I e l'M6800 sono esempi di elaboratori dotati di questo tipo di meccanismo dì interrupt a priorità, ciascuno con otto livelli di priorità.

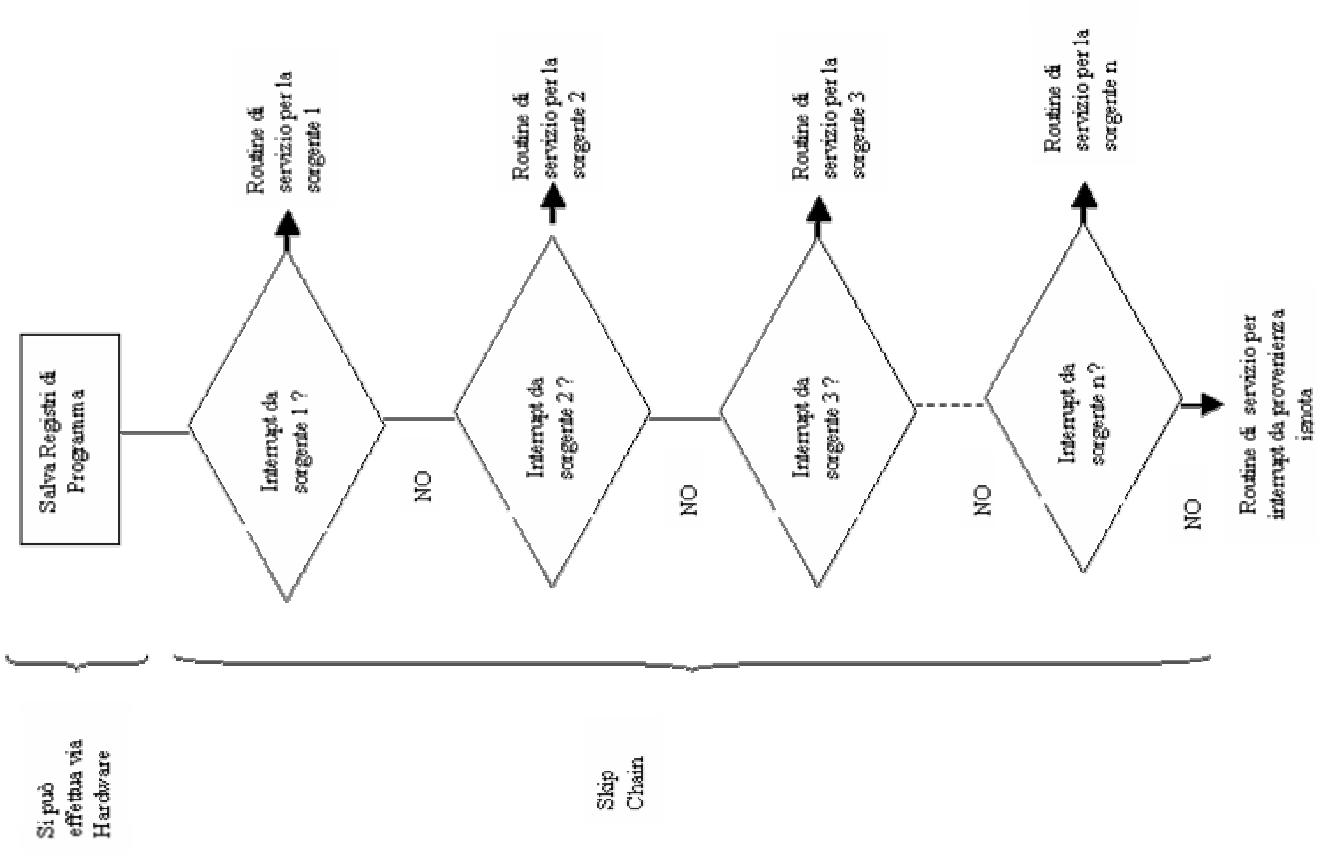


Figura 23 - Skip chain

La seconda funzione del FLIH è quella di avviare la gestione di un interrupt richiamando una routine di gestione adeguata al tipo di interrupt in questione. Forniremo maggiori dettagli sulle routine di servizio per le unità di I/O nel capitolo 6, mentre nel capitolo 11 ci occuperemo della gestione delle error trap. In questa fase richiamiamo l'attenzione solo sul fatto che, poiché le routine di gestione degli interrupt girano nel modo privilegiato con gli interrupt stessi totalmente o parzialmente inibiti, è auspicabile tenerle in esecuzione per il più breve tempo possibile. In generale ogni routine svolgerà qualche intervento minimo (ad esempio, trasferendo un carattere proveniente da un'unità di input ad un buffer) e passerà la responsabilità di ulteriori interventi (per esempio, la risposta relativa al carattere ricevuto) ad un processo eseguibile normalmente nel modo non privilegiato.
È importante notare che il verificarsi di un interrupt, sia esterno che interno, spesso muta lo stato di qualche processo. Ad esempio, un processo che ha richiesto un trasferimento tra periferiche e che si trova in attesa di eventi mentre quest'ultimo è ancora in corso sarà messo nuovamente in stato di pronto per mezzo dell'interrupt che si verifica al completamento del trasferimento stesso. E di nuovo, certe supervisor call, quali un'operazione di wait su un semaforo con valore zero, oppure una richiesta di I/O, metteranno il processo corrente nell'impossibilità di continuare. In tutti questi casi il cambiamento di stato viene effettuato dalla routine per la gestione degli interrupt che modifica le informazioni di stato presenti nel descrittore del processo interessato.
Una conseguenza del cambiamento di stato è che il processo in esecuzione sul processore relativo prima del verificarsi dell'interrupt può non essere quello più idoneo ad essere eseguito successivamente. Può darsi il caso, ad esempio, che l'interrupt metta in stato di pronto un processo con una priorità in un certo senso più alta di quello correntemente in esecuzione. Nel prossimo paragrafo prenderemo in considerazione la questione relativa a quando va commutato un processore centrale tra i processi ed a quale processo va favorito.
E' compito della routine di smistamento (dispatcher), a volte denominata low-level scheduler (programma di schedulazione di basso livello), allocare i processori centrali tra i vari processi esistenti nel sistema. Si entra nella routine di smistamento ogni volta che un processo corrente non riesce a continuare, oppure ogni qual volta vi è motivo di supporre che si potrebbe meglio impiegare un processore in altro modo. Sul verificarsi di questi eventi forniamo qui di seguito maggiori dettagli:

1. in seguito ad un interrupt esterno che cambia lo stato di qualche processo;
2. in seguito ad una supervisor call che mette temporaneamente il processo corrente nell'impossibilità di continuare;

3. in seguito ad un error trap che provoca la sospensione del processo corrente mentre l'errore è oggetto d'intervento.

Tutte queste eventualità corrispondono ad una casistica particolare di interrupt, si tratta cioè di interrupt che cambiano lo stato di qualche processo. Per amore di semplicità, non faremo distinzione tra questi interrupt e quelli, come un'operazione di wait su un semaforo con valore positivo, che non hanno ripercussioni sullo stato di alcun processo; in effetti, diciamo che, in ultima analisi, si entra nella routine di smistamento in seguito a tutti gli interrupt. L'aggravio del sistema operativo conseguente all'attivazione della routine di smistamento nei casi in cui nessun processo ha cambiato stato è più che controbilanciato dai vantaggi ottenuti grazie alla gestione uniforme di tutti gli interrupt.

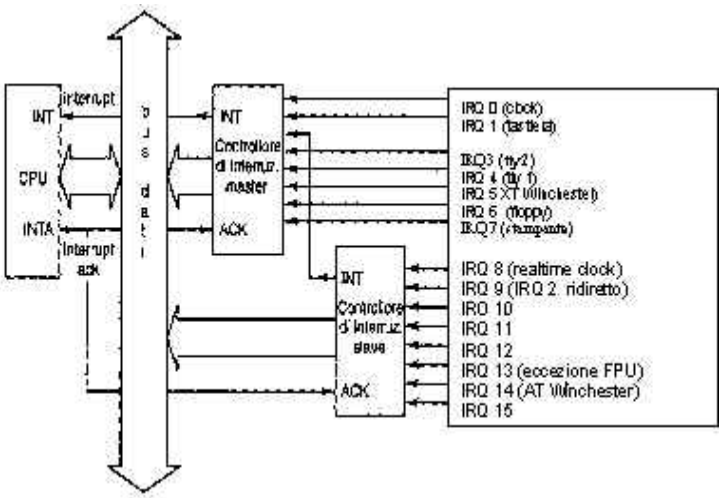
Vediamo con un esempio in maggior dettaglio quello che succede. Supponiamo che il processo utente 3 sia in esecuzione allo scatenarsi di una interruzione da disco. Il program counter ed eventualmente uno o più registri sono inseriti nello stack (corrente) dall'hardware di gestione delle interruzioni. Il computer salta quindi all'indirizzo specificato dal vettore di interruzione del disco. Il compito dell'hardware si esaurisce qui. Da questo punto in poi, quello che accade è compito del software. La procedura di servizio dell'interruzione inizia salvando tutti i registri nella voce della tabella dei processi relativa al processo in esecuzione. L'indice del processo corrente e un puntatore alla sua struttura nella tabella dei processi vengono conservati in variabili globali, in modo da poter essere reperiti velocemente. In seguito, le informazioni depositate nello stack dall'interruzione vengono rimosse e il puntatore allo stack viene modificato in modo da riferire uno stack temporaneo utilizzato dal gestore delle interruzioni. Operazioni come il salvataggio dei registri o l'impostazione del puntatore allo stack non possono essere espresse in linguaggio C e vengono pertanto eseguite da una piccola routine in linguaggio Assembler. Al termine, questa routine chiama una procedura C che svolge il resto del lavoro.

Il passo seguente consiste nella costruzione di un segnale da inviare al processo di gestione del disco, che è bloccato in sua attesa. Il messaggio contiene la segnalazione dell'avvenuta interruzione, per distinguersi dai messaggi che i processi utente inviano per richiedere la lettura di blocchi da disco e altre operazioni simili. Lo stato del processo del disco viene quindi cambiato da bloccato a pronto e lo scheduler di basso livello viene invocato. In molti sistemi processi diversi hanno priorità diverse, in modo da poter dare la precedenza ai gestori dei dispositivi di I/O nei confronti dei processi utente. Se il processo di gestione del disco è ora il processo eseguibile a più alta priorità, sarà scelto dallo scheduler per l'esecuzione. Se il processo appena interrotto era altrettanto o maggiormente importante, sarà a sua volta scelto dallo scheduler per ritornare in esecuzione mentre il processo di gestione del disco dovrà attendere ancora un po'. In entrambi i casi, la procedura C invocata dal codice di interruzione in linguaggio Assembler ritorna, e il codice Assembler carica i registri e la mappa di memoria del processo destinato a diventare il nuovo processo corrente per poi avviarne l'esecuzione.

4.4.2 Gli interrupt su Pentium

I sistemi che utilizzano processori Intel a 32 bit sono normalmente dotati di due chip controllori di questo tipo. Ognuno può controllare otto ingressi, ma uno dei due è uno slave che porta la sua uscita a uno degli ingressi del master, in modo da poter rilevare quindici diversi dispositivi esterni mediante la combinazione dei due controllori, come mostrato in figura.

Figura 24 - Le interruzioni su un PC Intel a 32 bit



Nella figura, i segnali di interruzione arrivano sulle diverse linee IRQn mostrate sulla destra. Il collegamento con il pin INT della CPU indica al processore che vi è stata un'interruzione. Il segnale INTA (interrupt acknowledge o riscontro di interruzione) proveniente dalla CPU fa sì che il controllore responsabile dell'interruzione metta sul bus dati del sistema le informazioni che indicano al processore quale routine di servizio da eseguire. I chip controllori delle interruzioni sono programmati durante l'inizializzazione del sistema. La programmazione determina quale output debba essere inviato al processore in corrispondenza un segnale ricevuto su una qualsiasi linea di input, insieme ad altri parametri operativi del controllore. Il dato immesso sul bus è un numero a 8 bit, usato per indicizzare una tabella che contiene fino a 256 elementi.

Ci sono diversi modi per rispondere a un'interruzione; i campi più importanti degli interrupt riferiscono il segmento di codice eseguibile corrispondente alla routine di servizio e l'indirizzo iniziale all'interno di esso. La CPU esegue il codice riferito dal descrittore selezionato. Il risultato è esattamente identico all'esecuzione dell'istruzione in linguaggio assembler INT <nnn>. La sola differenza è che nel caso di un'interruzione hardware la <nnn> ha origine da registro nel chip controllore delle interruzioni, invece che da un'istruzione del programma in memoria. Il meccanismo di commutazione dei task che viene attivato nei processori Intel a 32 in risposta a un'interruzione è complesso, e la modifica del program counter in modo eseguire un'altra funzione ne è solo una parte. Quando la CPU riceve un'interruzione mentre sta eseguendo un processo, essa prepara un nuovo stack da usarsi durante la gestione dell'interruzione. La locazione di questo stack è determinata da una voce del Segmento di Stato del Processo (task) (TSS o Task State Segment). Esiste una sola struttura questo tipo in tutto il sistema. La CPU immette automaticamente in questo nuovo stack (con altrettanti push) diversi registri chiave, compresi quelli necessari a ripristinare lo stack del processo interrotto e a ripristinare il suo program counter. Quando il codice del gestore dell'interruzione inizia la sua esecuzione, esso usa quest'area nella tabella dei processi come il suo stack, e la maggior parte delle informazioni necessarie per ritornare al processo interrotto saranno già state salvate in esso. Il gestore delle interruzioni salva nello stack il contenuto di altri registri riempiendo lo stackframe, e passa quindi a uno stack fornito dal nucleo per svolgere quanto necessario per servire l'interruzione. La terminazione di una routine di servizio delle interruzioni viene fatta cambiando stack e ritornando dallo stack di nucleo allo stackframe nella tabella dei processi (non necessariamente allo stesso stackframe che era stato creato nell'ultima interruzione estraendo esplicitamente dallo stack i registri addizionali, ed eseguendo una istruzione "iret" (Return from interrupt). Questa istruzione fa sì che la CPU inverta il procedimento ripristinando i registri che erano stati immessi nello stack dall'hardware e ritornando a stack in uso prima dell'interruzione. Nell'istante in cui riceve un'interruzione la CPU disabilita tutte le interruzioni. Questo fa sì che non si possa mai avere un traboccamento dello stackframe all'interno di un elemento nella tabella dei processi. Questo procedimento avviene automaticamente, per quanto esistano anche delle istruzioni assembler per abilitare e disabilitare le interruzioni.

Il gestore delle interruzioni le abilita nuovamente dopo essere passato allo stack di nucleo, collocato al di fuori della tabella dei processi. Il gestore deve ovviamente disabilitare di nuovo tutte le interruzioni prima di ritornare allo stack all'interno della tabella dei processi, ma mentre sta gestendo un'interruzione ne possono essere intercettate ed elaborate altre. La CPU conserva traccia delle interruzioni annidate, e usa un metodo semplice per trasferire il controllo alla routine di servizio delle interruzioni e ritornare da essa quando il gestore dell'interruzione viene interrotto. Quando si riceve una nuova interruzione mentre un gestore (o altro codice di nucleo) è in esecuzione, non si crea un nuovo stack ma, al contrario, la CPU immette nello stack corrente i registri essenziali necessari al ripristino del codice interrotto. La gestione dell'iret richiede un meccanismo di ritorno più semplice anche in un altro caso: quando è in esecuzione il codice del nucleo. Il processore sa come gestire un'istruzione iret esaminando il selettore del segmento codice estratto dallo stack come parte dell'istruzione. I livelli di privilegio menzionati in precedenza controllano la risposta alle interruzioni ricevute quando un processo è in esecuzione e quando si esegue il codice di nucleo (comprese le routine di servizio delle interruzioni). Il meccanismo semplificato viene usato quando il livello di privilegio del codice interrotto è uguale al livello di privilegio del codice da eseguirsi in risposta all'interruzione. Solo quando il codice interrotto ha un livello di privilegio inferiore al codice di servizio dell'interruzione usiamo il meccanismo più elaborato, che prevede l'uso del TSS e un nuovo stack. Il livello di privilegio di ogni segmento codice è memorizzato all'interno del selettore del segmento e, dato che il selettore è uno degli elementi salvati nello stack durante un'interruzione, esso può essere esaminato di ritorno

dall'interruzione per determinare cosa deve fare l'istruzione iret. L'hardware esegue un'altra operazione quando si crea un nuovo stack per servire un'interruzione. L'hardware controlla che il nuovo stack sia grande a sufficienza da contenere almeno la quantità minima di informazioni che vi deve essere immessa. Questa precauzione protegge il codice di nucleo a più elevato privilegio dall'essere accidentalmente (o maliziosamente) distrutto da un processo utente che invochi una chiamata di sistema con uno stack inadeguato. Questi meccanismi sono appositamente costruiti all'interno del processore per essere usati nella realizzazione di sistemi operativi atti a supportare processi paralleli.

4.5 LA ROUTINE DI SMISTAMENTO

Il funzionamento della routine di smistamento è piuttosto semplice:

```
il processo corrente per questo processore è ancora il più idoneo ad essere eseguito?
Se si:
    restituiscigli il controllo al punto indicato dal contatore di programma memorizzato
    nell'hardware dedicato agli interrupt.
Se no:
    Salva l'ambiente volatile del processo corrente nel suo descrittore.
    Recupera l'ambiente volatile del processo più idoneo dal suo descrittore.
    Trasferisci ad esso il controllo, alla posizione indicata dal contatore di programma
    ripristinato.
```

Per determinare quale sia il processo più adatto ad essere eseguito è sufficiente ordinare tutti i processi eseguibili secondo criteri di priorità. L'assegnazione di priorità ai processi non fa parte delle funzioni della routine di smistamento, bensì del gestore dei processi di alto livello, che sarà descritto nel capitolo 8. Per il momento osserviamo che le priorità vengono calcolate in base a fattori quali l'ammontare delle risorse necessarie, il tempo trascorso dall'esecuzione dell'ultimo processo e l'importanza dell'utente originario. Per quanto riguarda la routine di smistamento, le priorità di processo vengono date aprioristicamente.

Nel nostro sistema operativo ipotetico colleghiamo i descrittori di tutti i processi eseguibili in una coda d'attesa ordinata per priorità decrescente, in modo che il processo preferenziale sia il primo in cima. Definiamo questa struttura coda d'attesa al processore. Il ruolo svolto dalla routine di smistamento, quindi, consiste nell'attivare il primo processo presente nella coda d'attesa, sempre che non sia già in esecuzione su qualche altro processore. Potrebbe anche trattarsi dello stesso processo già in esecuzione prima che la routine di smistamento fosse richiamata.

Notiamo per inciso che l'introduzione della coda d'attesa al processore significa che l'intervento svolto dal gestore degli interrupt per rendere eseguibile un processo è di duplice natura: in primo luogo deve cambiare l'entrata di stato nel descrittore di processo e, in secondo luogo, deve agganciare il descrittore alla coda d'attesa al processore, nella posizione indicata dalla sua priorità. Vedremo nel prossimo paragrafo che questa operazione può essere comodamente svolta effettuando un'operazione di signal su un semaforo sul quale il processo interessato abbia eseguito un wait.

Naturalmente è possibile che in un momento particolare la coda d'attesa contenga meno processi di quanti sono i processori, magari perché svariati processi sono in attesa di input od output. Tale situazione, che probabilmente rappresenta il risultato di una cattiva decisione concernente la schedulazione di alto livello, implica che non vi sia lavoro da svolgere per una qualche unità centrale. Anziché consentire che un processore giri a vuoto entro la routine di smistamento, risulta più conveniente inserire un processo aggiuntivo, chiamato null process (processo nullo), avente la priorità più bassa e sempre in stato di pronto. Può darsi che il null process non sia niente di più che un ciclo iterativo a vuoto (idle loop), oppure, al contrario, potrebbe svolgere una qualche funzione pratica, come l'esecuzione di programmi atti a verificare il buon funzionamento del processore. È anche successo, per quanto riguarda il null process, che sia stato utilizzato per la risoluzione dei problemi relativi alle mosse finali del gioco degli scacchi od il calcolo dello sviluppo decimale di pi-greco. Il null process si trova sempre alla fine della coda d'attesa al processore.

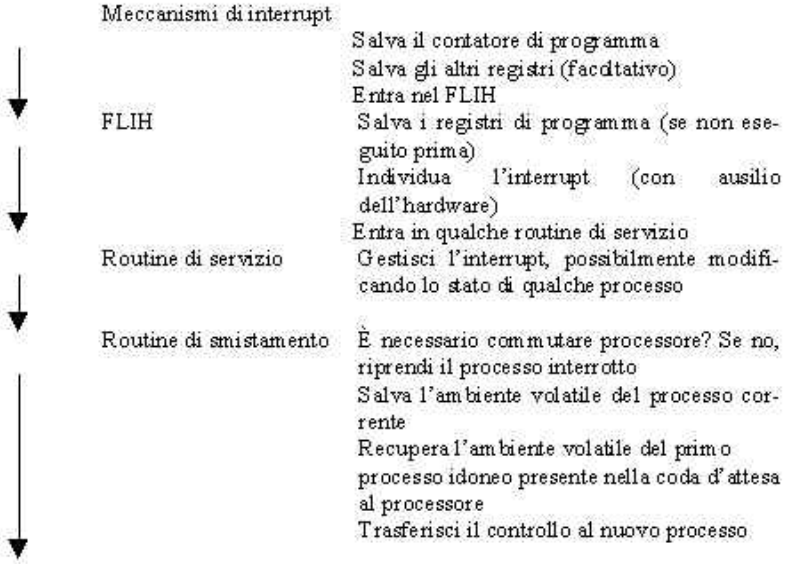
A questo punto si può riassumere come segue il funzionamento della routine di smistamento:

```
Il processo corrente su questo processore è ancora il primo processo non in esecuzione presente
nella coda d'attesa al processore?
Se si:
    riattivalo
Se no:
    Salva l'ambiente volatile del processo corrente.
    Recupera l'ambiente volatile del primo processo non in esecuzione presente nella coda d'attesa
    al processore.
    Riattiva l'esecuzione di questo processo.
```

Prima di mettere da parte la routine di smistamento bisogna sottolineare che abbiamo scelto una struttura molto semplice per quanto riguarda i processi in stato di pronto. Molti sistemi operativi li suddividono in parecchie classi, differenziate attraverso criteri quali il fabbisogno di risorse, il tempo massimo d'attesa ammissibile, o l'entità del tempo di CPU da concedere prima dell'interdizione della risorsa, Il (programma) monitor del DEC System-10, per esempio, possiede tre code d'attesa al processore, ciascuna delle quali è dedicata a quei processi cui sono concessi, rispettivamente, 2 secondi, 0,25 secondi e 0,02 secondi prima dell'interdizione della risorsa. In ogni coda vige il principio secondo cui il primo processo ad arrivare è il primo anche ad essere soddisfatto, e la priorità più alta è assegnata alle code d'attesa aventi il minor tempo di interdizione. Inizialmente, un qualsiasi processo viene sistemato nella coda da 0,02 secondi; se rimane in esecuzione per l'intero arco di tempo concessogli, viene retrocesso alla coda d'attesa da 0,25 secondi e, analogamente, se anche qui rimane in esecuzione per l'intero periodo concessogli (questa volta maggiore) viene declassato alla coda d'attesa da 2 secondi. Si ottiene così la garanzia che i processi connessi a terminali in linea (caratterizzati da minori esigenze di utilizzo della CPU) ricevano rapidamente attenzione, mentre i job più vincolati al processo-re ricevono un'attenzione di maggiore durata, anche se con minor frequenza. Approfondiremo questo punto nel capitolo 8, dove prenderemo in esame l'attività di schedulazione,

Concludiamo questo paragrafo riassumendo il rapporto esistente tra il gestore degli interrupt di primo livello e la routine di smistamento, con l'ausilio dello schema riportato in seguito.

Figura 25 - Rapporto tra le routine del nucleo



4.6 L'IMPLEMENTAZIONE DI wait E signal

La parte finale della costruzione del nucleo consiste nell'implementazione di una qualche forma di meccanismo per la comunicazione tra i processi. Come indicato nel capitolo 3, a tale scopo utilizzeremo le operazioni di wait e signal come primitive, una scelta che deriva dall'assai diffusa conoscenza dei semafori e dalla facilità con cui si possono implementare. Nella letteratura specializzata si possono trovare, tuttavia, altri meccanismi per la comunicazione tra i processi (Andrews e Schneider, 1938). Alcuni di essi, come ad esempio i programmi monitor (si veda il paragrafo 3.4), possono essere implementati a loro volta tramite i semafori. Wait e signal vengono incluse nel nucleo per i seguenti motivi:

- devono essere a disposizione di tutti i processi e, quindi, devono essere implementate ad un livello basso;
- l'operazione di wait potrebbe dar luogo all'arresto di un processo, provocando-ne l'entrata nella routine di smistamento per provvedere alla riassegnazione del processore relativo. Pertanto, l'operazione di wait deve poter accedere alla routine di smistamento;
- un metodo pratico per far sì che una routine di gestione degli interrupt risvegli (metta in stato di pronto) un processo, consiste nell'esecuzione di una signal su un semaforo sul quale un processo abbia eseguito una wait.

Pertanto, all'operazione di signal devono poter accedere le routine di gestione degli interrupt.
Le operazioni che dobbiamo attuare (si veda il capitolo 3) sono le seguenti:

```
wait(s) : when s >0 do decremento di s
signal(s) : incremento di s
```

dove s è un semaforo generico. Per l'attuazione pratica procederemo nel modo seguente.

4.6.1 Raggruppamento dei processi

L'operazione di wait comporta il raggruppamento dei processi quando un semaforo ha valore 0 ed il rilascio degli stessi allorché un'operazione di signal ne aumenta il valore ad 1. Il modo naturale di attuare questo meccanismo consiste nell'associare a ciascun semaforo una coda d'attesa al semaforo. Quando un processo esegue "senza successo" un'operazione di wait (opera, cioè, su un semaforo con valore zero) viene aggiunto alla coda d'attesa al semaforo e messo in attesa d'eventi. Per contro, quando viene eseguita un'operazione di signal su un semaforo, si può togliere qualche processo dalla coda d'attesa relativa (a meno che non sia vuota) e renderlo nuovamente eseguibile (metterlo, cioè, in attesa di esecuzione). Il semaforo, pertanto, deve essere attuato impiegando due componenti: un intero ed un puntatore alla coda d'attesa (che può essere nullo).
A questo punto, la nostra implementazione risulta la seguente:

```
wait(s) :   if s > 0 then s:= s - 1
            else aggiungi il processo alla coda d'attesa al semaforo e mettilo in attesa d'eventi;
signal(s) : if coda d'attesa è vuota then s: =s + 1
            else togli qualche processo dalla coda d'attesa al semaforo e mettilo in attesa
            d'esecuzione.
```

Si noti che se un processo va rilasciato non è necessario incrementare il semaforo all'interno di una signal, poiché lo stesso processo dovrebbe poi subito effettuare un nuovo decremento dello stesso semaforo nel completare la sua operazione di wait.

4.6.2 Inserimento e prelievo nella coda d'attesa

Non abbiamo ancora parlato di quei processi tanto fortunati da venir rimossi dalla coda d'attesa al semaforo in seguito ad un'operazione di signal. Non abbiamo nemmeno stabilito se un processo, aggiunto alla coda d'attesa in seguito ad un'operazione di wait non riuscita, vada inserito all'inizio, alla fine oppure in una qualche posizione intermedia. In altri termini, non abbiamo ancora precisato l'organizzazione della coda d'attesa.
Per la maggior parte dei semafori risulta adeguata una coda che preveda semplicemente il rilascio dei processi in ordine cronologico d'entrata (secondo il cosiddetto metodo first-in first-out), poiché assicura che tutti i processi raggruppati, alla fin fine, vengono liberati. In alcuni casi può essere preferibile dare alla coda d'attesa un ordine basato su altri criteri, quale ad esempio la priorità, analogamente a quanto avviene per la coda d'attesa al processore. Quest'ultimo metodo

organizzativo dà la garanzia che i processi con alta priorità di utilizzo del processore non rimangano inattivi a lungo nella coda d'attesa al semaforo. Il punto importante è che semafori diversi possono richiedere diverse organizzazioni per quanto concerne le relative code d'attesa: diviene, pertanto, necessario inserire nell'attuazione pratica dei semafori un'altra componente che indichi il tipo di organizzazione valido per la coda in questione. Si potrebbe trattare semplicemente di una descrizione codificata del metodo organizzativo applicato oppure, nei casi più complessi, di un puntatore relativo ad un piccolo (segmento di) programma atto all'esecuzione delle operazioni di inserimento e di prelievo dalla coda d'attesa. La struttura dei semafori secondo il nostro schema di realizzazione pratica è mostrato in figura.

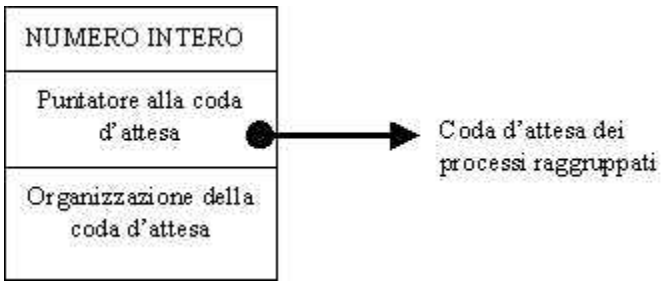


Figura 26 - Struttura di un semaforo

4.6.3 Allocazione dei processi

Ambedue le operazioni di wait e signal possono modificare lo stato di un processo: la prima ponendolo in attesa di eventi e la seconda agendo in modo contrario. Pertanto, si deve prevedere un'uscita (exit) diretta alla routine di smistamento per determinare quale sia il processo successivo da mettere in esecuzione. Se non vi sono cambiamenti in nessuno degli stati dei processi (o una wait in un semaforo con valore positivo, o una signal in un semaforo con relativa coda d'attesa vuota) la routine di smistamento riprende il processo corrente, in quanto esso risulta ancora essere il primo processo non in corso d'esecuzione presente nella coda d'attesa al processore. Si potrebbe anche scegliere, in questo caso particolare, di restituire il controllo da wait o signal direttamente al processo corrente, anziché passare attraverso la routine di smistamento; ne conseguirebbe un guadagno dal punto di vista dell'efficienza, anche se a scapito di un leggero aumento della complessità della codifica di wait e signal. Nella presente trattazione preferiremo la semplicità all'efficienza, pur riconoscendo che anche l'opinione contraria può risultare ugualmente valida.

4.6.4 Indivisibilità

Come è già stato sottolineato nel capitolo 3, sia wait che signal sono operazioni indivisibili, nel senso che in ogni momento dato solo ad un processo è permesso eseguirle. Entrambe, quindi, devono essere attuate come procedure che iniziano con un qualche tipo di operazione di lock (blocco) e terminano con una di unlock (sblocco). In una configurazione ad un singolo processore è facile attuare l'operazione di lock disabilitando il meccanismo degli interrupt: si garantisce così ad ogni processo il mantenimento del controllo dell'unità centrale durante l'esecuzione di wait e signal, non potendo essere interrotto in alcun modo. D'altra parte, l'operazione di unlock viene eseguita semplicemente riabilitando gli interrupt. In una macchina con più processori centrali questa procedura non risulta adeguata, in quanto è possibile per due processi entrare contemporaneamente in wait o in signal, essendo eseguiti su processori distinti. In tal caso, è necessario qualche altro meccanismo, come, ad esempio, un'istruzione di "test and set". Si tratta di un'istruzione che verifica e modifica il contenuto di una locazione di memoria con un'unica operazione. Durante l'esecuzione di questa istruzione vengono inibiti i tentativi compiuti da altri processi di accedere alla locazione interessata.
Il concetto base è quello di usare una posizione specifica come flag (analogamente alla variabile porta vista nel paragrafo 3.3), che indichi se è concesso o meno entrare nella procedura wait e signal. Poniamo che ciò sia permesso solo se il flag è diverso da zero; in caso contrario non è consentito l'ingresso. L'operazione di lock consiste nell'esecuzione di un'istruzione di "test and set" sul flag, che ne determina il valore e contemporaneamente lo azzera. Se il valore del flag è diverso da zero, il processo prosegue, altrimenti inizia un'iterazione dell'istruzione di "test and set" fino a che il processo, in quel momento all'interno della procedura wait o signal, non sblocca il flag settandolo su un valore diverso da zero. Va tenuto presente che si evitano difficoltà analoghe a quelle verificatesi a proposito della variabile porta nel paragrafo 3.3, implementando l'intera istruzione di "test and set" come operazione singola ed indivisibile. Questo meccanismo di blocco e sblocco è stato adottato su molti calcolatori, per esempio quelli della serie IBM 370.
Un'alternativa all'impiego dell'istruzione di "test and set" è rappresentata da un'istruzione che serve per l'interscambio del contenuto di due locazioni di memoria, soluzione adottata nella serie Burroughs 6000 e nel Plessey 250. L'operazione di lock ha inizio con l'interscambio dei valori relativi a un flag ed a una posizione precedentemente azzerata. Viene quindi esaminato il valore di questa seconda locazione per vedere se è permesso accedervi, mentre qualsiasi altro processo che tenti di entrarvi troverà lo zero lasciato dall'interscambio. Se si verifica quest'ultima situazione, il processo ripete semplicemente l'operazione di lock, fino a che il flag non viene settato ad un valore diverso da zero ad opera di un qualche altro processo che esegue un'operazione di unlock.
Si noti che entrambi questi meccanismi comportano qualche forma di busy waiting, in base a cui un processo che non riesce a superare l'operazione di lock impegna il suo processore in un ciclo iterativo, tentando ripetutamente l'operazione fino a che il flag non viene modificato. Fintantoché le procedure wait e signal sono semplici, la durata del busy waiting dovrebbe essere piuttosto breve.

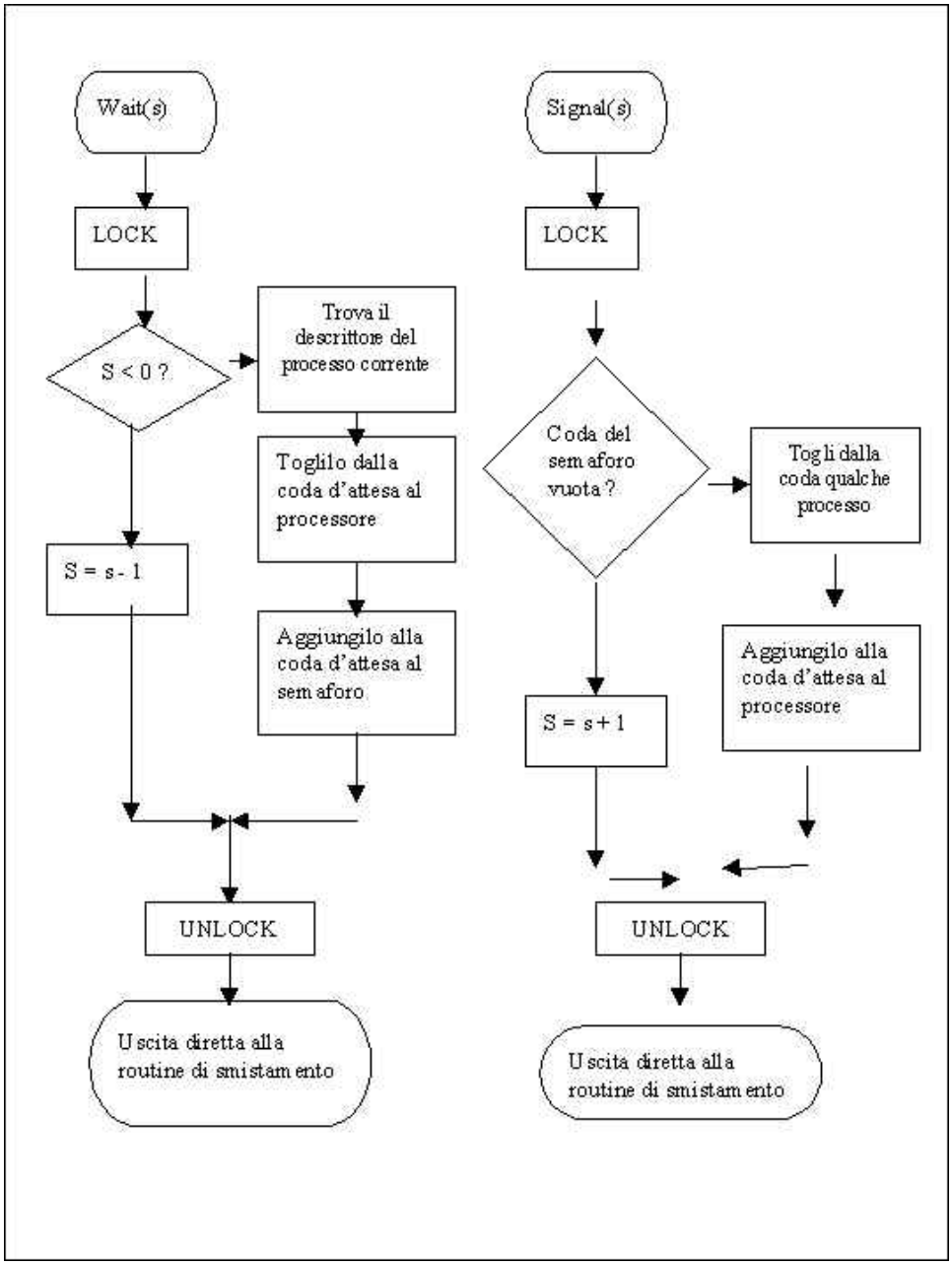


Figura 27 - Implementazione di wait e signal

Vale la pena sottolineare che le operazioni di lock e unlock non possono essere impiegate in sostituzione di quelle di wait e signal. Il busy waiting o l'inibizione degli interrupt, metodi utilizzati per mettere in pratica l'operazione di lock, non sarebbero accettabili vista l'entità dei tempi di ritardo cui possono essere soggetti i processi in seguito ad un'operazione di wait. La tavola seguente mostra i diversi ambiti concettuali entro i quali si svolgono questi due tipi di operazioni.

	wait e signal	**lock e unlock**
Scopo	Sincronizzazione generale dei processi	Mutua esclusione dei processi dalle procedure wait e signal
Livello di implementazione	Software	Hardware
Meccanismo di ritardo	Inserimento in coda d'attesa	Busy waiting/inibizione degli interrupt
Tempo caratteristico di ritardo	Svariati secondi	Svariati microsecondi

Dalla precedente discussione si deduce che è possibile implementare le operazioni di wait e signal sotto forma di due procedure correlate, così come indicato in figura 27, le quali saranno chiamate tramite supervisor call che, a loro volta, fanno parte del repertorio di istruzioni di tutti i processi. A questo punto abbiamo completato il nucleo del sistema che,

quindi, risulta costituito dal gestore degli interrupt di primo livello, dalla routine di smistamento e dalle procedure wait e signal, tutti eseguibili in modo privilegiato.

5 GESTIONE DELLA MEMORIA

La gestione della memoria rappresenta lo strato successivo della nostra “cipolla”. La poniamo vicina al nucleo, perché, se un processo non dispone di un certo spazio di memoria, non è in grado di fare molto. È certo che non riuscirebbe ad andare in esecuzione, dal momento che non c'è spazio per il programma relativo, e che non riuscirebbe nemmeno a svolgere operazioni di input/output, non essendovi spazio per i buffer.
Idealmente, il desiderio di ogni programmatore è una memoria infinitamente grande, veloce e che sia non volatile, cioè non perda il suo contenuto quando l'alimentazione elettrica scompare. Visto che ci siamo, tanto vale chiedere che sia anche poco costosa. Sfortunatamente, la tecnologia non ci mette a disposizione memorie simili. Di conseguenza, la maggior parte dei computer contengono una gerarchia di memoria comprendente una piccola quantità di memoria tampone (cache), molto veloce, costosa e volatile, alcuni megabyte di memoria principale (RAM) di media velocità, medio costo e volatile, e centinaia o migliaia di megabyte di memoria disco, lenta, economica e non volatile. Il sistema operativo ha il compito di coordinare l'uso di queste memorie.
La parte del sistema operativo che gestisce la gerarchia di memoria viene chiamata il gestore della memoria. Il suo compito è sapere, istante per istante, quali parti di memoria siano in uso e quali no, allocare la memoria ai processi quando ne hanno bisogno e deallocarla quando hanno terminato, e gestire lo swapping tra la memoria principale e il disco quando la prima è troppo piccola per contenere tutti i processi.
In questo capitolo tratteremo degli obiettivi da porsi in relazione alla gestione della memoria ed introdurremo il concetto di memoria virtuale. Daremo una descrizione dei modi in cui si può realizzare quest'ultima e, successivamente, prenderemo in esame vari metodi miranti a regolamentarne l'implementazione. Infine, vedremo come sia possibile inserire la gestione della memoria nel nostro sistema operativo ipotetico

5.1 OBIETTIVI

Gli obiettivi da porsi in relazione alla gestione della memoria sono di cinque tipi diversi.

5.1.1 Rilocazione

In un elaboratore operante in multiprogrammazione la memoria disponibile, in linea generale, viene suddivisa in ogni dato istante tra i vari processi e per il singolo programmatore non è possibile sapere in anticipo quali saranno gli altri programmi residenti in memoria nel momento in cui viene messo in esecuzione il programma da lui redatto. Questo vuol dire che non gli è dato sapere con precisione, al momento di scrivere il programma, quale sarà la parte di memoria destinata ad ospitarlo e, quindi, non può scriverlo in termini di indirizzi assoluti di memoria. Se lo spazio assegnato al programma da lui preparato rimanesse fisso per tutta la durata dell'esecuzione, è ovvio che sarebbe possibile trasformare indirizzi simbolici o relativi in indirizzi assoluti all'atto del caricamento in macchina; tutto ciò, però, accade raramente. Con l'avvicinarsi dei processi al loro completamento, lo spazio utilizzato si libera, diventando disponibile per altri processi, ed al fine di ottimizzarne il successivo utilizzo potrebbe risultare necessario cambiare posizione ai processi esistenti in memoria. In particolare, potrebbe tornare comodo spostare i processi in modo tale da unire piccole zone non contigue di memoria libera, formando un'unica area disponibile, di estensione ed utilità maggiori. La regione di memoria destinata ad un processo può così variare durante la vita dello stesso ed il sistema deve occuparsi della trasformazione degli indirizzi impiegati dal programmatore in quegli indirizzi che realmente si riferiscono all'ubicazione fisica del processo.

5.1.2 Protezione

Allorché svariati processi si trovano a condividere lo stesso spazio di memoria diventa naturalmente essenziale, ai fini della loro integrità, che a nessuno di essi sia concesso modificare i dati contenuti nelle locazioni di memoria già destinate agli altri. I controlli relativi al tempo di compilazione, effettuati sugli indirizzi di programma, non rappresentano una protezione adeguata, in quanto la maggioranza dei linguaggi consente il calcolo dinamico degli indirizzi in fase esecutiva, ad esempio, calcolando gli indici o i puntatori di un array ed inserendoli nelle strutture dati. Tutti i riferimenti di memoria generati da un processo, pertanto, devono essere controllati durante la fase esecutiva, in modo da garantire che si riferiscano solamente allo spazio di memoria allocato a quel processo specifico. A rigor di termini è necessario controllare solo gli accessi per scrittura, ma se è richiesta la riservatezza delle informazioni contenute devono essere controllati anche quelli per lettura.

5.1.3 Condivisione

Nonostante la protezione sia necessaria, vi sono circostanze in cui a più processi dovrebbe essere permesso accedere alla stessa parte di memoria. Ad esempio, se più processi stanno eseguendo il medesimo programma, risulta vantaggioso consentire a ciascuno di essi di accedere alla stessa copia di quel programma, anziché prevederne copie distinte. In modo analogo, a volte può essere necessario che i processi condividano una struttura dati ed abbiano così accesso comune alla zona di memoria che la contiene. Il sistema di gestione della memoria, pertanto, deve consentire l'accesso controllato ad aree di memoria in condivisione, senza compromettere la funzione essenziale della protezione.

5.1.4 Organizzazione logica

L'elaboratore tradizionale possiede uno spazio di indirizzamento monodimensionale, ovvero lineare, in quanto gli indirizzi sono numerati sequenzialmente da zero fino al limite superiore della memoria. Pur rispecchiando da vicino l'hardware della macchina, questa organizzazione non riflette realmente le modalità secondo le quali di solito sono scritti i programmi. La maggior parte di essi è strutturata secondo qualche principio (in moduli o procedure) e fa riferimento a zone distinte contenenti dati, modificabili o meno. A titolo d'esempio un compilatore può venir scritto con moduli separati relativamente all'analisi lessicale, all'analisi sintattica ed alla generazione del codice e tra le sue aree di dati possono essere comprese una tavola di word (parole) riservate (non modificabile) ed una tavola dei simboli (la quale viene modificata mano a mano che s'incontrano nuovi identificatori durante la compilazione). Se le suddivisioni logiche esistenti nel programma e nei dati rispecchiano la segmentazione corrispondente allo spazio di indirizzamento, se ne possono trarre parecchi vantaggi. In primo luogo, vi è la possibilità di codificare i segmenti indipendentemente gli uni dagli altri ed anche di prevedere che tutti i rimandi da un segmento all'altro vengano compilati dal sistema nella fase di esecuzione. In

secondo luogo, senza appesantire ulteriormente il sistema, è possibile garantire livelli diversi di protezione a segmenti distinti (ad esempio, solo lettura, solo esecuzione). Come terza ed ultima soluzione è possibile introdurre meccanismi tramite i quali i segmenti siano in grado di essere condivisi tra processi.

5.1.5 Organizzazione fisica

Da un punto di vista storico, il desiderio generalizzato di disporre di grosse quantità di spazio per la memorizzazione dei dati e gli alti costi della memoria di tipo veloce hanno portato, quasi universalmente, all'adozione di sistemi di memorizzazione a due livelli. La caratteristica soluzione di compromesso tra la velocità richiesta e gli alti costi relativi consiste nell'integrazione di una quantità relativamente piccola (qualche miliardo di byte al massimo) di memoria centrale ad accesso diretto tramite una memoria secondaria, o memoria di massa, di ben più vaste dimensioni (finanche svariate centinaia di miliardi di byte). La memoria centrale viene realizzata con l'impiego della tecnologia dei semiconduttori e presenta un tempo d'accesso dell'ordine dei 100 nanosecondi; la memoria di massa, invece, si basa solitamente su dischi magnetici, con un tempo d'accesso che raggiunge anche i 100 millisecondi.
Dal momento che nei sistemi generalmente è possibile accedere direttamente solo alle informazioni contenute in memoria centrale risulta ovvio che l'organizzazione del flusso di informazioni tra le memorie principali e quelle di massa riveste un carattere di importanza primaria. E' possibile, naturalmente, far gravare l'onere di questa organizzazione sul singolo programmatore, affidandogli la responsabilità di effettuare gli spostamenti di segmenti, programmi o dati dalla memoria centrale o verso di essa, secondo le necessità. In realtà, ciò è quanto veniva messo in pratica ai primordi dell'era del calcolo automatico, tanto che la tecnica della overlay programming (la scrittura di segmenti di programma che si sovrappongono in memoria) divenne un'arte vera e propria. Ci sono, tuttavia, due validi motivi che vanno a sfavore di tale impostazione. Il primo consiste nel fatto che il programmatore non vuole impegnare gran parte delle proprie capacità nello sforzo di scrivere sovrapposizioni complesse; è troppo occupato a risolvere i propri problemi per sprecare energie su questioni marginali. Compilatori realizzati in modo razionale possono rivelarsi utili in questo senso, generando automaticamente il codice di sovrapposizione nei punti più opportuni del programma, anche se in molti casi il compilatore non ha informazioni sufficienti per sapere quando saranno richieste le sovrapposizioni. Il secondo motivo consiste nel fatto che, a causa della rilocazione dinamica, al programmatore non è dato sapere, al momento di scrivere il programma, quanto spazio sarà disponibile e dove sarà ubicato nell'ambito della memoria. La programmazione overlay diventa così inattuabile
Risulta chiaro da queste tesi che il compito di effettuare spostamenti di informazioni tra i due livelli di memoria dovrebbe ricadere sul sistema stesso. Come sia possibile mettere in pratica tale funzione, lo vedremo in seguito in questo capitolo.

5.2 LA MEMORIA VIRTUALE

Gli obiettivi elencati in precedenza si possono raggiungere attraverso l'accorgimento concettualmente semplice (e gradevolmente elegante) consistente nell'impiego di un meccanismo per la conversione degli indirizzi, o mappa di indirizzamento, per trasformare gli indirizzi utilizzati dal programmatore nelle corrispondenti locazioni di memoria fisica realmente allocate per il programma. La conservazione della mappa di indirizzamento fa parte delle funzioni del sistema e nel prossimo paragrafo ne tratteremo le possibili implementazioni. Il passo fondamentale, in questa fase, è la distinzione tra gli indirizzi di programma (quelli utilizzati dal programmatore) e le locazioni di memoria fisica nelle quali sono mappati. La gamma degli indirizzi di programma va sotto il nome di spazio di indirizzamento (o spazio di denominazione); la gamma delle locazioni di memoria dell'elaboratore è lo spazio di memoria. Se si denota lo spazio di indirizzamento con N e lo spazio di memoria con M, la mappa di indirizzamento può essere così definita: f->N := M
Negli elaboratori attuali lo spazio di memoria è lineare (cioè le locazioni di memoria sono numerate sequenzialmente a partire da zero) e la sua estensione equivale alla quantità di memoria centrale inclusa nella configurazione. Vedremo che lo spazio di indirizzamento non deve tuttavia essere necessariamente lineare e che, secondo l'implementazione specifica della mappa di indirizzamento, la sua estensione può essere inferiore, uguale o maggiore di quella dello spazio di memoria.
Un modo alternativo per guardare alla mappa di indirizzamento consiste nel considerarla come un metodo per consentire al programmatore di servirsi di una gamma di indirizzi di programma che può essere anche molto diversa da quella delle locazioni disponibili in memoria. Il programmatore, pertanto, “vede” (e ne effettua il programma relativo) una memoria virtuale con caratteristiche differenti da quelle della memoria reale. La mappa di indirizzamento è concepita in modo da produrre una memoria virtuale conveniente per il programmatore, raggiungendo così tutti gli obiettivi (o solamente alcuni) esposti nel paragrafo precedente. La messa a disposizione della memoria virtuale rappresenta un buon esempio della trasformazione (trattata nel capitolo 1) di un elaboratore puro e semplice in una macchina virtuale con caratteristiche di maggior praticità.
Nel prossimo paragrafo descriveremo varie memorie virtuali con le mappe di indirizzamento relative.
Lo schema per la gestione della memoria più semplice che ci sia consiste nell'eseguire un programma alla volta, condividendo la memoria tra il programma e il sistema operativo. Tre variazioni sul tema sono mostrate in figura 28. Il sistema operativo può essere in fondo alla memoria RAM (Memoria ad Accesso Casuale), come mostrato in figura 28(a), oppure può essere nella ROM (Memoria di Sola Lettura) posta in cima alla memoria, come mostrato in figura 28(b) o ancora i device driver possono essere in cima alla memoria, all'interno di una ROM e il resto del sistema in basso, all'interno della RAM, come mostrato in figura 28(c).

Figura 28 - Tre modi semplici di organizzare la memoria con un sistema operativo e un processo utente

a	Sistema Operativo in RAM	Programma Utente	
b	Programma Utente	Sistema Operativo in ROM	
c	Sistema Operativo in RAM	Programma Utente	Device Driver in ROM

L'ultimo modello viene usato, ad esempio, nei piccoli sistemi MS-DOS. Sui PC IBM la porzione del sistema contenuta nella ROM viene chiamata BIOS (Basic Input Output System — Sistema di Ingresso Uscita di Base).
Quando il sistema è organizzato in questo modo un solo processo alla volta può essere in esecuzione. Non appena l'utente batte un comando, il sistema operativo carica in memoria il programma richiesto, leggendolo dal disco, e lo esegue. Quando il processo termina, il sistema operativo stampa il carattere di prompt e attende un nuovo comando. Quando riceve il comando, carica in memoria un nuovo programma, sovrascrivendo il primo.

Nei sistemi timesharing il fatto di avere più processi in memoria nello esso istante si traduce in una migliore utilizzazione della CPU. Anche sui personal computer, del resto, è spesso utile poter eseguire due o più processi contemporaneamente.
Il modo più semplice per realizzare la multiprogrammazione consiste nel dividere la memoria in n partizioni (eventualmente uguali). Questo partizionamento può, ad esempio essere effettuato manualmente all'avviamento del sistema.
Quando arriva un job può essere inserito nella coda di ingresso per la partizione più piccola in grado di contenerlo. Poiché in questo schema le partizioni sono fisse, qualsiasi spazio che appartenga a una partizione e non sia utilizzato da un job viene perso. In figura 29(a) vediamo come appare questo sistema a partizioni fisse e code di ingresso separate.
Lo svantaggio dell'ordinare i job in arrivo in code separate diventa evidente quando la coda per una partizione grande è vuota mentre la coda per una partizione piccola è piena, come accade per le partizioni 1 e 3 di figura 29(a). Uno schema alternativo prevede la gestione di una sola coda, come in figura 29(b). Ogni volta che si libera una partizione, il job più vicino alla testa della coda e di dimensioni inferiori alla partizione appena liberata potrebbe essere caricato nella partizione vuota ed eseguito.

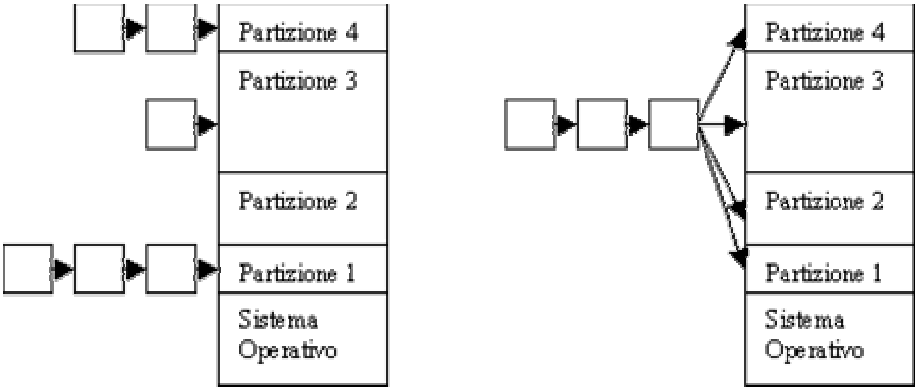


Figura 29 - (a) Partizioni fisse in memoria con code di ingresso separate per ciascuna partizione. (b) Partizioni fisse in memoria con una sola coda di ingresso

5.3 L'IMPLEMENTAZIONE DELLA MEMORIA VIRTUALE

5.3.1 Registri base e registri limite

I primi due obiettivi esposti nel paragrafo 5.1, più precisamente la rilocazione e la protezione, possono essere ottenuti tramite una mappa di indirizzamento piuttosto semplice, come precisato di seguito.
Quando si carica un processo in memoria, l'indirizzo della locazione più bassa viene posto in un registro base (o registro di riferimento) e tutti gli indirizzi di programma vengono interpretati in relazione a questo indirizzo base. La mappa di indirizzamento consiste perciò semplicemente nell'aggiunta dell'indirizzo di programma all'indirizzo base, generando così la corrispondente locazione di memoria; più precisamente `f(a) := B+a`
dove a è l'indirizzo di programma e B è l'indirizzo base del processo.
La rilocazione avviene semplicemente spostando il processo e ripristinando l'indirizzo base sul valore appropriato. La protezione dello spazio di memoria all'interno di vari processi si può ottenere inserendo un secondo registro (il registro limite) contenente l'indirizzo della locazione più alta cui può accedere un determinato processo. La mappa di indirizzamento (vedi fig. 30) eseguita dall'hardware di indirizzamento della memoria, pertanto, si svolge secondo lo schema che segue:

a. se a < 0, allora violazione di memoria;
b. a':= B + a;
c. se a' > limite, allora violazione di memoria;
d. a' è la locazione richiesta.

Si noti che lo spazio di indirizzamento mappato tramite questo schema è lineare e che la sua estensione, in quanto data dalla differenza tra il registro base e quello limite, è necessariamente minore o uguale a quella dello spazio di memoria. Gli obiettivi (3) e (4) menzionati nel paragrafo 5.1, però, non sono stati raggiunti.
Affinché l'operazione di mapping degli indirizzi non sia eccessivamente dispendiosa in termini di tempo, il registro base ed il registro limite devono essere implementati con un hardware di tipo veloce. Si può ridurre il costo relativo ai registri e sveltire anche il mapping, togliendo i bit meno significativi, il che implica che l'estensione dello spazio degli indirizzi deve essere un multiplo di 2 (dove n è il numero di bit tolti).
Una piccola variante dello schema fondato su registri base e limite consiste nell'impiego del registro limite per contenere la lunghezza dello spazio di memoria, anziché il suo limite superiore. In tal caso, la mappa di indirizzamento diventa la seguente:

a. se a<0 oppure a>lunghezza, allora violazione di memoria;
b. a':= B + a;
c. a' è la locazione richiesta.

Il vantaggio che se ne trae è che il controllo del limite non dipende più dal risultato dell'addizione e così sia i controlli che le addizioni possono essere eseguiti in parallelo. L'addizione viene interrotta (aborted) se si verifica una violazione di memoria.

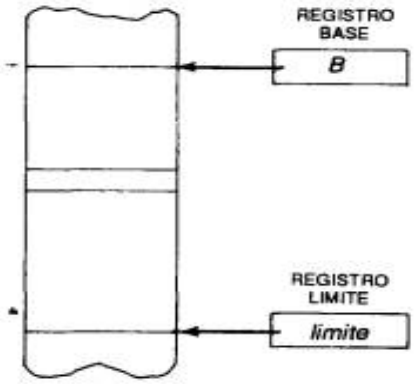


Figura 30 - Mappa di indirizzamento per registro base e registro limite

Per motivi economici, non è praticamente fattibile l'inserimento di una coppia di registri base e registri limite per ogni processo eventualmente presente in memoria. In sostituzione, per ogni processore viene fornita una singola coppia di registri che viene caricata con l'indirizzo base e l'indirizzo limite relativo al processo attivo correntemente. Questi valori fanno parte dell'ambiente volatile del processo e vengono memorizzati al momento della sospensione dello stesso.
Figura 30.Mappa di indirizzamento per registro base e registro limite
La condivisione di programmi rientranti può essere realizzata fornendo due coppie di registri, anziché una sola: una coppia serve a delimitare lo spazio occupato in memoria dal codice di rientro ed i valori contenuti in essa sono comuni a tutti i processi che utilizzano il codice, mentre l'altra coppia serve a delimitare le aree dati relative al codice e contiene valori diversi per ogni processo interessato. Il primo DEC System-10 è un esempio di macchina che fa uso di questa tecnica.

5.3.2. Swapping

Una tecnica intermedia tra quella esposta precedentemente e le tecniche di memoria virtuale vera e propria, consiste nel caricare in memoria ciascun processo per intero, eseguirlo per un po', scaricalo sul disco alternando l'uso della memoria con un altro processo e riprenderlo successivamente.
Questa tecnica nota con il nome di swapping, per essere applicata richiede che il gestore della memoria abbia a disposizione una bitmap o una lista di tutte le porzioni di spazio di memoria allocabili (possono essere un piccolo numero di words o segmenti più grandi e di dimensioni variabili).
La gestione della bitmap o della lista dei segmenti viene fatta contrassegnando ciascuna di queste porzioni di memoria con un bit di assegnazione. Poiché i processi possono richiedere nel corso del tempo di allocare nuovo spazio, normalmente a ciascun processo viene allocato in partenza uno spazio maggiore di quello richiesto (per l'area dati e per lo stack), per consentire al sistema operativo di rispondere senza particolari interventi alla richiesta di nuovo spazio.
Lo stato della memoria si presenterà ad un certo istante come illustrato in figura. (la zona dati cresce verso l'alto, lo stack cresce verso il basso.

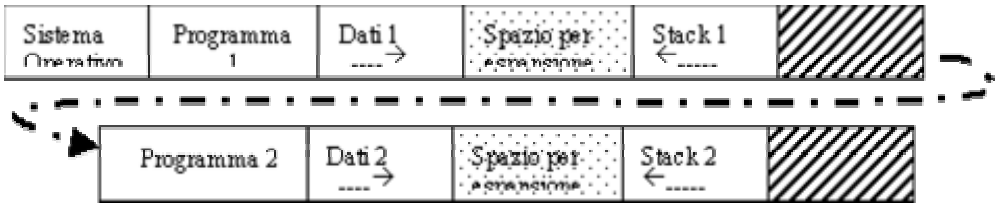


Figura 31- Allocazione di memoria con i segmenti dati e stack che crescono

5.3.3. Paginazione

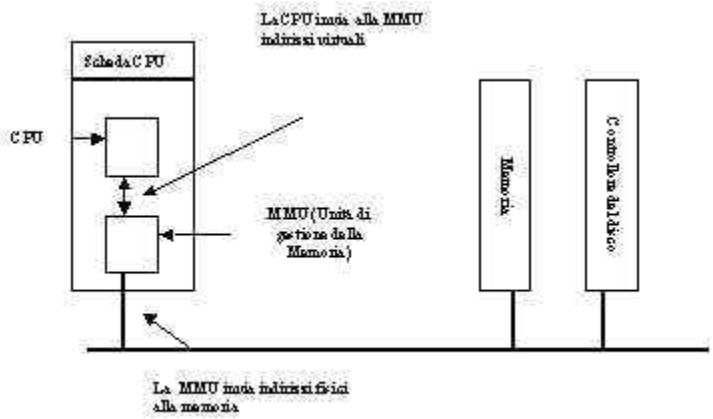
Negli schemi precedentemente descritti, la dimensione dello spazio di indirizzamento è necessariamente inferiore o uguale a quella dello spazio di memoria. Se si vuole assicurare al programmatore una memoria virtuale più ampia di quella fisica a disposizione, esentandolo dal faticoso compito di scrivere sovrapposizioni, dobbiamo concepire una mappa di indirizzamento che abolisca apparentemente la distinzione tra memoria centrale e secondaria. Il concetto di memoria ad un livello, in cui la memoria secondaria viene fatta apparire come un'estensione di quella centrale, è stato introdotto per la prima volta sull'elaboratore Atlas, presso l'Università di Manchester, verso il 1960 e da allora ha esercitato un profondo influsso sulla progettazione degli elaboratori.
La realizzazione della memoria ad un livello è possibile per mezzo della tecnica della paginazione, con la quale lo spazio virtuale degli indirizzi viene suddiviso in pagine di dimensione identica (sull'Atlas erano di 512 word) e la memoria centrale viene analogamente suddivisa in pagine (page frame) anch'esse omogenee per estensione. Le pagine della memoria centrale vengono condivise tra i processi esistenti correntemente nel sistema, cosicché in ogni momento un dato processo ha soltanto poche pagine residenti in memoria centrale (le sue pagine attive), mentre quelle rimanenti sono residenti su memoria di massa (le pagine inattive). Il meccanismo della paginazione svolge due funzioni:

1. esegue l'operazione di mapping degli indirizzi; determina, cioè, a quale pagina di programma si riferisce l'indirizzo e trova l'eventuale locazione di memoria centrale da essa occupato;
2. trasferisce le pagine dalla memoria secondaria alla centrale quando è necessario e le riporta su memoria di massa una volta terminato il loro utilizzo.

Descriveremo queste funzioni una per volta.
Per determinare la pagina cui si riferisce un indirizzo di programma, i bit più significativi degli indirizzi vengono interpretati come numero di pagina, mentre quelli meno significativi rappresentano il numero di word entro la pagina data. In tal

modo, se la dimensione della pagina corrisponde a 2n, gli n bit finali dell'indirizzo rappresentano il numero della parola, mentre i rimanenti bit quello della pagina di riferimento. Il numero totale di bit presenti nell'indirizzo è sufficiente per indirizzare l'intera memoria virtuale. A titolo d'esempio, sull'Atlas la lunghezza dell'indirizzo di programma era pari a 20 bit, il che dava luogo ad una memoria virtuale di 220 word; la dimensione della pagina era di 512 word (29) e perciò i 9 bit finali rappresentavano il numero di word, mentre gli 11 bit di testa rappresentavano il numero di pagina.
Il numero complessivo di pagine contenute in memoria centrale, pertanto, ammontava a 211 (rispetto alle 32 pagine presenti nella memoria fisica originaria).
Vale la pena sottolineare che la divisione dell'indirizzo in numero di word e di pagina è una funzione eseguita dall'hardware ed è invisibile al programmatore. Per quanto lo riguarda, infatti, la sua attività di programmazione si svolge entro il vasto spazio sequenziale di indirizzamento.

Figura 32 - posizione e le funzioni della Memory management Unit



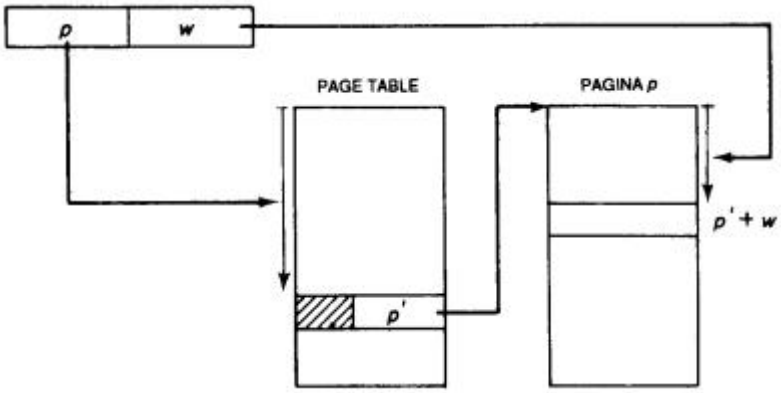
La mappa di indirizzamento che rimanda dai numeri di pagina e di word alla locazione fisica in memoria si realizza per mezzo di una page table (tabella delle pagine) in cui la variabile p-esima contiene la locazione p' dello spazio (page frame) contenente la pagina numero p. (Ci occuperemo tra un istante dell'eventualità in cui la pagina p-esima non sia in memoria centrale). Si aggiunge a p' il numero di word w e si ottiene così la locazione richiesta (si veda la figura 33).
La mappa di indirizzamento è perciò la seguente:

```
f(a)=f(p, w)=p'+w
```

dove l'indirizzo di programma a, la pagina numero p e la parola numero w sono correlati con la dimensione della pagina Z in base a quanto segue:

```
p := parte intera di (a/Z)
w := resto di (a/Z)
```

Figura 33 - Page table



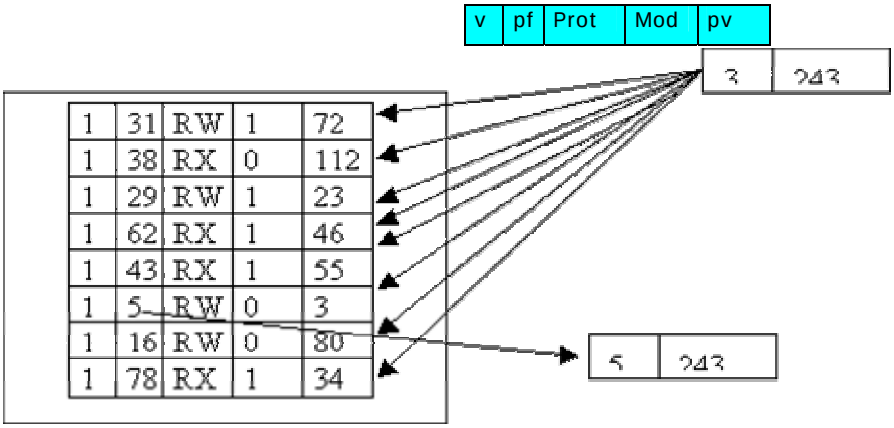
Poiché solitamente il numero di pagine (la quantità di memoria reale) allocate per un processo è inferiore al numero di pagine utilizzate realmente dallo stesso è possibile che un indirizzo di programma faccia riferimento ad una pagina che non si trova in memoria centrale in quel momento. In tal caso, la variabile corrispondente nella page table sarà vuota e, qualora si tenti di accedervi, verrà generato un interrupt di “page fault” (errore per mancanza di pagina). L'interrupt fa sì che il meccanismo di paginazione avvii il caricamento in memoria centrale della pagina mancante, presente in quel momento su memoria di massa, aggiornando di conseguenza la page table. Il processo corrente viene messo in attesa d'eventi, fino a che non sia completato il caricamento. La locazione della pagina all'interno della memoria secondaria può essere contenuta in una tabella separata o nella stessa page table. In quest'ultimo caso è necessario un “bit di presenza” in ciascuna variabile della page table, per indicare se la pagina è presente o meno in memoria centrale e se il campo dell'istruzione relativa all'indirizzo va inteso come rappresentante una pagina libera o una locazione di memoria di massa. Se non esiste alcuna pagina libera vuota al verificarsi di un page fault, si dovrà spostare qualche altra pagina verso la memoria secondaria, al fine di creare spazio per la pagina entrante. La scelta della pagina da scaricare (swap out) in tal modo è determinata dal risultato di un algoritmo di paginazione che prenderemo in esame, assieme a vari altri algoritmi, nei paragrafi successivi. Per il momento osserviamo che le informazioni necessarie a questo algoritmo possono essere

contenute in un piccolo numero di bit aggiunti a ciascuna variabile della page table (la parte tratteggiata in fig. 33). Tali informazioni possono consistere in:

- numero di volte che si è fatto riferimento ad una pagina;
- l'ultima volta che si è fatto riferimento alla pagina;
- se la pagina è stata oggetto di scrittura, o meno.

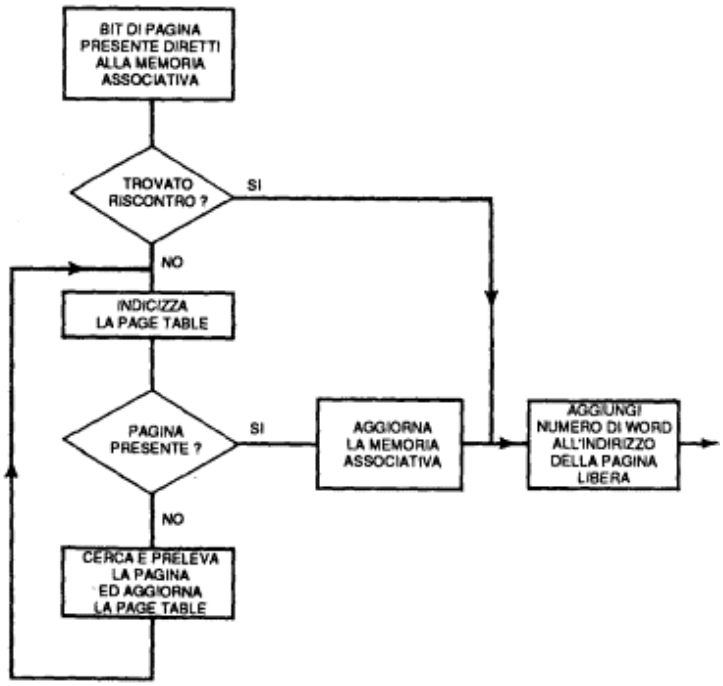
Bisogna forse chiarire che l'intera operazione di mapping degli indirizzi viene svolta dall'hardware (La page table è contenuta nella MMU), a parte il caso in cui una pagina deve essere riportata in memoria centrale dalla memoria secondaria. In quest'ultima evenienza, l'applicazione dell'algoritmo di paginazione e la modifica della page table vengono eseguiti dal software.
La disamina precedente ci dà un'idea generica del funzionamento del meccanismo di paginazione, anche se all'atto pratico vi si devono apportare parecchie modifiche al fine di ottenere un'implementazione reale. In particolare, nel sistema descritto il tempo necessario per effettuare ciascun riferimento in memoria in realtà viene raddoppiato dalla necessità di accedere dapprima alla page table. Un modo per ovviare a questo problema potrebbe consistere nel mantenere la page table in una serie di registri veloci, anziché in una memoria ordinaria. Comunque, la dimensione della page table è proporzionale a quella dello spazio di indirizzamento e perciò il numero dei registri necessari diventerebbe troppo vasto, tanto da non rendere pensabile la sua realizzazione per motivi di ordine economico.

Figura 34 - Registri PAR



La soluzione del problema consiste nell'adottare una tecnica molto diversa per l'accesso alle pagine attive. Questa tecnica richiede di inserire nella macchina una memoria associativa consistente in un insieme ridotto di registri per l'indirizzamento delle pagine (page address registers; PAR), ognuno contenente il numero di una pagina attiva. I PAR hanno la proprietà di poter essere oggetto di ricerca simultanea per ogni numero di pagina relativo ad una particolare istruzione di programma. Per esempio, in figura 5.3 l'istruzione di programma 3243 è suddivisa in numero della pagina (3) e numero della word (243) (si suppone per comodità che la grandezza della pagina sia pari a 1000 word). Il numero di pagina, dunque, viene raffrontato contemporaneamente con il contenuto di tutti i PAR, riscontrando la sua corrispondenza con il PAR numero 5. Se ne ricava l'indicazione che la pagina numero 3 occupa in quel momento la page frame numero 5 e che, pertanto, la locazione logica richiesta è la 5243.

Figura 35 - Ricerca del frame in un sistema con PAR

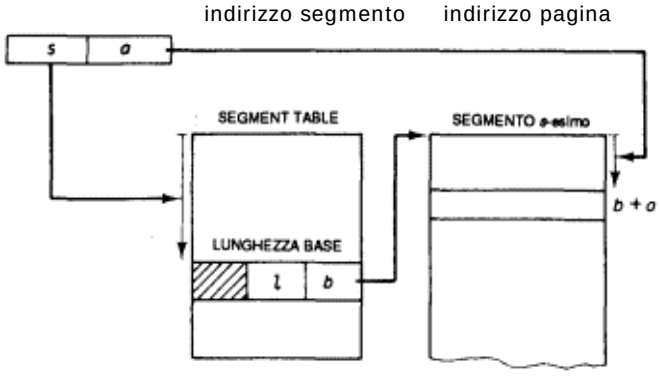


L'impiego di una memoria associativa riduce l'appesantimento della mappa di indirizzamento di un ordine di grandezza rispetto a quello sostenuto per via di una page table contenuta in memoria centrale.
Affinché tutte le pagine attive facciano riferimento ad un PAR, c'è bisogno di tanti PAR quante sono le pagine libere in memoria. Ciò risulta possibile in quei sistemi con memoria di dimensioni ridotte (ad esempio, l'Atlas), ma per i sistemi più grossi non è pensabile, per motivi economici, di poter prevedere tutti i PAR necessari, anche se ci si può aspettare che i ragionamenti di ordine economico mutino con lo sviluppo tecnologico. Nel caso in esame, si può giungere ad una soluzione di compromesso mantenendo in memoria una page table completa per ogni processo ed utilizzando una piccola memoria associativa per fare riferimento ad un numero limitato di pagine relative ai processi che sono stati attivi più recentemente. Stando così le cose, la pagina libera cui fa riferimento ciascun PAR non è più implicita nella posizione dello stesso all'interno della memoria associativa, ma deve essere inserita in un campo aggiuntivo facente parte del PAR stesso. L'hardware di indirizzamento della memoria svolge, quindi, l'operazione di mapping degli indirizzi mostrata in Figura 35. Anche questa volta l'intervento del software è necessario solo per la sostituzione della pagina.
Un problema non illustrato in figura 35 consiste nel distinguere, nell'ambito della memoria associativa, tra le pagine pertinenti al processo corrente e quelle appartenenti ad altri processi. Una soluzione consiste nell'ampliamento dei PAR, includendovi l'identificatore del processo, nonché il numero di pagina. Ciascun indirizzo presentato alla memoria associativa deve così comprendere, oltre all'identificatore del processo, anche i bit di pagina. Una soluzione alternativa consiste nell'aumentare i PAR di un unico bit che viene settato ad uno per le istruzioni del processo corrente ed a zero negli altri casi. Questa soluzione comporta necessariamente la disponibilità di una memoria associativa distinta per ogni processore presente nella configurazione, anche se è consigliabile prevederla comunque affinché la logica di indirizzamento della memoria non rappresenti una strozzatura (bottleneck) tra i processori e la memoria stessa.
Risulta chiaramente auspicabile che la memoria associativa contenga i numeri di quelle pagine cui, con più probabilità, si farà riferimento. Purtroppo non esiste alcun algoritmo generico in grado di garantire l'avverarsi di tale situazione (si confronti il paragrafo 5.4 circa il movimento di pagine tra memoria secondaria e centrale); in pratica, la memoria associativa solitamente viene riempita ciclicamente con gli indirizzi delle pagine cui si è fatto riferimento più di recente. Questo algoritmo piuttosto rudimentale in pratica risulta decisamente efficace; con una memoria associativa formata solamente da 8 o 16 registri, a seconda dell'estensione della memoria centrale, si può prevedere un hit rate medio superiore al 99%.
Se si accede alla page table tramite indirizzamento diretto e non per mezzo di indici, se ne può ridurre la dimensione al numero di pagine effettivamente in uso, anziché inserirvi tutte quelle presenti nello spazio di indirizzamento. Il concetto di fondo consiste nel far sì che la page table faccia riferimento solo a quelle pagine che sono state usate, invece di contenere una voce di riferimento per ogni pagina presente nello spazio di indirizzamento.
Ogniqualvolta si porta in memoria centrale una pagina nuova, il suo numero e l'indirizzo della pagina libera corrispondente vengono inseriti nella page table, nel punto determinato da un'opportuna funzione di indirizzamento casuale. Questa stessa funzione può essere impiegata per locallzzare le pagine durante l'operazione di mapping degli indirizzi. Lo svantaggio derivante dalla riduzione dell'estensione della page table consiste in un aumento del tempo d'accesso alla memoria necessario per alcuni riferimenti. Ciò è dovuto a due fattori: da un lato bisogna considerare il tempo utilizzato per il calcolo della stessa funzione di indirizzamento casuale e, dall'altro, nel caso in cui siano in contrasto le voci della page table determinata dalla funzione, può darsi che sia necessario accedervi più di una volta prima di locallzzare la pagina richiesta. Ciò nonostante, questo schema può divenire fattibile servendosi di una memoria associativa che è in grado di diminuire, fino a circa l'1% del totale, il numero di riferimenti alla page table, nonché eseguendo l'operazione di indirizzamento casuale tramite hardware.

5.3.4 Segmentazione

Il terzo obiettivo del paragrafo 5.1, più precisamente l'organizzazione dello spazio di indirizzamento in modo da rispecchiare le suddivisioni logiche fra programmi e dati, si può raggiungere applicando la tecnica della segmentazione. Il concetto consiste nel dividere in segmenti lo spazio di indirizzamento, ciascuno dei quali corrisponde ad una procedura, un modulo di programma o una raccolta di dati. Vi si può riuscire in modo semplice, aggiungendo a ciascun processore svariate coppie di registri limite e base, cosicché sia possibile delimitare altrettante zone entro lo spazio di indirizzamento. Per esempio, l'elaboratore Modular-One possedeva tre coppie di registri ed i tre segmenti relativi sono utilizzati rispettivamente per il programma, i dati riservati ad un processo e quelli in condivisione tra più processi. I lati negativi di questo semplice meccanismo consistono nel fatto che il numero dei segmenti è necessariamente limitato, per motivi economici, e che è indispensabile stabilire una convenzione per determinare quali segmenti sono utilizzati ed a quale scopo.

Figura 36 - Semplice mappa di indirizzamento per segmentazione



Un uso più elastico della memoria virtuale consiste nel consentire al programmatore di fare uso di un vasto numero di segmenti, così come ritiene più opportuno, facendovi riferimento tramite nomi da lui stesso assegnati. Lo spazio di indirizzamento diviene così bidimensionale, poiché le singole istruzioni di programma sono individuate attribuendovi sia un nome di segmento che un indirizzo dentro al segmento relativo. In realtà, per motivi di convenienza in fase di implementazione, il sistema operativo sostituisce il nome del segmento con un numero unico, quando vi fa riferimento per

la prima volta. Generalizzando, un'istruzione di programma pertanto è formata da una coppia di valori (s, a) in cui s è il numero di segmento ed a è l'indirizzo dentro il segmento stesso.
La mappa di indirizzamento può essere implementata prevedendo per ogni processo una segment table (tabella dei segmenti) la cui voce s-esima contiene la locazione di base e la lunghezza del segmento s-esimo del processo. A volte le voci della segment table vengono denominate descrittori di segmento. In forma semplificata, l'operazione di mapping (si veda la fig. 36) si svolge nel modo seguente:

a. preleva l'istruzione di programma (s, a);
b. utilizza s per indicizzare la segment table;
c. se a < 0 oppure se a > l, allora violazione di memoria;
d. (b+a) è la locazione di memoria richiesta.

Si attua la protezione da eventuali violazioni di memoria raffrontando l'indirizzo di word con la lunghezza l del segmento. Si può prevedere una protezione maggiore, inserendo in ciascun descrittore di segmento un numero di bit di protezione (la parte che in figura 36 è ombreggiata) indicanti le modalità secondo cui si può accedere al segmento corrispondente.
Si osservi che un segmento può anche essere facilmente condiviso tra svariati processi. Basta prevedere un descrittore per il segmento interessato, all'interno della segment table di ogni processo. I bit di protezione presenti in ogni descrittore possono essere diversi, in modo che, ad esempio, un processo possa accedere ad un segmento in condivisione con modo "solo lettura", mentre un altro processo potrebbe essere in grado di effettuarvi anche operazioni di scrittura. La segmentazione consente così una condivisione elastica di programmi e dati tra i processi.
Vale la pena mettere in rilievo che, nonostante l'apparente somiglianza tra la figura 34 e la figura 36, le mappe di indirizzamento per la paginazione e per la segmentazione sono del tutto diverse per le seguenti caratteristiche:

- lo scopo della segmentazione consiste nella suddivisione logica dello spazio di indirizzamento, mentre quello della paginazione consiste nella divisione fisica della memoria per realizzarne una ad un livello;
- le pagine hanno un'estensione fissa determinata dall'architettura della macchina, mentre i segmenti possono essere di una qualsiasi dimensione stabilita dall'utente (fino ad un limite massimo dipendente dal modo in cui l'istruzione di programma è suddivisa in numeri di segmento e word);
- la suddivisione dell'istruzione di programma in numero di word e numero di pagina è una funzione dell'hardware e, nel caso in cui il numero di word superi un determinato valore, si provoca automaticamente l'incremento del numero di pagina, mentre la divisione in numero di segmento e numero di word è di tipo logico e non vi è passaggio automatico al segmento successivo nell'eventualità di un overflow nel numero di word (se ciò avviene si genera una violazione di memoria).

In pratica, l'operazione di mapping degli indirizzi non è così semplice come sembra suggerire quanto detto in precedenza. La maggiore complicazione è data dal fatto che per programmi di grosse dimensioni può non essere possibile contenere in memoria centrale tutti i segmenti, specialmente a causa della condivisione della stessa tra i vari processi. Effettivamente si tratta di una situazione in cui la memoria virtuale è più ampia della memoria fisica e la si può gestire o impiegando la paginazione, oppure caricando e scaricando interi segmenti dalla memoria, a seconda delle esigenze.

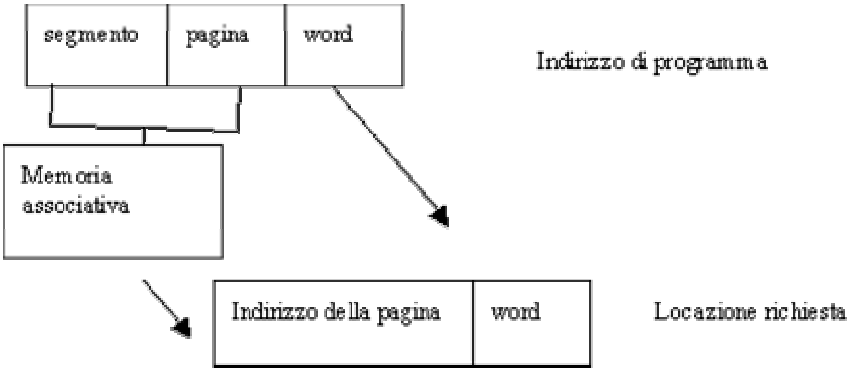
5.3.5 Segmentazione con paginazione

Se si fa uso della paginazione, generalmente ogni segmento è formato da varie pagine e possiede la sua propria page table. A condizione che alcune delle pagine del segmento si trovino in memoria, la voce della segment table indica la page table del segmento; altrimenti è vuota. L'operazione di mapping degli indirizzi svolta dall' hardware d'indirizzamento risulta dunque la seguente:

a. preleva l'istruzione di programma (s, a);
b. utilizza s per indicizzare la segment table;
c. se la s-esima voce è vuota, allora crea una nuova (vuota) page table, altrimenti preleva l'indirizzo della page table;
d. separa l'indirizzo di word a nel numero di pagina p e nel numero di word w;
e. utilizza p per indicizzare la page table;
f. se la voce p-esima è vuota, allora cerca e preleva la pagina dalla memoria di massa, altrimenti preleva l'indirizzo di pagina libera p';
g. aggiungi p' al numero di parola w per ottenere la locazione richiesta.

I passi (a), (b) e (c) rappresentano il mapping dovuto a segmentazione; i passi (d), (e), (f) e (g), corrispondenti alla figura 37 rappresentano il mapping dovuto alla paginazione dei segmenti.
E possibile evitare i riferimenti aggiuntivi di memoria richiesti per accedere alla segment table ed alla page table ricorrendo ad una memoria associativa, in modo analogo a quanto descritto precedentemente a proposito della sola paginazione. In tal caso, ciascun elemento della memoria associativa contiene il numero di segmento e il numero di pagina per quelle pagine cui si è avuto accesso più recentemente. I bit di segmento e di pagina

Figura 37 - di mapping degli indirizzi per segmenti paginati con l'impiego di memoria associativa



per ogni istruzione di programma vengono presentati alla memoria associativa (figura 28) e, se si trova riscontro, il numero di parola viene aggiunto all'indirizzo della pagina libera corrispondente, ottenendo la locazione di memoria richiesta. Solamente se non si trova riscontro viene chiamata l'intera operazione descritta in antecedenza.
Se non ci si serve della paginazione, le voci della segment table si presentano come indicato in figura 30. Un errore dovuto alla mancanza di un segmento fa sì che venga portato in memoria centrale l'intero segmento richiesto, mentre può darsi che qualche altro segmento venga relegato su memoria di massa al fine di creare spazio a sufficienza per contenere quello desiderato. I criteri relativi all'allocazione della memoria tra i vari segmenti saranno descritti nel paragrafo che segue.
I vantaggi derivanti dall'impiego della paginazione per implementare uno spazio di indirizzamento segmentato sono di duplice natura. In primo luogo, non è necessario che l'intero segmento sia presente in memoria centrale (vi devono essere sistemate solo le pagine in uso corrente). In secondo luogo, non è necessario che le pagine occupino zone contigue di memoria: possono essere distribuite in tutta la memoria ed ubicate laddove si trovi una pagina libera appropriata. A sfavore della paginazione va considerata la complessità della mappa di indirizzamento e l'appesantimento determinato dalle page table. Esempi di sistemi dotati di segmenti paginati sono l'Honeywell 645, l'ICL 2900 e le versioni più grandi degli IBM 370; sistemi con segmenti non paginati sono, invece, ad esempio, le grosse macchine Burroughs (come il B6700) ed il PDP-II/45.

5.3.6 Segmentazione con paginazione: il Pentium Intel

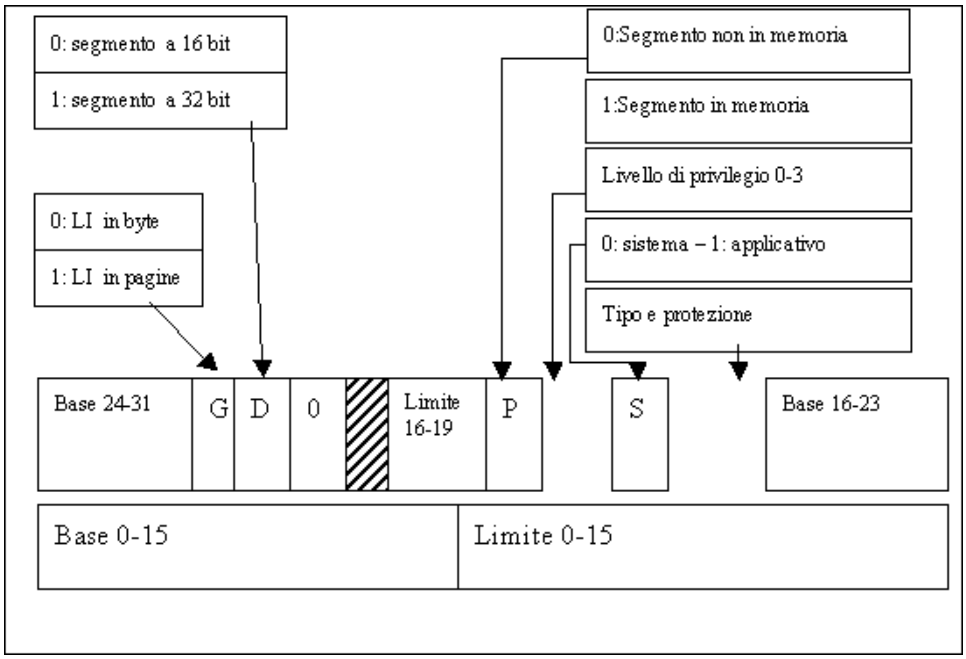
Per molti aspetti, la memoria virtuale del Pentium (e del Pentium Pro) ricorda MULTICS, inclusa la presenza sia della segmentazione che della paginazione. Mentre MULTICS ha 256K segmenti indipendenti, ognuno contenente fino a 64K di parole a 36 bit, il Pentium ha 16K segmenti indipendenti, ognuno contenente fino a 1 miliardo di parole a 32 bit. Benché vi siano meno segmenti, la maggiore dimensione del segmento è di gran lunga più importante, in quanto pochi programmi richiedono più di 1000 segmenti, ma molti programmi necessitano di segmenti in grado di contenere alcuni megabyte.
Il cuore della memoria virtuale del Pentium consiste di due tabelle, la LDT (Local Descriptor Table) e la GDT (Global Descriptor Table). Ogni programma ha la sua LDT, mentre vi è una sola GDT, condivisa da tutti i programmi sul computer. La LDT descrive i segmenti locall a ogni programma, compresi il suo codice, i suoi dati, il suo stack e così via, mentre la GDT descrive i segmenti di sistema, compreso il sistema operativo stesso.
Per accedere a un segmento, un programma Pentium carica innanzitutto il selettore del segmento in uno dei sei registri di segmento della macchina. Durante l'esecuzione, il registro CS contiene il selettore del segmento codice e il registro DS contiene il selettore del segmento dati. Gli altri segmenti sono meno importanti. Ogni selettore è un numero a 16 bit, come mostrato in figura.

Figura 38 - Un selettore del Pentium

INDICE 13 bit	0=GDT 1=LDT	2 bit 0-3 Priv

Uno dei bit del selettore indica se il segmento è locale o globale (cioè se è contenuto nella LDT o nella GDT). Altri tredici bit specificano l'indice dell'elemento nella LDT o nella GDT. Ognuna di queste tabelle può pertanto contenere fino a un massimo di 8K descrittori di segmento. Gli altri 2 bit sono relativi alla protezione e saranno descritti in seguito. Il descrittore 0 è proibito. Esso può essere tranquillamente caricato in un registro di segmento per indicare che il registro di segmento non è attualmente disponibile. Questo registro, se usato, provoca un'eccezione.
Nell'istante in cui si carica un selettore in un registro di segmento, il descrittore corrispondente viene estratto dalla LDT o GDT e immagazzinato in registri da un microprogramma, in modo da potervi accedere velocemente. Un descrittore consiste in 8 byte, che comprendono l'indirizzo base del segmento, la dimensione e altre informazioni, come rappresentato in figura 39.
Il formato del selettore è stato abilmente scelto per facilitare l'individuazione del descrittore. Prima si seleziona o la LDT o la GDT, sulla base del bit 2 del selettore. Il selettore viene poi copiato in un registro interno di lavoro, e i 3 bit di ordine inferiore vengono posti a 0. Infine, l'indirizzo della LDT o della GDT viene aggiunto al valore nel registro per ottenere un puntatore diretto al descrittore. Il selettore 72 si riferisce ad esempio alla voce 9 della GDT, posizionata all'indirizzo GDT+72

Figura 39 - Descrittore di segmento nel Pentium



Tracciamo ora i passi con i quali una coppia (selettore, offset) viene convertita in un indirizzo fisico. Non appena il microprogramma sa qual è il registro di segmento usato, esso può individuare il descrittore completo corrispondente a tale selettore nei suoi registri interni. Se il segmento non esiste (selettore 0) o è attualmente memorizzato in una pagina sul disco, si ha un'eccezione.
A questo punto il microprogramma controlla se l'offset oltrepassa il limite del segmento, nel qual caso si ha un'altra eccezione. Logicamente, il descrittore dovrebbe semplicemente contenere un campo a 32 bit che fornisca la dimensione del segmento ma, essendoci solo 20 bit a disposizione, lo schema utilizzato è diverso. Se il campo Gbit (Granularità) è a 0, il campo Limit (Limite) è la dimensione esatta del segmento, fino a 1MB. Se Gbit è a 1, il campo Limit fornisce la dimensione del segmento in pagine anziché in byte. La dimensione di pagina del Pentium è fissata in 4K byte, di conseguenza 20 bit sono sufficienti per segmenti fino a 232 byte.
Assumiamo che il segmento sia presente in memoria e l'offset sia entro i limiti; il Pentium aggiunge il campo a 32 bit Base, contenuto nel descrittore, all'offset, per formare quello che viene chiamato un indirizzo lineare, come mostrato in figura 38. Il campo Base viene spezzato in tre parti e sparso all'interno del descrittore per compatibilità con il 286, nel quale il campo Base è costituito da soli 24 bit. In effetti, il campo Base consente di iniziare ogni segmento in un punto arbitrario all'interno dello spazio di indirizzamento lineare a 32 bit.
Se la paginazione viene disabilitata (usando un bit nel global control register), l'indirizzo lineare viene interpretato come indirizzo fisico e inviato alla memoria per la lettura o la scrittura dei dati. Quando la paginazione è disabilitata, abbiamo quindi uno schema a segmentazione puro, in cui l'indirizzo base di ogni segmento viene fornito all'interno del suo descrittore. I segmenti possono incidentalmente sovrapporsi, probabilmente perché costerebbe troppa fatica e prenderebbe troppo tempo verificare che essi siano disgiunti.
Viceversa, se la paginazione è abilitata, l'indirizzo lineare viene interpretato come indirizzo virtuale e tradotto nel corrispondente indirizzo fisico usando le tabelle delle pagine, più o meno come nei nostri esempi precedenti. La sola vera complicazione è che, con un indirizzo virtuale a 32 bit e una pagina di 4K, un segmento può contenere 1 milione di pagine: pertanto si usa uno schema di corrispondenza con tabelle a due livelli per ridurre la dimensione della tabella delle pagine dei segmenti piccoli.
Ogni programma in esecuzione ha un direttorio delle pagine che consiste in 1024 elementi a 32 bit. Esso è posizionato in un indirizzo puntato da un registro globale. Ogni elemento in questo direttorio punta a una tabella delle pagine contenente anch'essa 1024 elementi a 32 bit. Gli elementi nella tabella delle pagine puntano ai frame di pagina. Lo schema viene mostrato in figura 39.
In figura 39 vediamo un indirizzo lineare diviso in tre campi, Dir, Page e 0ff. Il campo Dir viene usato per indicizzare il direttorio delle pagine e individuare il puntatore appropriato alla tabella delle pagine. Il campo Page viene quindi usato come indice nella tabella delle pagine per trovare l'indirizzo fisico del fame di pagina. 0ff viene infine aggiunto all'indirizzo del frame di pagina per ottenere l'indirizzo fisico del byte o della parola richiesta.
Gli elementi nella tabella delle pagine sono ognuno di 32 bit, 20 dei quali contengono un indice di frame di pagina. I bit rimanenti contengono i bit di accesso e i bit di modifica, posti a 1 dall'hardware a vantaggio del sistema operativo, i bit di protezione e altri bit di utilità.
Ogni tabella delle pagine ha elementi sufficienti a contenere 1024 frame di pagina di 4K ciascuna una tabella delle pagine gestisce pertanto 4 megabyte di memoria. Un segmento più corto di 4M avrà un direttorio delle pagine con un solo elemento, ovvero puntatore alla sua unica e sola tabella delle pagine. In questo modo, la memoria necessaria per le tabelle dei segmenti più corti è di sole due pagine, invece del milione di pagine che sarebbero necessarie in una tabella delle pagine a un solo livello.
Per evitare di effettuare ripetuti riferimenti in memoria, il Pentium, come MULTICS, una piccola PAR che traduce direttamente negli indirizzi fisici dei fame di pagina combinazioni Dir-Page usate più di recente. Solo quando la combinazione corrente n è presente nella PAR il meccanismo della figura 39 viene effettivamente usato e PAR aggiornata.
Con un po' di attenzione ci si accorge che, quando si usa la paginazione, non c'è effettivamente alcun motivo per cui il campo Base nel descrittore sia diverso da zero. Base non fa altro che forzare l'uso di elementi situati in mezzo al direttorio delle pagine, invece che all'inizio. In assoluto, il vero motivo per il quale è stato inserito un campo base è per consentire la

segmentazione pura (non paginata), e per compatibilità con il 286 nel quale la paginazione era sempre disabilitata (il 286 ha cioè solo la segmentazione pura, non la paginazione).
Vale anche la pena di far notare che, nel caso in cui un'applicazione non abbia bisogno della segmentazione ma si accontenti di un singolo spazio di indirizzamento a 32 bit paginato, questo modello è possibile. Tutti i registri di segmento possono essere impostati con lo stesso selettore, il cui descrittore Base è uguale a 0 e Limit pari al massimo in paginazione.

5.4 METODI DI ALLOCAZIONE DELLA MEMORIA

Nel paragrafo che precede ci siamo occupati esclusivamente dei meccanismi per la gestione della memoria e non dei metodi che servono ad amministrarli. In questo paragrafo concentreremo la nostra attenzione sui metodi che consentono di ottimizzare l'utilizzo dei meccanismi visti.
I metodi di gestione della memoria rientrano in tre vaste categorie:

1. Metodi di sostituzione, che determinano quali informazioni vanno rimosse dalla memoria centrale creando, cioè, regioni di memoria disponibili;
2. Metodi di ricerca e prelievo (fetch), che specificano quando le informazioni vanno caricate in memoria centrale, ad esempio se dietro richiesta o in anticipo;
3. Metodi di posizionamento, che specificano dove vanno messe le informazioni all'interno della memoria centrale selezionando così un sottoinsieme di alcune regioni disponibili.

Vedremo che i metodi rispondenti al tipo (2) sono del tutto identici, sia per i sistemi paginati, che per quelli che non lo sono. I metodi di tipo (1) e (3), però, sono diversi nei casi di sistemi paginati o non paginati: le differenze nascono dal fatto che le pagine hanno una dimensione fissa, contrariamente alla variabilità dell'estensione dei blocchi di informazioni gestiti dai sistemi non paginati.

5.4.1 Metodi di posizionamento per sistemi segmentati

Prenderemo ora in considerazione il trasferimento di informazioni in memoria centrale per quei casi in cui lo spazio di indirizzamento virtuale è segmentato oppure implementato tramite coppie di registri limite e base. In entrambi i casi denoteremo con la parola “segmento” i blocchi di informazioni che sono in corso di trasferimento, premettendo che, se si tratta di un singolo sistema con registri limite e base, un segmento rappresenterà nella sua interezza lo spazio di indirizzamento di un processo.
La situazione che si presenta consiste nel periodico inserimento o cancellazione di segmenti dalla memoria centrale. Inserimento e cancellazione si verificano facendo ricorso, rispettivamente, ai metodi di ricerca e prelievo (fetch) e di sostituzione. Se il sistema è in uno stato di equilibrio (più precisamente, entro un certo intervallo di tempo, lo spazio liberato tramite cancellazione è uguale a quello riempito tramite inserimento) la memoria si presenta suddivisa come una scacchiera, come si vede in figura.

Figura 40 - Memoria segmentata

Risulta chiaro che il metodo di posizionamento deve mantenere una lista delle locazioni e delle dimensioni dei “buchi” non allocati: si tratta della hole list (lista dei buchi). Il compito del metodo di posizionamento consiste nel decidere quale buco utilizzare, aggiornando la lista in seguito a ogni inserimento. Se il segmento da posizionare è più piccolo del buco da utilizzare, verrà posto dalla parte “sinistra” ovvero sul “fondo” del buco. Questa tattica rende minima l'entità della frammentazione. Se, però, il segmento è più vasto di qualsiasi buco disponibile, il metodo di posizionamento ha il compito ulteriore di spostare i segmenti già in memoria per creare un buco largo a sufficienza per i propri scopi.
Nelle opere specifiche su questo argomento sono descritti numerosi algoritmi di posizionamento, per esempio quello di Knuth (1968). Qui accenneremo ai quattro più importanti. In tutti i casi, le dimensioni dei buchi sono indicate con x1, x2,...xn,.

(a) Il più adatto
I buchi sono messi in lista per ordine crescente di dimensione, cioè x1 <= x2 < =.....<=xn . Se s è la misura del segmento da posizionare, allora trova il più piccolo i tale che s <= xi.

(b) Il meno adatto
I buchi sono messi in lista per ordine decrescente di dimensione, cioè x1 > = x2 > =.....> xn=. Posiziona il segmento nel primo buco e reinserisci il buco formato dallo spazio rimanente nella posizione più opportuna entro la lista.

(c) Il primo buco adatto
I buchi sono elencati per ordine crescente di indirizzo base. Trova il più piccolo i tale che s < = xi . Con l'andar del tempo, l'impiego di questo algoritmo porta all'accumularsi di tanti piccoli buchi verso l'inizio della lista. Onde evitare un tempo di ricerca eccessivo per i segmenti più grandi, la posizione di inizio ricerca si può far avanzare ciclicamente di un elemento dopo ogni operazione di ricerca.

(d) Buco gemellare
Per questo algoritmo le dimensioni dei segmenti devono essere potenze di 2, cioè s=2i per un i inferiore ad un massimo k. Si predispone una lista distinta per tutti i buchi aventi dimensione pari a 21,22,...2k . Si può prelevare un buco dalla lista (i+ 1) dividendolo a metà, creando in tal modo una coppia di “gemelli” con dimensione 2i inserita nella lista degli i. Viceversa, una coppia di gemelli può essere tolta da quest'ultima lista, unendoli assieme e mettendoli nella lista (i+ 1). L'algoritmo che serve a trovare un buco di dimensioni 2i è ricorsivo:

```
procedure prendi un buco (i);
begin if  i = K + I then insuccesso;
    if  lista degli i è vuota then
            begin prendi un buco (i+ 1);
                    dividi il buco in due gemelli;
                    metti i gemelli nella lista degli i
```

```
            end;
            prendi il primo buco nella lista degli i
end
```

Al lettore diffidente potrebbe sembrare strano che gli algoritmi (a) e (b) siano entrambi attuabili, in quanto adottano strategie diametralmente opposte per quanto riguarda l'ordinamento della hole list. Forse l'algoritmo (a) è quello più interessante da un punto di vista intuitivo, poiché sembra rendere minimo lo spreco di spazio frammentato per ogni buco prescelto (seleziona, cioè, il buco più piccolo che risulta idoneo allo scopo). Tuttavia, l'algoritmo (b) si attiene alla filosofia secondo cui l'allocazione di uno spazio prelevato da un buco grande ha più probabilità di lasciare a disposizione un altro buco abbastanza grande da tornare utile in seguito, mentre l'allocazione di uno spazio prelevato da un buco piccolo comporta la creazione di un altro buco ancora più piccolo con scarse probabilità di utilizzo.
Un problema comune a tutti i metodi di posizionamento descritti consiste nella frammentazione: la suddivisione della memoria libera in buchi tanto piccoli da renderne impossibile l'occupazione da parte di nuovi segmenti. Se si verifica questa situazione risulta necessario un qualche compattamento della memoria: tutti i segmenti, cioè, devono essere spostati verso il basso in modo da far comparire un unico grande buco all'inizio della memoria. È più agevole eseguire il compattamento se i buchi sono listati per ordine di indirizzo, come accade nell'algoritmo (c). Una soluzione alternativa consiste nel compattare la memoria ogniqualvolta viene prelevato un segmento, evitando così completamente il fenomeno della frammentazione; l'appesantimento del sistema dovuto a frequenti operazioni di compattamento è compensato dal vantaggio di non dover effettuare operazioni di ricerca nella lista dei buchi ogniqualvolta si deve posizionare un segmento.

5.4.2 Metodi di posizionamento per sistemi paginati

I metodi di posizionamento per i sistemi paginati sono di gran lunga più semplici che non per i sistemi senza paginazione. Per posizionare k pagine impieghiamo semplicemente un criterio di sostituzione per liberare k spazi corrispondenti (page frames). La frammentazione, nel senso descritto per i sistemi non paginati, non può verificarsi, poiché tutte le pagine e le pagine libere hanno dimensioni identiche. Ciò nonostante, un sistema paginato può andare soggetto ad una forma diversa di frammentazione, la frammentazione interna, la quale deriva dal fatto che lo spazio necessario ad un processo solitamente non è un multiplo esatto della dimensione delle pagine. In conseguenza di ciò, una parte dell'ultima pagina libera allocata in genere va sprecata; si può prevedere che in media lo spreco corrisponda ad una mezza pagina. Se lo spazio di indirizzamento è formato da più segmenti paginati, il grado di frammentazione interna si moltiplica per il numero dei segmenti correntemente contenuti in memoria centrale.

5.4.3 Metodi di sostituzione per sistemi paginati

Il compito di un metodo di sostituzione consiste nel decidere quali blocchi di informazioni relegare su memoria di massa al momento di trovare spazio per nuovi blocchi in memoria centrale. In questa sezione del paragrafo 5.4 prendiamo in considerazione il caso in cui i blocchi di informazioni corrispondono a pagine; vedremo più avanti che si possono applicare gli stessi metodi, con opportune modifiche, in situazioni di assenza di paginazione.
In linea di principio, si vuole sostituire quella pagina che in futuro non sarà richiamata se non dopo un lungo intervallo di tempo. Purtroppo, è sempre difficile fare dei pronostici e la cosa migliore da farsi è desumere, dall'andamento precedente, qual è l'andamento che probabilmente si avrà in futuro. L'accuratezza della deduzione dipende dalla prevedibilità del comportamento di un programma, cosa che prenderemo in esame ulteriormente in questo stesso paragrafo, in riferimento ai metodi di ricerca e prelievo (fetch).
I tre algoritmi di sostituzione più comuni sono esposti di seguito.

(a) Pagina usata meno recentemente (LRU; least recently used)
"Sostituisci la pagina usata meno recentemente". Alla base vi è il presupposto che il comportamento futuro segua da vicino quello più recente. L'appesantimento relativo gravante sul sistema consiste nel registrare la sequenza di accesso a tutte le pagine.

(b) Pagina usata meno frequentemente (LFU; least frequently used)
"Sostituisci la pagina che è stata usata meno frequentemente durante l'intervallo di tempo immediatamente precedente". La giustificazione in questo caso è analoga a quanto detto nel precedente punto (a) e il conseguente appesantimento del sistema consiste nel prevedere un "conteggio di utilizzo" (use count) per ogni pagina. Uno svantaggio è dato dal fatto che, in generale, una pagina caricata recentemente avrà un numero basso di conteggio di utilizzo e potrebbe pertanto venire sostituita sconsideratamente. Per evitare tale fenomeno si può inibire la sostituzione di quelle pagine che sono state caricate durante l'intervallo di tempo immediatamente precedente.

(c) FIFO (First In-First Out)
"Sostituisci la pagina che è rimasta più a lungo residente in memoria". Si tratta di un algoritmo semplice che appesantisce il sistema solamente a causa della necessaria registrazione della sequenza di caricamento delle pagine. Non tiene conto, però, della possibilità che la pagina da più tempo in memoria sia anche quella cui si fa riferimento più spesso.
Studi svolti in simulazione (Colin, 1971) hanno dimostrato che esiste una diversità di prestazioni tra i tre algoritmi (in relazione al numero di trasferimenti richiesti durante l'esecuzione di svariati lavori) che, pur variando a seconda del tipo di lavoro in corso d'esecuzione, raramente ammonta ad oltre il 15%. In generale, l'algoritmo (c) offre prestazioni più scarse rispetto agli altri due.
Infine, è importante osservare che, se le pagine non sono state modificate, non è necessario riportarle su memoria secondaria, a condizione che in quest'ultima ne esista già una copia. Si può prendere nota dell'eventuale avvenuta modifica di una pagina particolare utilizzando un solo bit della voce della page table corrispondente.

5.4.4 Metodi di sostituzione per sistemi non paginati

In questa parte del paragrafo prendiamo in considerazione i metodi idonei a situazioni in cui i blocchi di informazioni da sostituire consistono in segmenti, nel senso più ampio del termine, così come definito al precedente punto (1).
L'obiettivo principale è lo stesso dei sistemi paginati: sostituire, cioè, il segmento che ha meno probabilità di venire utilizzato nell'immediato futuro. Si potrebbe prevedere, pertanto, che risultino validi gli stessi metodi. Ciò è quanto realmente avviene, anche se con una restrizione fondamentale, dovuta al fatto che non tutti i segmenti in questo caso occupano spazi identici di memoria. Perciò, la considerazione relativa a quale segmento riportare in memoria secondaria viene influenzata dalla dimensione dello stesso. Se si deve portare in memoria centrale un segmento piccolo, sarà necessario sostituire solo un segmento piccolo; d'altra parte, il posizionamento di un segmento più vasto richiede la sostituzione di un segmento altrettanto vasto (o di svariati segmenti più piccoli).

Decisamente l'algoritmo più semplice consiste nel sostituire quel singolo segmento (sempre che esista) il quale, assieme ad eventuali buchi contigui, liberi uno spazio sufficiente per il segmento entrante. Nel caso esistano più segmenti di tal genere in grado di soddisfare questa condizione, per selezionarne uno si può usare uno dei metodi esaminati in precedenza, per esempio il LRU. Se nessun segmento singolo è abbastanza vasto da creare spazio sufficiente, bisognerà sostituire più segmenti; una possibilità di scelta consiste nel privilegiare il più piccolo gruppo di segmenti contigui in grado di liberare lo spazio occorrente.
Il pericolo intrinseco in un algoritmo di questo tipo è che il segmento (o gruppo di segmenti) sostituito, essendo scelto prevalentemente in base a criteri dimensionali, potrebbe risultare nuovamente necessario entro breve tempo. Si può ridurre tale rischio selezionando i segmenti meramente sulla base, poniamo, del metodo LRU, ma poiché i segmenti prescelti probabilmente non saranno contigui, risulterà necessaria qualche forma di compattamento. Il lettore comprenderà agevolmente che l'importanza relativa fatta ricadere sulla dimensione dei segmenti, sulla previsione degli andamenti futuri e sul compattamento, può dare luogo a numerosi algoritmi complessi, difficili da valutare se non nella pratica. Non daremo qui seguito a tale argomento, bensì rimandiamo il lettore interessato a studi più particolareggiati, come quelli di Knuth (1968) e Denning (1970).

5.4.5 Metodi di ricerca e prelievo (fetch)

I metodi di ricerca e prelievo (fetch) determinano quando spostare un blocco di informazioni dalla memoria secondaria in quella centrale. Le argomentazioni che spingono a preferire un metodo anziché un altro sono pressoché identiche, sia che i blocchi siano formati da pagine, che da segmenti (intesi in senso lato). I metodi di ricerca e prelievo vengono suddivisi in due gruppi principali: prelievo su richiesta ed anticipato. I metodi operanti su richiesta prelevano i blocchi quando sono necessari, mentre i metodi di anticipazione li prelevano precedentemente al manifestarsi delle esigenze.
È ovvio che i metodi su richiesta sono i più facili da mettere in pratica: una condizione di blocco mancante (segmento o pagina che sia) genera una richiesta di ricerca e prelievo, mentre i metodi di posizionamento e/o sostituzione provvedono all'allocazione di memoria per il nuovo blocco. Nei sistemi non paginati solitamente il trasferimento dei blocchi avviene su richiesta, così come pure avviene per alcuni elaboratori che si servono della paginazione. I metodi d'allocazione anticipata, per essere pienamente efficaci, fanno affidamento sulle previsioni relative alla tendenza futura del comportamento di un programma. Tali previsioni si possono basare su due fattori:

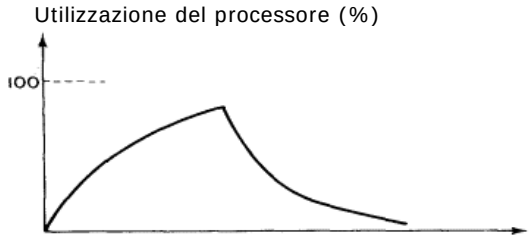
a. la natura della costruzione dei programmi;
b. le inferenze fondate sul comportamento precedente di un processo.

Si consideri per un momento il punto (a).
Molti programmi presentano un comportamento definito comunemente operativo in un contesto: suddividendo il tempo d'esecuzione in tanti brevi intervalli, nell'ambito di ciascuno di essi un programma tende ad operare entro un modulo logico particolare, traendo le proprie istruzioni da un'unica procedura ed i propri dati da un'area singola. I riferimenti di programma tendono così ad essere raggruppati in piccole locazioni di spazio logico. La locazione di riferimento si rafforza per mezzo del frequente verificarsi di cicli iterativi; più è ridotto il ciclo, minore è la frammentazione dei riferimenti. L'osservazione di questo comportamento ha condotto al postulato (Denning, 1970) del cosiddetto principio di localltà: “i riferimenti di programma tendono ad essere raggruppati in zone di spazio logico circoscritte, le quali tendono a mutare solo in modo discontinuo e periodico”.
La validità del principio di localltà varierà da programma a programma; sarà maggiore, per esempio, nel caso di programmi che accedono ad array sequenziali, rispetto a programmi che accedono a strutture di dati complesse. Il principio è stato impiegato da Denning nell'ambito di una memoria paginata per formulare il modello dell'area di lavoro relativo al comportamento dei programmi, di cui daremo una breve descrizione nel paragrafo successivo.

5.5 IL MODELLO DELL'AREA DI LAVORO

Il modello dell'area di lavoro per il comportamento dei programmi (Denning, 1968) rappresenta il tentativo di definire una struttura portante per poter comprendere le prestazioni dei sistemi paginati in condizioni di multiprogrammazione. I metodi di gestione della memoria da noi esaminati nel paragrafo precedente si basano sullo studio relativo al modo in cui si comportano i processi in stato di isolamento; non tengono conto di eventuali interazioni che potrebbero verificarsi per il fatto di avere in macchina parecchi processi alla volta. Infatti, il competere per l'utilizzo dello spazio di memoria può portare i processi ad un comportamento che non si verificherebbe se ognuno di essi fosse eseguito separatamente dagli altri.
A dimostrazione di ciò che può succedere, si consideri un sistema dotato di un solo processore e memoria paginata, nel quale il livello di multiprogrammazione (cioè il numero di processi presenti in memoria) è in costante aumento. Si potrebbe prevedere che, col crescere del livello di multiprogrammazione, aumenti pure l'utilizzazione del processore, poiché la routine di smistamento avrebbe sempre maggiori probabilità di trovare un processo da mettere in esecuzione. A dire il vero, l'osservazione ci conferma che in linea generale ciò è quanto avviene, fintanto che il livello di multiprogrammazione viene mantenuto al di sotto di una determinata soglia, in funzione dell'estensione della memoria a disposizione. Tuttavia, se il livello di multiprogrammazione supera tale limite, si riscontra (vedi figura 41) un marcato aumento del movimento di paginazione tra memoria centrale e secondaria, accompagnato da un'improvvisa flessione dell'utilizzazione del processore. La spiegazione di questo fenomeno sta nel fatto che l'alto livello di multiprogrammazione rende impossibile manténere in memoria per tutti i processi pagine sufficienti ad evitare la generazione di un numero enorme di interrupt causati da page fault. Ciò significa che il canale della memoria di massa può diventare saturo, che la maggior parte dei processi resta bloccata in attesa di un trasferimento di pagina e che il processore viene sottoutilizzato. Questo stato di cose viene definito con il termine descrittivo di thrashing (letteralmente “batosta”).

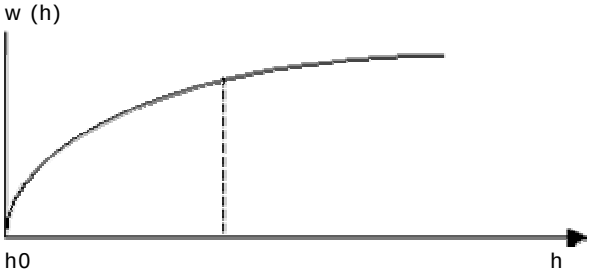
Figura 41 - Il fenomeno del thrashing



Livello di multiprogrammazione

L'insegnamento da trarre da questo esempio dimostrativo è che per ogni processo è necessario mantenere in memoria centrale un determinato numero minimo di pagine, l'area di lavoro propria del processo, prima che esso possa utilizzare effettivamente la CPU. Se le pagine presenti in memoria sono inferiori a tale minimo, il processo verrà interrotto continuamente per l'assenza di pagine, il che contribuisce a dar luogo al fenomeno del thrashing. Per evitare questa situazione, il livello di multiprogrammazione non deve superare il livello che consente di tenere in memoria centrale le aree di lavoro relative a tutti i processi.
A questo punto sorge la questione relativa all'individuazione delle pagine che costituiscono l'area di lavoro di un processo. La soluzione consiste nell'esaminare accuratamente la storia recente del processo interessato e nel fare ricorso al principio di localltà ricordato nel paragrafo precedente. In modo più formale, l'area di lavoro di un processo all'istante t è definita come
w(t,,h) =[pagina i | pagina i c N e pagina i compare negli ultimi h riferimenti]
In altri termini, l'area di lavoro è data dall'insieme di pagine cui si è fatto riferimento recentemente, dove "l'essere recente" è uno dei parametri (h) dell'insieme. In base al principio di localltà si potrebbe pensare che gli elementi dell'area di lavoro cambino lentamente nel tempo. Denning ha dimostrato che la dimensione w(h) prevista per l'area di lavoro varia al variare di h, come illustrato in figura.

Figura 42 - Previsione dell'estensione dell'area di lavoro



Con l'aumentare di h (cioè, più si guarda indietro nel tempo) si prevede di riscontrare sempre meno pagine aggiuntive nell'area di lavoro. Ciò consente di fissare per h un valore (poniamo h0) opportunamente piccolo, nella consapevolezza che anche un valore più alto non darebbe luogo ad un aumento significativo dell'estensione dell'area di lavoro.
Per quanto riguarda i metodi di sostituzione e di ricerca e prelievo, il valore dell'area di lavoro si basa sulla seguente regola:
"Esegui un processo solo se tutta l'area di lavoro relativa si trova in memoria centrale e non spostare mai una pagina facente parte dell'area di lavoro di un processo".
Sebbene le aree di lavoro di alcuni processi siano definite in modo piuttosto arbitrario e benché il principio di localltà non sia valido per alcuni programmi, l'applicazione della regola suesposta può dare un contributo significativo alla prevenzione del fenomeno del thrashing. Il modello dell'area di lavoro viene largamente usato, nonostante tra i sistemi operativi la definizione dell'area di lavoro sia piuttosto variabile.
È importante comprendere appieno come la regola data in precedenza sia più di un puro metodo di gestione della memoria, in quanto rende implicita la correlazione tra allocazione di memoria ed allocazione della CPU. Fino ad ora nel presente capitolo ci siamo occupati della gestione della memoria come questione distinta dalla gestione delle altre risorse. Nella pratica la gestione di una singola risorsa non può essere sempre disgiunta dai fattori relativi alle altre risorse, ma per ora rimandiamo al capitolo 8 la trattazione completa dell'allocazione delle risorse.

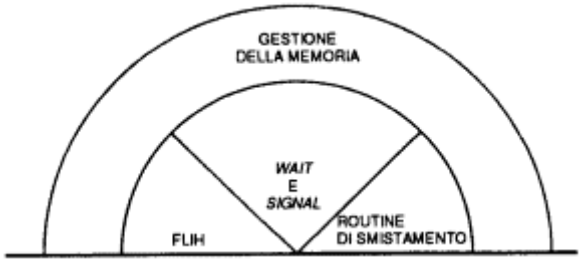
5.6 IMPLEMENTAZIONE NEL NOSTRO SISTEMA IPOTETICO

Il lettore potrebbe nutrire l'impressione che i metodi di gestione della memoria esaminati in questo capitolo presentino un aspetto troppo specialistico e che la scelta di un algoritmo per ogni caso specifico risulti in un certo senso arbitraria. In certa misura questo è vero; molti dei metodi vengono a delinearsi tramite esperienza e buon senso che, in alcuni casi, si basano su motivazioni di tipo analitico e statistico.
Uno strumento d'ausilio formidabile per la scelta degli algoritmi è costituito dalla simulazione; da un punto di vista ideale, ogni possibile scelta andrebbe verificata con una simulazione della combinazione dei lavori prevista. A tale proposito, alcuni studi vengono riportati da Knuth (1968), mentre Denning (1970) ha presentato una bibliografia esauriente. Al fine di applicare nel nostro sistema operativo ipotetico i meccanismi visti nel paragrafo 5.3, aggiungiamo quanto segue all'ambiente volatile di ciascun processo:

- una copia del contenuto del registro limite e del registro base oppure
- un puntatore alla sua segment table oppure
- un puntatore alla sua page table a seconda dell'architettura della macchina a nostra disposizione.

Se disponiamo di una macchina senza paginazione, introdurremo anche la hole list quale struttura dati a cui la tabella centrale è collegata tramite puntatore. Lo strato relativo alla gestione della memoria nel nostro sistema è costituito da un codice per l'implementazione dei metodi presi in esame nel paragrafo 5.4.
A questo punto il nostro sistema ha raggiunto lo stadio di sviluppo illustrato in figura.

Figura 43 - Stato attuale del sistema operativo ipotetico



6 INPUT E OUTPUT

6.1 Considerazioni generali

Giunti a questo punto dello sviluppo del nostro sistema operativo ipotetico, abbiamo delineato un ambiente nel quale possono esistere i processi e la memoria può venire allocata per contenerne i dati ed i programmi relativi. Ora volgiamo la nostra attenzione alle modalità di comunicazione dei processi con il mondo esterno, cioè a quei meccanismi per effettuare l'input e l'output delle informazioni.
Una delle funzioni principali di un sistema operativo, infatti, è il controllo di tutti i dispositivi di I/O del computer. Il sistema deve comandare i dispositivi, intercettare le interruzioni e gestire gli errori. Deve anche fornire un'interfaccia semplice e di facile uso tra i dispositivi e il resto del sistema. Per quanto possibile, l'interfaccia dovrebbe essere la stessa per tutti i dispositivi (indipendenza dal dispositivo). Il codice di I/O rappresenta una frazione significativa di tutto il sistema operativo. Argomento di questo capitolo è appunto il modo in cui il sistema operativo gestisce l'I/O.
Il motivo deriva dall'ampiezza della gamma delle periferiche impiegate: una configurazione particolare potrebbe comprenderne di molto diverse relativamente alle caratteristiche ed ai modi di funzionamento. Più precisamente, le periferiche si possono differenziare per uno o più dei seguenti aspetti.

(1) Velocità
Per varie periferiche vi può essere una differenza di svariati ordini di grandezza tra le velocità di trasferimento dei dati. Un disco magnetico, ad esempio, può essere in grado di trasferire 106 caratteri al secondo, contrariamente alla velocità della tastiera di un terminale pari a solo pochi caratteri al secondo (in funzione della bravura del dattilografo, ovviamente).

(2) Unità di trasferimento
In base alla periferica utilizzata, i dati si possono trasferire in unità di informazione corrispondenti a caratteri, word, byte, blocchi o record.

(3) Rappresentazione dei dati
Un singolo dato può essere codificato in modi diversi e su diversi supporti di informazione. Anche entro un singolo supporto, quale un nastro magnetico, si possono usare codici differenti. Le periferiche differiscono per il tipo di operazione che sono in grado di svolgere. Un esempio è dato dall'ovvia distinzione tra input ed output; un altro consiste nella capacità di riavvolgere il nastro magnetico, ma non la carta di una stampante parallela.

(5) Situazioni d'errore
Il mancato completamento di un trasferimento dati può essere determinato da varie cause, quali un errore di parità, l'inceppamento di schede o un errore in una somma di controllo, a seconda della periferica in uso.

E' chiaramente difficile gestire in modo uniforme le diversità esemplificate in precedenza. Comunque in questo capitolo tenteremo di costruire la struttura portante per un sistema di I/O nel quale vengano separate, per quanto possibile, le caratteristiche dipendenti dalla particolarità dell'unità periferica utilizzata ed in cui si riesca ad ottenere un certo livello di uniformità. Cominciamo col prendere in considerazione alcuni obiettivi del progetto e le relative implicazioni.
La struttura del capitolo è la seguente. Prima daremo un breve sguardo ad alcuni dei principi dell'hardware di I/O, quindi tratteremo il software di I/O in generale. Il software di I/O può essere strutturato in livelli, ognuno dei quali ha un ben definito compito da portare a termine. Tratteremo questi livelli per vedere cosa fanno e come si combinano insieme.

6.1.1 Dispositivi di I/O

L'hardware di I/O viene visto in modo diverso da persone diverse. Gli ingegneri elettronici lo trattano in termini di circuiti integrati, cavi, alimentatori, motori e tutti gli altri componenti fisici contenuti in esso. I programmatori vedono l'interfaccia verso il software, i comandi accettati dall'hardware, le funzioni che esso fornisce e gli errori che possono essere generati. In questo libro trattiamo la programmazione dei dispositivi di I/O, non la loro progettazione, costruzione o manutenzione; i nostri interessi resteranno pertanto circoscritti a come si programma l'hardware, non al suo funzionamento interno. Ciononostante, la programmazione di molti dispositivi di I/O è spesso intimamente collegata con il loro funzionamento interno. Nelle prossime tre sezioni forniremo un po' di nozioni di carattere generale sull'hardware di I/O e su come si colleghi alla programmazione.
I dispositivi di I/O possono essere grossolanamente suddivisi in due categorie: dispositivi a blocchi e dispositivi a caratteri. Un dispositivo a blocchi registra le informazioni in blocchi di dimensioni fisse, ognuno con un suo indirizzo. Le dimensioni del blocco vanno di solito da 512 a 32768 byte. La proprietà fondamentale di un dispositivo a blocchi è che ogni blocco può essere letto o scritto indipendentemente dagli altri. I dischi sono i dispositivi a blocchi più comuni.
Ma a ben vedere, il confine tra i dispositivi indirizzabili a blocchi e quelli che non lo sono non è così ben definito. Tutti sono d'accordo nel considerare il disco un dispositivo indirizzabile a blocchi perché, indipendentemente dalla posizione corrente del braccio, è sempre possibile spostarsi su un altro cilindro e attendere quindi che il blocco desiderato passi sotto la testina. Consideriamo ora un lettore di nastri 8 mm o DAT utilizzato per effettuare il backup del disco. Il nastro contiene in genere blocchi di dimensione fissa. Se si comanda al lettore di nastri la lettura del blocco N, esso può riavvolgere il nastro e andare avanti fino a che non arriva al blocco N. Quest'operazione è analoga a una ricerca sul disco, eccetto che il tempo necessario per completarla è di gran lunga superiore. Inoltre, non è detto che sia possibile riscrivere un blocco nel mezzo del nastro. Anche se fosse possibile utilizzare i nastri come dispositivi con accesso casuale al blocco, questa rappresentazione sarebbe in qualche modo una forzatura: le unità a nastro non vengono normalmente usate in questo modo.
L'altro tipo di dispositivi di I/O sono quelli a caratteri. Un dispositivo orientato al carattere fornisce o accetta una sequenza di caratteri senza curarsi della strutturazione a blocchi. Un dispositivo di questo tipo non è indirizzabile e non ha alcuna operazione di posizionamento (seek). Le stampanti, le periferiche di rete, i mouse nonché molti altri dispositivi che non somigliano affatto ai dischi possono essere visti come dispositivi a caratteri.
Questo schema di classificazione non è perfetto. Alcuni dispositivi ne rimangono infatti esclusi. Gli orologi di sistema, ad esempio, non sono indirizzabili a blocchi, né generano o accettano sequenze di caratteri. Essi non fanno altro che causare interruzioni a intervalli di tempo ben definiti. Anche i display mappati in memoria non rientrano in questo modello. Ciononostante, il modello di dispositivi a blocchi e dispositivi a caratteri è abbastanza generale da poter essere usato come base concettuale per far sì che la parte del software di sistema operativo che tratta l'I/O sia indipendente dal dispositivo.

Il file system, ad esempio, tratta unicamente i dispositivi astratti a blocchi e lascia al software di livello inferiore, chiamato device driver (gestore di periferica), la parte dipendente dal dispositivo.

6.1.2 Controllori di dispositivo

Le unità di I/O sono in genere costituite da una componente meccanica e una componente elettronica. È spesso possibile separare le due componenti per ottenere un progetto più modulare e generale. La parte elettronica viene chiamata controllore del dispositivo o adattatore (adapter). Sui personal computer questa prende spesso la forma di una scheda a circuito stampato che può essere inserita in una delle connessioni presenti sulla parentboard del computer (chiamata in precedenza con il nome improprio di motherboard). La parte meccanica è il dispositivo stesso.
La scheda del controllore ospita di solito un connettore, nel quale si può inserire un cavo che porta al dispositivo stesso. Molti controllori possono gestire due, quattro o perfino otto dispositivi identici. Se l'interfaccia tra il controllore e il dispositivo è un'interfaccia standard, sia esso uno standard ufficiale come ANSI, IEEE o ISO o uno standard di fatto, le ditte possono costruire controllori o dispositivi che rispettino tale interfaccia. Molte ditte, ad esempio, costruiscono dispositivi per dischi che rispettano le interfacce per controllori IDE (Integrated Drive Electronics) o SCSI (Small Computer System Interface).
Menzioniamo questa differenza tra controllore e dispositivo in quanto il sistema operativo ha quasi sempre a che fare con il controllore e non con il dispositivo. Molti piccoli computer usano il modello a bus singolo per mettere in comunicazione la CPU e i controllori. I grandi mainframe utilizzano spesso un modello diverso, con bus multipli e computer specializzati per l'I/O chiamati I/O channels (canali di I/O), che risparmiano alla CPU principale una parte del carico computazionale.
L'interfaccia tra il controllore e il dispositivo è spesso di livello molto basso. Un disco, ad esempio, può essere strutturato con un formato di 16 settori di 512 byte per traccia. Ciò che in realtà esce dal dispositivo è una sequenza seriale di bit che inizia con un preambolo, prosegue con i 4096 bit per settore e contiene infine un codice di correzione d'errore o checksum (ECC o Error Correcting Code). Il preambolo viene scritto quando viene definito il formato del disco e contiene l'indice di cilindro e di settore, la dimensione del settore e dati analoghi, insieme alle informazioni di sincronizzazione.
Il compito del controllore è di convertire la sequenza seriale di bit in un blocco di byte ed effettuare un'eventuale correzione d'errore. Di solito il blocco dei byte viene inizialmente assemblato, un bit alla volta, in un buffer all'interno del controllore. Dopo che il suo checksum è stato controllato e il blocco è stato dichiarato libero da errori, può essere copiato nella memoria principale.
Anche il controllore di un terminale video funziona a basso livello come un dispositivo seriale a bit. Il controllore legge dalla memoria i byte contenenti i caratteri da mostrare e genera i segnali usati per modulare il raggio del tubo catodico affinché scriva sullo schermo. Il controllore genera anche i segnali che comandano il ritracciamento orizzontale del raggio catodico dopo che la scansione di una riga è terminata, e i segnali per effettuare il ritracciamento verticale dopo che l'intero quadro è stato scandito. Se non fosse per il controllore dello schermo, il programmatore del sistema operativo dovrebbe programmare esplicitamente la scansione analogica dello schermo. In presenza del controllore, il sistema operativo inizializza il controllore con alcuni parametri, come il numero di caratteri per linea e il numero di linee contenute nello schermo, e lascia che il controllore si prenda cura di controllare esplicitamente il raggio di elettroni.
Ogni controllore usa un certo numero di registri per comunicare con la CPU. Su alcuni computer questi registri fanno parte del normale spazio di indirizzamento della memoria. Questo schema viene chiamato memory mapped I/O (I/O allocato nello spazio di memoria). Il 68000, ad esempio, usa questo metodo. Altri computer usano uno spazio di indirizzamento speciale per l'I/O, e allocano a ogni controllore una determinata porzione di tale spazio. L'assegnamento degli indirizzi di I/O ai dispositivi viene fatto mediante la logica di decodifica degli indirizzi del bus, compresa all'interno del controllore. Alcuni produttori dei cosiddetti PC compatibili IBM usano indirizzi diversi da quelli utilizzati da IBM. In aggiunta alle porte di I/O, molti controllori utilizzano le interruzioni per far sapere alla CPU quando sono pronti per la lettura o la scrittura dei propri registri. Un'interruzione è, in primo luogo, un evento elettrico. Una linea hardware di richiesta di interruzione (IRQ) corrisponde a un ingresso fisico nel circuito controllore delle interruzioni. Il numero di tali ingressi è limitato; i PC della categoria Pentium ne hanno a disposizione solo 15 per tutti i dispositivi di I/O. Alcuni controllori sono integrati fisicamente nella parentboard di sistema, come ad esempio il controllore della tastiera di un PC IBM. Nel caso il controllore di un dispositivo sia invece inserito nel backplane, si possono talvolta usare interruttori o ponticelli per selezionare quale linea di IRQ sarà utilizzata dal dispositivo e per evitare conflitti (benché in alcune schede, come nelle Plug & Play, gli IRQ possano essere impostati via software). Il controllore delle interruzioni trasforma ogni ingresso IRQ in un vettore di interruzione che individua il corrispondente software di servizio. La figura 46 mostra, ad esempio, gli indirizzi di I/O, le interruzioni hardware e i vettori di interruzione allocati ad alcuni dei controllori su un PC IBM.

??? Figura 44Figura 45

Figura 46 - Alcuni esempi di controllori, con i loro indirizzi di I/O, le loro linee fisiche di interruzione e i loro vettori di interruzione su un tipico PC IBM con MS-DOS

Controllore I/O	Indirizzo I/O	IRQ hardware	Vettore di interruzione
Clock	040—043	0	8
Tastiera	060—063	1	9
Harddisk	1F0—1F7	14	118
RS.232 secondaria	2F8—2FF	3	11
Stampante	378—37F	7	15
Floppy disk	3F0— 3F7	6	14
RS-232 primaria	3F8—3FF	4	12

Il sistema operativo effettua l'I/O scrivendo comandi nei registri del controllore. Il controllore dei floppy disk del PC IBM, ad esempio, accetta 15 comandi diversi come READ, WRITE, SEEK, FORMAT e RECALLBRATE. Molti dì questi comandi hanno dei parametri che vengono a loro volta caricati nei registri del controllore. Quando un comando è stato accettato, la CPU lascia che il controllore faccia il suo lavoro e passa a occuparsi d'altro. Quando il comando è stato completato, il controllore genera un'interruzione per permettere al sistema operativo di riprendere il controllo della CPU e controllare i risultati dell'operazione. La CPU ottiene i risultati e lo stato del dispositivo leggendo uno o più byte di informazione dai registri del controllore.

REGISTRI	FUNZIONE IN LETTURA	FUNZIONE IN SCRITTURA
0	dati	dati
1	errore	precompensazione in lettura
2	numero di settori	numero di settori
3	indice di settore (0-7)	indice di settore (0-7)
4	cilindro (parte inferiore, 8-15)	cilindro (parte inferiore, 8-15)
5	cilindro (parte superiore, 16-23)	cilindro (parte superiore, 16-23)
6	seleziona drive/testina (24-27)	seleziona drive/testina (24-27)
7	stato	comando

I numeri tra parentesi sono i bit dell'indirizzo logico di blocco selezionato da ogni registro in modalità LBA.

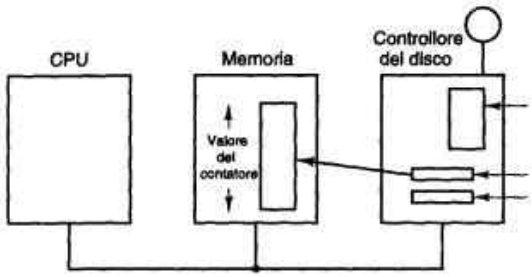
7	6	5	4	3	2	1	0
1	LBA	1	D	HS3	HS2	HS1	HS0

Valori: LBA = 0 Modalità cilindro/testina/settore, 1 Modalità di indirizzamento Blocco Logico
D = 0 drive master, 1 drive slave
i quattro bit HSn indicano: in modalità C/T/S la testina, in modalità Blocco Logico i bit 24-27 selezionano il blocco.

6.1.3 Accesso diretto in memoria (DMA)

Molti controllori, specialmente quelli per i dispositivi orientati al blocco, supportano l'accesso diretto in memoria o DMA (Direct Memory Access). Per spiegare come funziona il DMA diamo prima uno sguardo a come si effettuano le letture dal disco quando il DMA non viene usato. Innanzitutto il controllore legge il blocco (uno o più settori) dal dispositivo in modo seriale, bit a bit, fino a che l'intero blocco non è nel suo buffer interno. In seguito, esso calcola il checksum per verificare che non vi siano stati errori in lettura. Quindi il controllore genera un'interruzione. Quando il sistema operativo va in esecuzione, esso può leggere il blocco di disco dal buffer del controllore, un byte o una parola alla volta, eseguendo un ciclo in cui a ogni iterazione un byte o una parola vengono letti da un registro del controllore del dispositivo e immagazzinati in memoria.
Naturalmente, programmare un ciclo sulla CPU che legga uno alla volta i byte dal controllore significa sprecare tempo di CPU. Il DMA è stato inventato per liberare la CPU da questi compiti di basso livello. Quando si usa il DMA, la CPU fornisce al controllore l'indirizzo sul disco del blocco e due ulteriori informazioni: l'indirizzo di memoria in cui mettere il blocco, e il numero di byte da trasferire, come mostrato in figura 47.
Dopo aver letto l'intero blocco dal dispositivo nel suo buffer e verificato il checksum, il controllore copia il primo byte o la prima parola nella memoria principale, all'indirizzo specificato dall'indirizzo di memoria interno al DMA. Quindi incrementa l'indirizzo nel DMA e decrementa il contatore del DMA del numero di byte appena trasferiti. Questo procedimento viene ripetuto fino a che il contatore del DMA diventa zero. A questo punto il controllore genera un'interruzione. Quando il sistema operativo prende il controllo della CPU, non ha più bisogno di copiare il blocco in memoria: esso è stato già trasferito.

Figura 47 - I trasferimenti in DMA vengono interamente gestiti dal controllore



Ci si potrebbe chiedere perché il controllore non memorizzi i byte direttamente in memoria principale non appena li riceve dal disco. In altre parole, perché abbia bisogno di un buffer interno. Il motivo è che, una volta che un trasferimento da disco è stato avviato, i bit arrivano dal disco a un ritmo costante, che il controllore sia pronto a riceverli o no. Se il controllore cercasse di scrivere i dati direttamente in memoria, dovrebbe passare attraverso il bus di sistema ogni volta che intende trasferire una parola. Se il bus fosse occupato perché utilizzato da qualche altro dispositivo, il controllore sarebbe costretto ad attendere. Se la successiva parola arrivasse dal disco prima che la precedente fosse stata trasferita, il controllore dovrebbe memorizzarla da qualche parte. Se il bus fosse molto occupato, il controllore finirebbe per memorizzare un gran numero di parole e dovrebbe allo stesso tempo svolgere un oneroso lavoro di gestione. Quando si memorizza il blocco in un buffer interno, il buffer non viene richiesto fino a che non si avvia l'accesso in DMA; di conseguenza il progetto del controllore è più semplice, dato che i trasferimenti dal DMA alla memoria non hanno vincoli temporali (alcuni vecchi controllori andavano in effetti a scrivere direttamente in memoria e si limitavano a gestire un piccolo buffer interno, ma quando il bus risultava molto occupato, un trasferimento si poteva concludere con un errore di overrun).
Il meccanismo appena descritto di gestione dei buffer in due tempi ha importanti implicazioni sulle prestazioni dell'I/O. Mentre i dati vengono trasferiti dal controllore alla memoria, sotto il controllo della CPU del controllore, il settore seguente

passa sotto la testina e i bit arrivano al controllore. I controllori più semplici non riescono a gestire l'ingresso e l'uscita simultanea dei dati, così, mentre ha luogo un trasferimento in memoria, il settore che passa sotto la testina viene perduto. Come risultato, il controllore sarà capace di leggere solo un blocco ogni due. La lettura di un'intera traccia comporterà due rotazioni, una per i blocchi pari e una per i blocchi dispari. Se il tempo necessario a trasferire un blocco dal controllore alla memoria attraverso il bus risultasse maggiore del tempo necessario a leggere un blocco dal disco, potrebbe essere necessario leggere un blocco e saltarne quindi due (o più).
La tecnica che prevede di saltare alcuni blocchi per dare tempo al controllore di trasferire i dati in memoria viene chiamata interleaving.
Non tutti i computer usano il DMA. Le motivazioni a sfavore del DMA sono che la CPU è spesso molto più veloce del controllore del DMA e può svolgere il suo lavoro molto più velocemente (quando il fattore limitante non è la velocità del dispositivo di I/O). Se non ha nient'altro da fare, non ha senso che la CPU (veloce) attenda che il controllore del DMA (lento) finisca. Inoltre, eliminando il controllore del DMA e lasciando che la CPU faccia tutto il lavoro in software si risparmiano soldi.

6.1.4 Hardware dei dischi

Tutti i dischi reali sono organizzati in cilindri, ognuno contenente tante tracce quante sono le testine allineate in verticale. Le tracce sono divise in settori, e il numero dei settori posizionati intorno alla circonferenza è tipicamente compreso tra 8 e 32 per i floppy disk, mentre arriva a diverse centinaia su alcuni hard disk. I modelli più semplici hanno lo stesso numero di settori su ogni traccia. Tutti i settori contengono lo stesso numero di byte, benché a pensarci bene si capisca come i settori più vicini al bordo esterno del disco saranno fisicamente più lunghi di quelli vicini all'asse. In ogni caso, il tempo necessario a leggere o scrivere ogni settore sarà lo stesso. La densità dei dati è ovviamente più alta sui cilindri più interni, e alcuni modelli di disco richiedono un cambiamento nella corrente di comando per le testine di lettura-scrittura quando si trovano a operare sulle tracce più interne. Questa geometria viene gestita dall'hardware del controllore del disco e non è visibile all'utente (o al realizzatore di un sistema operativo).
La differenza nella densità dei dati tra le tracce interne ed esterne comporta un sacrificio in termini di capacità, benché esistano sistemi di gestione più sofisticati. Sono stati realizzati alcuni modelli di floppy disk che ruotano a velocità più alte quando le testine sono sopra le tracce esterne. Questo metodo consente di ottenere più settori nelle tracce esterne, aumentando la capacità del disco. I moderni hard disk di grandi capacità contengono più settori nelle tracce esterne rispetto alle tracce interne. Di questo tipo sono i drive IDE (Integrated Drive Electronics), e le sofisticate elaborazioni effettuate dall'elettronica interna al drive ne mascherano i dettagli realizzativi. Essi appaiono al sistema operativo come se avessero una geometria semplice con lo stesso numero di settori per ogni traccia.
L'elettronica del drive e del controllore è altrettanto importante quanto l'hardware meccanico. L'elemento principale della scheda del controllore inserita nel backplane del computer è un circuito integrato specializzato, in effetti un piccolo microcomputer. La circuiteria della scheda controllore di un hard disk può essere più semplice di quella di un floppy disk, ma questo accade perché i drive degli hard disk hanno un potente controllore elettronico al loro interno. Una caratteristica del dispositivo con implicazioni importanti per il driver del disco è la possibilità per controllore di effettuare posizionamenti (seek) su due o più drive allo stesso tempo. Queste operazioni sono note come posizionamenti sovrapposti (overlapped seeks). Mentre il controllore e il software stanno aspettando il completamento di un posizionamento su un drive, il controllore può avviare un posizionamento su un altro drive. Molti controllori possono anche leggere o scrivere su un drive mentre effettuano posizionamenti su uno o più altri drive, ma un controllore di floppy disk non può leggere da o scrivere su due drive nello stesso istante (le letture e le scritture richiedono che il controllore sposti i bit in intervalli di pochi microsecondi, quindi ogni trasferimento usa la maggior parte del suo potere computazionale). La situazione è diversa per gli hard disk con controllori integrati, e in un sistema con più drive di questo tipo i controllori possono operare simultaneamente, almeno per quel che riguarda i trasferimenti tra il disco e la memoria di buffer del controllore. In ogni caso, è possibile un solo trasferimento alla volta tra il controllore e la memoria di sistema. La possibilità di effettuare due o più operazioni allo stesso tempo può ridurre in modo considerevole il tempo medio di accesso.
La figura 49 mette a confronto i parametri dei floppy doppia-faccia, doppia-densità, ovvero il dispositivo di memoria standard per i PC IBM, con i parametri di un tipico drive hard disk di media capacità, come quelli che si possono trovare sui computer basati sul Pentium.
Quando si leggono le specifiche dei moderni hard disk bisogna fare attenzione al fatto che la geometria specificata e usata dal software del driver può essere differente dal formato fisico. L'hard disk descritto in figura 49 è ad esempio specificato con “parametri di configurazione raccomandati” di 1048 cilindri, 16 testine e 63 settori per traccia. L'elettronica del controllore montata sul disco converte i parametri logici di testina e settore forniti dal sistema operativo in quelli fisici utilizzati dal disco. Questo è un ulteriore esempio di un compromesso studiato per mantenere la compatibilità con i sistemi precedenti, in questo caso il vecchio firmware. I progettisti del PC IBM originario riservarono nella ROM del BIOS un campo dì soli 6 bit per il contatore dì settore.

Figura 48

Figura 49 - I parametri del disco per i floppy da 360KB del PC IBM originario e per un hard disk Western Digital WD AC2540 da 540MB.

Parametro	IBM 360-KB floppy disk	WD 540-MB hard disk
Numero di cilindri	40	1048
Tracce per cilindro	2	4
Settori nella traccia	9	252
Settori nel disco	720	1056384
Byte per settore	512	512
Byte nel disco	368640	540868608
Tempo di seek(cilindri adiacenti)	6 msec	4 msec
Tempo di seek(caso medio)	77 msec	11 msec
Tempo di rotazione	200 msec	13 msec
Tempo per avviare/fermare il motore	250 msec	9 sec
Tempo per trasferire i settore	22 msec	53 microsec

Un disco che abbia più di 63 settori fisici per traccia deve funzionare con un insieme artificiale di parametri logici di disco. In questo caso, le specifiche del costruttore indicano che ci sono in realtà quattro testine, e pertanto sembrerebbe che ci siano 252 settori per traccia, come indicato in figura. Questa è però un'ulteriore semplificazione, perché i dischi di questo tipo contengono più settori nelle tracce esterne di quanti ne abbiano nelle tracce interne. Il disco descritto in figura ha in effetti quattro testine, ma contiene in realtà un po' più di 3000 cilindri. I cilindri sono raggruppati in una dozzina di zone che hanno da un minimo di 57 settori per traccia nelle zone più interne a un massimo di 105 nei cilindri più esterni. Questi numeri non si trovano nelle specifiche del disco e le traduzioni effettuate dall'elettronica del drive consentono di fare a meno di questi dettagli.

6.1.5 Software del disco

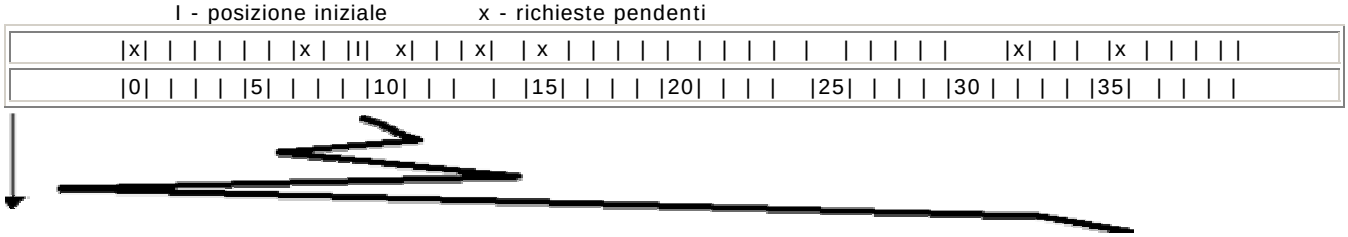
In questa sezione tratteremo alcuni argomenti relativi ai drive di disco in generale. Consideriamo innanzitutto il tempo necessario a leggere o scrivere un blocco sul disco. Il tempo necessario è determinato da tre fattori:

1. Il tempo di posizionamento (il tempo necessario a spostare il braccio che sostiene la testina sul cilindro giusto).
2. Il ritardo di rotazione (il tempo necessario affinché il settore richiesto, ruotando, giunga sotto la testina).

Ad esempio, se dopo essere arrivati al cilindro 16, arriva una nuova richiesta per il cilindro 8, avrà priorità su quella per il cilindro 1. Se a questo punto arriva una richiesta per il cilindro 13, il braccio si sposterà sul cilindro 13 anziché sull'i. Se il disco viene utilizzato molto, il braccio tenderà a restare per la maggior parte del tempo nel suo mezzo, e le richieste relative a entrambi gli estremi dovranno pertanto attendere fino a che una fluttuazione statistica nel carico non faccia sì che non vi siano più richieste per i cilindri interni. Le richieste lontane dal centro riceverebbero un pessimo servizio. Gli obiettivi di tempo di risposta minimo ed equità sono in questo caso in conflitto. Anche i palazzi alti hanno a che fare con questo compromesso. Il problema di pianificare i movimenti di un ascensore in un palazzo è simile al problema di scheduling per il braccio di un disco. Le richieste arrivano in continuazione, chiamando a caso l'ascensore sui piani dell'edificio (cilindri). Il microprocessore che controlla l'ascensore potrebbe facilmente registrare la sequenza in cui i clienti hanno premuto il tasto di chiamata e servirli usando una politica FCFS, così come potrebbe usare la SSF.

Figura 50 - L'algoritmo di scheduling SSF

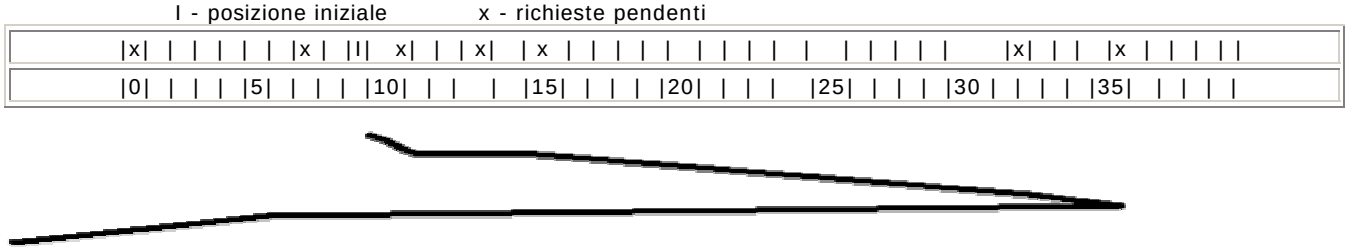
I - posizione iniziale x - richieste pendenti



La maggior parte degli ascensori usa invece un algoritmo diverso per conciliare i diversi obiettivi di efficienza ed equità. Gli ascensori continuano a muoversi nella stessa direzione fino a che non vi sono più richieste, quindi cambiano direzione. Questo algoritmo, noto nel mondo dei dischi (e in quello degli ascensori) come l'algoritmo dell'ascensore, richiede al software di gestire un bit: il bit della direzione corrente, UP o DOWN. Quando una richiesta ha termine, il driver del disco (o dell'ascensore) controlla il bit. Se il bit vale UP, il braccio (o la cabina) viene spostato alla richiesta pendente immediatamente superiore. Se non vi sono richieste pendenti per una posizione superiore, il bit di direzione viene rovesciato. Quando il bit è posto a DOWN, il movimento è diretto verso la posizione immediatamente inferiore, se ve ne sono.

Figura 51 - L'algoritmo dell'ascensore

I - posizione iniziale x - richieste pendenti



La figura 51 mostra l'algoritmo dell'ascensore usando le stesse sette richieste di figura 50 e assumendo che il bit di direzione sia inizialmente UP. L'ordine in cui i cilindri sono serviti è 12, 16, 34, 36, 9 e 1, il che porta a movimenti del braccio dii, 4, 18, 2, 27 e 8 cilindri, per un totale di 60 cilindri. In questo caso l'algoritmo dell'ascensore è lievemente migliore di quello SSF, benché si riveli di solito peggiore. Una piacevole proprietà dell'algoritmo dell'ascensore è che, dato un qualsiasi insieme di richieste, il limite superiore degli spostamenti complessivi è fissato in due volte il numero dei cilindri.

Una leggera variante di questo algoritmo che comporta una minore varianza nei tempi di risposta (Teory, 1972) consiste nello scandire il disco sempre nella stessa direzione. Quando è stata servita la richiesta pendente sul cilindro di indice massimo, il braccio si sposta sul cilindro di indice minimo per il quale esiste una richiesta pendente e continua poi a spostarsi verso l'alto. In effetti, è come se il cilindro minimo venisse pensato immediatamente al di sopra del cilindro massimo.

Alcuni controllori di disco forniscono al software un modo per controllare l'indice del settore attualmente sotto la testina. Con un simile controllore è possibile un'ulteriore ottimizzazione. Se ci sono due o più richieste pendenti per lo stesso cilindro, il driver può inoltrare una richiesta per il settore che passerà per primo sotto la testina. Si noti che, quando ci sono più tracce in un cilindro, è possibile servire più richieste consecutive per tracce diverse senza alcuna penalità. Il controllore può selezionare istantaneamente una delle sue testine, dal momento che la selezione delle testine non implica alcun movimento del braccio né un ritardo di rotazione.

Nei moderni hard disk il ritmo di trasferimento dei dati è tanto più veloce di quello dei floppy disk da richiedere un qualche meccanismo automatico di caching. Solitamente qualsiasi richiesta di lettura di un settore provocherà la lettura di tale settore, così come di tutto il resto della traccia corrente a seconda dello spazio disponibile nella memoria di cache del controllore. Il disco da 540M descritto in figura 50 ha una cache di 64K o 128K. L'uso della cache è determinato dinamicamente dal controllore. Nella modalità più semplice, la cache è divisa in due sezioni, una per le letture e una per le scritture.

Quando ci sono più dispositivi, una tabella delle richieste pendenti dovrebbe essere gestita in modo separato per ogni dispositivo. Quando un drive è inattivo, si dovrebbe impostare una ricerca per muovere il suo braccio verso il cilindro in cui esso sarà successivamente richiesto (assumendo che il controllore consenta ricerche sovrapposte). Quando il trasferimento corrente termina, si può effettuare un controllo per verificare se uno dei drive sia posizionato sul cilindro

corretto. Se uno o più lo sono, il prossimo trasferimento può essere iniziato su uno dei drive già posizionati sul cilindro corretto. Se nessuno dei bracci è nel posto giusto, il driver dovrebbe richiedere una nuova ricerca sul dispositivo che ha appena completato un trasferimento e aspettare la successiva interruzione per vedere quale braccio arrivi per primo a destinazione.

6.1.6 Gestione degli errori

I dischi RAM non devono preoccuparsi delle ottimizzazioni legate ai ritardi di ricerca o di rotazione: in un qualsiasi istante è possibile leggere o scrivere un qualsiasi blocco senza alcun movimento fisico. Un'altra area in cui i dischi RAM sono più semplici dei dischi reali è la gestione degli errori. I dischi RAM funzionano sempre; quelli reali non altrettanto, in quanto sono soggetti a una vasta gamma di errori. Alcuni dei più comuni sono:

1. Errori di programmazione (ad es. richieste per settori inesistenti).
2. Errori di checksum transitori (ad es. causati da polvere sulle testine).
3. Errori di checksum permanenti (ad es. un blocco sul disco fisicamente danneggiato)
4. Errori di ricerca (ad es. il braccio spostato al cilindro 6 ma arrivato al 7)
5. Errori del controllore (ad es. il controllore si rifiuta di accettare comandi).

Il driver del disco ha il compito di gestire tutti questi errori nel miglior modo possibile.
Gli errori di programmazione si verificano quando il driver chiede al controllore di ricercare un cilindro inesistente, leggere da un settore inesistente, usare una testina che non esiste o trasferire dati da o verso locazioni di memoria inesistenti. La maggior parte dei controllori verificano i parametri passati loro e si lamentano nel caso in cui non siano validi. In teoria, questi errori non dovrebbero mai accadere, ma cosa deve fare il driver se il controllore indica che se ne è verificato uno? Per un sistema fatto in casa, la miglior soluzione è fermarsi e stampare un messaggio del tipo “Chiamate il programmatore”, in modo che l'errore possa essere individuato e sistemato. Per un prodotto software commerciale in uso su migliaia di computer intorno al mondo questo approccio è meno attraente. Probabilmente la sola cosa da fare è terminare la richiesta corrente per il disco con un errore e sperare che esso non si ripeta troppo spesso.
Gli errori transitori di checksum sono provocati da granelli di polvere presenti nell'aria tra la testina e la superficie del disco. Nella maggior parte dei casi questi errori possono essere eliminati ripetendo l'operazione alcune volte. Se l'errore persiste, il blocco deve essere marcato come difettoso (bad block) ed evitato. Un modo per evitare di usare i blocchi difettosi consiste nello scrivere un programma molto speciale che prenda in ingresso una lista di blocchi difettosi e confezioni attentamente un file contenente tutti i blocchi difettosi. Una volta che questo file è stato creato, l'allocatore del disco penserà che tali blocchi siano occupati e non li allocherà mai. Non vi saranno problemi fintanto ché nessuno cercherà di leggere il file dei blocchi difettosi. Purtroppo, non leggere il file dei blocchi difettosi è più facile a dirsi che a farsi. Molti dischi vengono periodicamente archiviati (mediante operazioni di backup) copiando il loro contenuto, una traccia alla volta, su un disco o nastro di backup.

6.2 Obiettivi del progetto ed implicazioni relative

6.2.1 Indipendenza dal codice dei caratteri

Ovviamente non è auspicabile che, per la stesura dei propri programmi, un utente debba disporre di conoscenze particolareggiate sui codici dei caratteri impiegati dalle varie periferiche. Il sistema di I/O deve assumersi il compito di riconoscere i diversi codici, presentando ai programmi utente i dati in forma codificata.

6.2.2 Indipendenza dal tipo di periferica

A questo proposito si presentano due aspetti. In primo luogo, un programma dovrebbe essere indipendente dalla particolare tipologia del modello di periferica a cui capita di essere destinato. Ad esempio, non dovrebbe risultare importante su quale unità a nastro magnetico è montato un nastro specifico o quale stampante viene utilizzata per l'output di un programma. Un'indipendenza di tal genere assicura di evitare fallimenti di programmi dovuti semplicemente alla rottura di una determinata periferica o alla sua destinazione ad altro compito. Concede, infatti, al sistema operativo la facoltà di assegnare una periferica di tipo idoneo a seconda delle disponibilità complessive presenti in un dato momento.
Strettamente collegato all'indipendenza dal tipo di periferica è l'obiettivo dell'uniformità dello spazio dei nomi. Il nome di un file o di un dispositivo dovrebbe essere semplicemente una stringa o un intero e non dipendere in alcun modo dal dispositivo.
In UNIX, tutti i dischi possono essere integrati arbitrariamente nella gerarchia del file system in modo che l'utente non abbia bisogno di sapere a quale nome corrisponda un determinato dispositivo. Per esempio, un floppy disk può essere montato sulla directory /usr/ast/backup in modo che il comando per copiare un file in /usr/ast/backup/monday copi il file all'interno del floppy disk. In questo modo tutti i file e i dispositivi sono indirizzati nello stesso modo: con un nome di percorso.
Per quanto concerne il secondo aspetto, più impegnativo del precedente, è auspicabile che i programmi siano indipendenti, per quanto possibile, dal tipo di unità utilizzata per il loro I/O. A tale riguardo non intendiamo, chiaramente, spingerci fino al punto di poter inviare l'output ad una tastiera; pensiamo, invece, che dovrebbe risultare necessario apportare ad un lavoro soltanto dei cambiamenti minimi per metterlo in grado, per esempio, di ricevere dati provenienti da disco anziché da nastro magnetico.
È il sistema operativo che si deve occupare dei problemi che nascono dall'effettiva diversità di questi dispositivi e dal fatto che i dispositivi richiedono gestori (device driver) profondamente diversi per scrivere fisicamente i dati

6.2.3 Efficienza

Poiché tutte le operazioni di I/O in un sistema di calcolo si rivelano spesso delle “strozzature” che provocano un calo nelle prestazioni del sistema stesso, è auspicabile svolgerle il più possibile in modo efficiente.

6.2.4 Uniformità di gestione delle periferiche

Per favorire la semplicità e l'assenza d'errori è auspicabile che tutte le periferiche siano gestite in maniera uniforme. Per le ragioni già citate, all'atto pratico questo può risultare difficile.

6.2.5 Gestione degli errori

Un altro argomento importante per il software di I/O è la gestione degli errori. In generale, gli errori dovrebbero essere gestiti per quanto possibile in prossimità dell'hardware. Se il controllore scopre un errore in lettura, dovrebbe cercare, se può, di correggerlo esso stesso. Se non ne ha la possibilità, è il gestore del dispositivo che deve allora prendersene cura,

magari semplicemente ritentando la lettura del blocco. Molti errori sono transitori, come ad esempio gli errori in lettura causati da granelli di polvere sulla testina, e scompaiono quando l'operazione viene ripetuta. I livelli superiori dovrebbero essere informati del problema solo se i livelli inferiori non sono in grado di affrontarlo. In molti casi, il recupero da un errore può essere fatto in modo trasparente a basso livello senza che i livelli superiori ne vengano a conoscenza.

6.2.6 Sincronizzazione

Un ulteriore argomento chiave è la separazione tra trasferimenti sincroni (bloccanti) e asincroni (gestiti a interruzione). La maggior parte dell'I/O fisico è asincrono — la CPU inizia il trasferimento e continua le sue elaborazioni fino a che non arriva un'interruzione. I programmi utente sono però molto più semplici da scrivere se le operazioni di I/O sono bloccanti - dopo un comando di lettura il programma viene automaticamente sospeso fino a che i dati non sono disponibili nel buffer. È il sistema operativo che deve far sembrare sincrone (bloccanti) ai programmi utente le operazioni che in realtà sono guidate dalle interruzioni, cioè asincrone.

6.2.7 Condivisione e non

L'ultimo concetto che tratteremo in questa sede è quello relativo ai dispositivi condivisi e a quelli dedicati. Alcuni dispositivi di I/O, come i dischi, possono essere usati da più utenti contemporaneamente. Non c'è nessun problema se più utenti aprono dei file sullo stesso disco nel medesimo istante. Altri dispositivi, come le unità a nastro, devono invece essere dedicati a un singolo utente fino a che quell'utente non ha finito. Solo alla fine un altro potrà ottenere l'unità a nastro. Chiaramente non è opportuno che due o più utenti scrivano sullo stesso nastro blocchi mischiati a caso. L'introduzione dei dispositivi dedicati (unshared) comporta tutta una serie di problemi. Il sistema operativo deve essere capace di gestire dispositivi sia condivisi che dedicati in modo da evitare problemi.
Alcune implicazioni relative a questi obiettivi appaiono evidenti sin d'ora. Per prima cosa, l'indipendenza dal codice dei caratteri implica l'esistenza di una rappresentazione interna, unificata per tutti i caratteri. Questa rappresentazione prende il nome di codice interno dei caratteri ed a ogni periferica si devono associare dei meccanismi di trascodifica per effettuare le opportune conversioni al momento dell'input o dell'output. Per quelle periferiche che possono gestire numerosi codici dei caratteri bisogna prevedere un meccanismo di trascodifica distinto per ogni codice supportato. La trascodifica viene effettuata subito dopo l'input ed appena prima dell'output, così da rendere necessaria la conoscenza del codice interno unificato esclusivamente a quei processi intimamente connessi con la gestione delle periferiche. In linea di massima, il meccanismo di trascodifica consisterà in una tavola o, per alcuni casi, in un breve segmento di programma.
Oltre a ciò, l'indipendenza dalle periferiche utilizzate comporta che i programmi debbano operare non sulle unità reali, bensì su unità virtuali, le quali sono variamente denominate: stream (Atlas), file speciali (UNIX) o data set (IBM). Questi flussi (stream) di dati vengono utilizzati all'interno dei programmi senza riferimento alcuno alle unità periferiche; il programmatore dirige semplicemente l'output e l'input verso determinati stream, i quali sono poi associati alle periferiche reali solitamente tramite il sistema operativo, sulla base delle informazioni fornite dall'utente nella descrizione del lavoro. Per i nostri scopi momentaneamente è sufficiente dire che la descrizione di un lavoro è una raccolta di informazioni fornite dall'utente, presentate dì solito all'inizio del lavoro, che servono al sistema operativo per decidere come e quando metterlo in esecuzione.
La terza implicazione relativa agli obiettivi del nostro progetto riguarda il fatto che si dovrebbe costruire il sistema di I/O in modo da associare chiaramente le caratteristiche delle periferiche alle unità stesse, anziché alle routine che le gestiscono (i gestori delle unità periferiche, driver). Così risulta possibile far sì che queste ultime presentino grosse somiglianze e che le loro differenze funzionali derivino esclusivamente da informazioni parametriche tratte dalle caratteristiche della particolare periferica interessata. La necessaria separazione delle caratteristiche delle periferiche si può realizzare codificandole in un descrittore di periferica collegato a ciascun dispositivo, servendosi quindi del descrittore quale fonte di informazioni per le routine di gestione delle periferiche. Le informazioni caratteristiche circa un'unità periferica che possono venir memorizzate nel descrittore relativo sono:

- l'identificazione dell'unità;
- le istruzioni che rendono operativa l'unità;
- i puntatori rivolti alle tavole di trascodifica dei caratteri;
- lo stato corrente (cioè, se l'unità è occupata, libera o rotta);
- processo utente corrente (un puntatore verso il descrittore di quell'eventuale processo che utilizza al momento la periferica).

Tutti i descrittori di periferica possono essere inseriti assieme in una lista delle periferiche, indirizzata tramite il puntatore della tabella centrale

6.2.8 Organizzare il software a livelli

L'idea di base è di organizzare il software come una sequenza di livelli in cui quelli inferiori si preoccupano di nascondere le peculiarità dell'hardware a quelli superiori, e quelli superiori si preoccupano di presentare agli utenti un'interfaccia piacevole, pulita e regolare. Nelle sezioni seguenti daremo uno sguardo a questi obiettivi e a come essi vengano raggiunti.
Questi obiettivi possono essere raggiunti in modo chiaro ed efficiente strutturando il software di I/O in quattro livelli:

1. Gestori di interruzione (in basso).
2. Device driver (gestori dei dispositivi).
3. Software di sistema operativo indipendente dal dispositivo.
4. Software di livello utente (in alto).

Nelle sezioni seguenti li prenderemo in considerazione i livelli 2, 3 e 4 e vedremo come siano collegati insieme. Precedentemente abbiamo parlato del livello 1.

6.2.9 Gestori di periferica

Tutto il codice dipendente dal dispositivo finisce nel gestore del dispositivo (o device driver). Ogni gestore controlla un tipo di dispositivo o, al più, una classe di dispositivi strettamente imparentati. Per esempio sarebbe probabilmente una buona idea avere un solo gestore del terminale anche se il sistema supportasse più marche, tutte leggermente differenti fra loro.
D'altro canto, un semplice terminale meccanico di tipo hardcopy, un terminale intelligente con grafica a bitmap e un mouse sono così differenti da richiedere gestori diversi. In precedenza, in questo capitolo, abbiamo visto cosa fanno i controllori di dispositivo. Abbiamo visto che ogni controllore dispone di uno o più registri di interfaccia usati per impartirgli comandi. Il gestore del dispositivo impartisce questi comandi e controlla che siano propriamente eseguiti. Pertanto, il gestore del disco è la sola parte del sistema operativo che sa quanti registri abbia quel particolare controllore del disco e a cosa essi servano. Solo il gestore è a conoscenza di settori, tracce, cilindri, testine, movimenti del braccio, fattori di

interleaving, controllo dei motori, tempi di posizionamento delle testine e di tutti gli altri meccanismi che fanno sì che il disco funzioni correttamente.
In termini generali, il compito di un gestore di dispositivo è accettare richieste astratte dal software indipendente dal dispositivo posto al di sopra di esso e far sì che queste richieste siano soddisfatte. Una richiesta tipica consiste nella lettura del blocco n. Se il gestore è libero nel momento in cui la richiesta arriva, inizia a eseguirla immediatamente. Se invece il gestore è già occupato, inserisce di norma la nuova richiesta in una coda di richieste pendenti da soddisfare non appena possibile.
Il primo passo nel soddisfare una richiesta di I/O, ad esempio per un disco, è di tradurla da termini astratti in termini concreti. Per un driver di disco questo significa calcolare la posizione fisica del blocco richiesto sul disco, controllare che il motore del drive stia girando, determinare se il braccio è posizionato sul cilindro giusto e così via. In breve, esso deve decidere quali operazioni richiedere al controllore e in quale sequenza.
Una volta che ha determinato quali comandi inoltrare al controllore, il driver inizia a richiederli scrivendo nei registri del controllore. Alcuni controllori possono gestire un solo comando alla volta, altri sono disponibili ad accettare una lista di comandi che poi eseguono per conto proprio, senza ulteriori interventi da parte del sistema operativo.
Dopo che il comando o i comandi sono stati inoltrati si possono verificare due situazioni. In molti casi il device driver deve attendere fino a che il controllore non soddisfa la richiesta, e quindi si blocca fino a che non arriva un'interruzione a sbloccarlo. In altri casi, invece, l'operazione termina senza alcun ritardo, e il driver non ha bisogno di bloccarsi. Come esempio della seconda situazione, su alcuni terminali per effettuare lo scorrimento dello schermo basta scrivere alcuni byte nei registri del controllore. Non è necessario alcun movimento meccanico e l'intera operazione può essere completata in pochi microsecondi.
Nel primo caso, il driver bloccato sarà sbloccato da un'interruzione. Nel secondo caso non verrà mai sospeso. In ogni caso, dopo il completamento dell'operazione il driver deve controllare gli eventuali errori. Se tutto è a posto, il driver in alcuni casi deve passare dei dati al software indipendente dal dispositivo (ad es. il blocco appena letto). Infine, il driver restituisce al chiamante alcune informazioni di stato usate per riportare eventuali errori. A questo punto, se ci sono altre richieste accodate, ne può scegliere una e avviarla. Se non c'è niente in coda, il driver si blocca in attesa della richiesta successiva.
La gestione degli errori viene effettuata in larga parte dai driver. La maggior parte degli errori è fortemente dipendente dal dispositivo, e solo il driver può sapere cosa fare (ad es. riprovare l'operazione, ignorare l'errore o andare in panic). Un tipico errore viene causato da un blocco danneggiato che non può più essere letto. Dopo che il driver ha provato a leggere il blocco un certo numero di volte, ci rinuncia e informa il software indipendente dal dispositivo. Se l'errore è avvenuto mentre si leggeva un file utente, può essere sufficiente riportare una segnalazione al chiamante. Viceversa, se l'errore ha avuto luogo mentre si leggeva una struttura dati critica per il sistema, come ad esempio un blocco contenente la bitmap che indica quali blocchi sono liberi, il sistema operativo può non avere altra scelta che stampare un messaggio di errore e quindi terminare.

6.2.10 Software di I/O indipendente dal dispositivo

Benché alcune porzioni del software di I/O siano specifiche del dispositivo, una larga parte di esso ne è indipendente. Il confine esatto tra il driver e il software indipendente dal dispositivo varia a seconda del sistema, in quanto alcune funzioni che magari sono svolte dal software indipendente dal dispositivo potrebbero in realtà essere effettuate dal driver, per motivi di efficienza o per altre ragioni. In questa sede daremo un breve sguardo al software indipendente dal dispositivo per dare una prospettiva dell'I/O e chiarire in quale contesto si inseriscono i driver.
Il compito principale del software indipendente dal dispositivo consiste nel realizzare le funzioni di I/O comuni a tutti i dispositivi e nel fornire un'interfaccia uniforme al software di livello utente. Un concetto importante di ogni sistema operativo è come assegnare nomi a oggetti come i file e i dispositivi di I/O. Il software indipendente dal dispositivo si preoccupa di mettere in corrispondenza i nomi simbolici dei dispositivi (usiamo qui il termine stream) con i driver appropriati.

Figura 52 - Informazioni sullo stream e sulla periferica per un processo

descrittore processo						
.....						
puntatore a lista stream	------->	stream 0	------->	stream 1	------->	stream 2
		filein		/dev/lpt00		Display

La corrispondenza tra stream e tipi di periferiche per un processo specifico si può registrare in una lista dei descrittori di stream cui fa riferimento il puntatore del relativo descrittore di processo (vedi figura 52); l'assegnazione di un'unità particolare di un determinato tipo viene effettuata nel momento in cui il processo utilizza per la prima volta lo stream corrispondente. Diciamo a questo punto che il processo apre lo stream, il quale può venir chiuso (intendendo che non va più usato) sia esplicitamente dal processo sia implicitamente al termine del medesimo. Relativamente al processo illustrato nella figura 52, sono stati assegnati per il flusso di output numero 1, che non è stato ancora aperto (#), si tratterà di una stampante parallela, l'unità nastro magnetico numero 1 quale stream di input numero 2, l'unità disco numero21 quale stream di input numero 3,
In UNIX un nome di dispositivo, come /dev/tty00, specifica in modo univoco l'i-node di un file speciale, e questo i-node contiene il major device number (numero di dispositivo principale) che viene utilizzato per locallzzare il driver appropriato. L'i-node contiene inoltre il minor device number (o numero di dispositivo secondario), che viene passato come parametro al driver per specificare l'unità su cui leggere o scrivere.
Strettamente collegata all'assegnazione dei nomi è la protezione. Come fa il sistema a proibire l'accesso ai dispositivi agli utenti non autorizzati? Nella maggior parte dei personal computer non esiste alcuna protezione. Ogni processo può fare ciò che vuole. Nella maggior parte dei mainframe, invece, l'accesso ai dispositivi di I/O è del tutto proibito ai processi utente. In UNIX si usa uno schema più flessibile. I file speciali corrispondenti ai dispositivi di I/O sono protetti dai soliti bit rwx. L'amministratore di sistema può quindi stabilire i permessi appropriati per ogni dispositivo.
Dischi diversi possono avere dimensioni di settore diverse. E compito del software indipendente dal dispositivo nascondere questi particolari e far sì che i livelli superiori vedano una dimensione di blocco uniforme, ad esempio trattando più settori

come un singolo blocco logico. Così facendo, i livelli superiori trattano solo con dispositivi astratti che utilizzano tutti la stessa dimensione di blocco logico, indipendentemente dalla dimensione fisica del settore. In modo simile, alcuni dispositivi a carattere consegnano i loro dati un byte alla volta (ad es. i modem), mentre altri li forniscono in unità di dimensione superiore (ad es. le interfacce di rete). Anche queste differenze devono essere nascoste.
La gestione dei buffer è un altro argomento di rilievo, sia per i dispositivi a blocchi che per quelli a caratteri. Nei dispositivi a blocchi, l'hardware generalmente richiede che si legga e scriva ogni volta un numero intero di blocchi, mentre i processi utente sono liberi di leggere e scrivere secondo unità arbitrarie. Se un processo utente scrive mezzo blocco, il sistema operativo normalmente conserverà i dati al suo interno fino a che non si scrivono i dati necessari a completare il blocco. Solo a questo punto il blocco potrà essere finalmente scritto sul disco. Per quel che riguarda i dispositivi a carattere, gli utenti possono scrivere dati verso il sistema più velocemente di quanto i dati stessi possano essere inoltrati in uscita, richiedendo quindi l'uso di buffer. Anche gli ingressi da tastiera che arrivano prima di essere effettivamente richiesti devono essere inseriti in un buffer.
Quando un file viene creato e riempito di dati, gli si devono allocare nuovi blocchi su disco. Per effettuare quest'allocazione il sistema operativo ha bisogno di una lista o di una tabella di bit (bitmap) relativa ai blocchi liberi, ma l'algoritmo per locallzzare un blocco libero è indipendente dal dispositivo e può essere eseguito al di sopra del livello del driver.
Alcuni dispositivi, come i masterizzatori di CD-ROM, possono essere usati da un solo processo alla volta. È compito del sistema operativo esaminare le richieste per l'uso del dispositivo e accettarle o respingerle, a seconda che il dispositivo richiesto sia o meno disponibile. Un modo semplice per gestire queste richieste è di imporre che i processi effettuino direttamente le OPEN sui file speciali dei dispositivi. Se il dispositivo non è disponibile, la OPEN fallirà. La chiusura di un simile dispositivo dedicato corrisponderà quindi al suo rilascio con l'operazione di CLOSE.

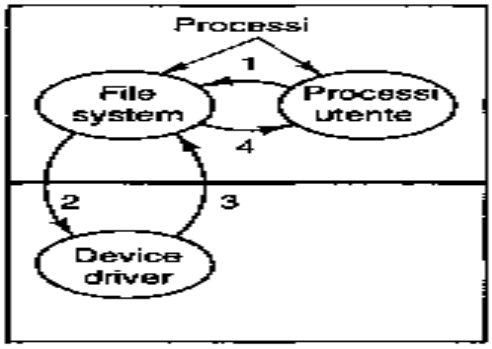
6.2.11 Software di I/O nello spazio utente

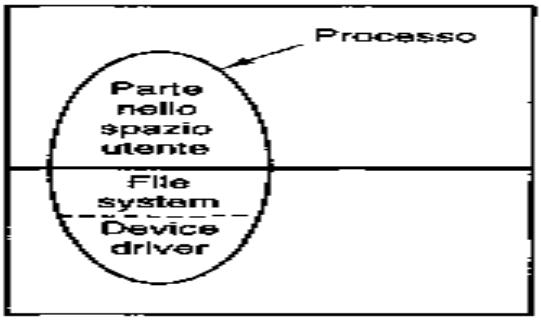
Benché la maggior parte del software di I/O sia contenuto nel sistema operativo, una piccola porzione consiste di librerie da collegare ai programmi utente e perfino di interi programmi che risiedono al di fuori del nucleo. Le chiamate di sistema, comprese le chiamate per l'I/O, sono normalmente costituite da procedure di libreria. Quando un programma C contiene la chiamata

```
count = write(fd, buffer, nbytes);
```

la procedura di libreria write sarà collegata con il programma e come tale farà parte del programma binario presente in memoria a tempo di esecuzione. L'insieme di tutte queste procedure di libreria è chiaramente una parte del sistema di I/O.
Benché queste procedure facciano poco più che mettere i loro parametri nel posto richiesto dalle chiamate di sistema, altre procedure di I/O svolgono effettivamente dei compiti. In particolare, la definizione del formato degli ingressi e delle uscite viene fatta da procedure di libreria. Un esempio tratto dal C è la funzione printf, che prende in ingresso una stringa che specifica il formato ed eventualmente altre variabili, costruisce una stringa ASCII, e chiama quindi la write per stampare la stringa. Un esempio di una procedura simile, usata per l'ingresso dei dati, è la scanf che legge i dati e li memorizza nelle variabili descritte in una stringa che specifica il formato utilizzando la stessa sintassi della printf. La libreria standard di I/O contiene alcune procedure che riguardano l'I/O e vengono eseguite come parte dei programmi utente.

Figura 53 - Diversi modi per realizzare la struttura del software di I/O





Sistema strutturato a processi
1-4 sono messaggi di richiesta e risposta tra tre processi indipendenti.

Sistema monolitico
La parte nello spazio utente chiama la parte nello spazio di nucleo mediante una trap.
Il file system chiama il device driver come una procedura

I progettisti di sistemi operativi discutono incessantemente i meriti dei sistemi monolitici, come UNIX, confrontati con i sistemi strutturati a processi. L'approccio di UNIX è più efficiente, dal momento che le chiamate a procedura sono molto più veloci dell'invio di messaggi. Comunque è bene che sappiate che molti progettisti di sistemi operativi non condividono questo modo di vedere le cose.
Non tutto il software di I/O di livello utente è costituito da procedure di libreria. Un'altra categoria importante è il sistema di spooling, un metodo che consente di gestire i dispositivi dedicati di I/O in un sistema multiprogrammato. Vedremo lo spooling in un paragrafo successivo.

6.3 Le procedure di I/O

Nel paragrafo precedente abbiamo fatto passi da gigante in direzione dell'uniformità di gestione delle periferiche, isolando tutte quelle caratteristiche più dipendenti da esse per metterle nel descrittore di periferica. A questo punto siamo in condizione di esaminare come il sistema operativo gestisce le richieste di I/O provenienti da un processo utente.
Una richiesta caratteristica proveniente da un processo utente costituirà una chiamata al sistema operativo nella forma generica:
DOIO (stream, modo, quantità, destinazione, semaforo)
Dove:

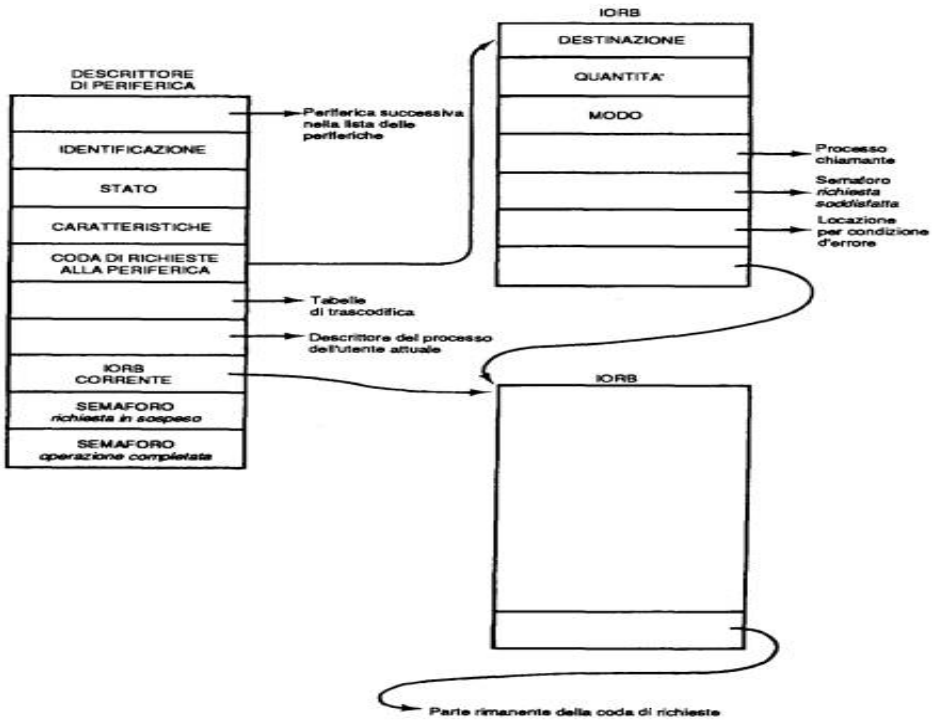
- **DOIO** è il nome di una procedura di I/O di sistema;
- **stream** è il numero di stream su cui deve aver luogo l'I/O;
- **modo** indica quale tipo di operazione è richiesta: per esempio trasferimento di dati e riavvolgimento nastro; può indicare inoltre, se pertinente, quale codice di caratteri va utilizzato;
- **quantità** è la quantità di dati eventualmente da trasferire;
- **destinazione** (o sorgente) è la locazione verso la quale (o dalla quale) deve avvenire l'eventuale trasferimento;
- **semaforo** è l'indirizzo di un semaforo di “richiesta soddisfatta” che deve essere soggetto ad un signal quando viene completata l'operazione di I/O.

La procedura di I/O denominata DOIO è rientrante, così da poter essere usata contemporaneamente da più processi. La sua funzione consiste nell'associare il numero di stream alla periferica reale più opportuna, nel verificare la corrispondenza dei parametri forniti e nell'iniziare il soddisfacimento della richiesta.
La prima di queste operazioni è estremamente semplice: la periferica corrispondente al flusso in questione è determinata dalle informazioni presenti nella lista dei descrittori di stream relativa al processo chiamante al momento dell'apertura del flusso stesso (si veda la figura 6.1). In seguito all'individuazione della periferica, si possono controllare i parametri della richiesta di I/O, verificandone la corrispondenza rispetto alle informazioni contenute nel descrittore della periferica; nel caso in cui si riscontrasse un errore si può rilasciare la risorsa e tornare al processo chiamante. Un particolare tipo di controllo eventualmente attuabile consiste nel verificare che la periferica sia in grado di operare nel modo di funzionamento voluto; un altro consiste nello stabilire se l'estensione e la destinazione del trasferimento di dati corrispondono al modo di funzionamento. Nel caso di periferiche in grado di trasferire solo caratteri singoli, la quantità di dati del trasferimento deve essere pari ad 1 ed essere destinata ad un registro o ad una locazione di memoria. Per quelle periferiche che, invece, trasferiscono direttamente in memoria blocchi di dati, la quantità specificata deve equivalere all'estensione del blocco (fissa o variabile, a seconda della periferica) che deve essere destinato alla locazione di memoria dove deve avere inizio il trasferimento. Una volta completati i controlli, la procedura di I/O riunisce i parametri relativi alla richiesta in un blocco di richiesta di I/O (IORB = I/O Request Block) che provvede poi ad aggiungere ad una coda formata da blocchi analoghi, i quali rappresentano le altre richieste di utilizzo della stessa periferica. Per quanto concerne queste ultime, possono provenire dallo stesso processo o, nel caso di una periferica condivisibile, come il disco magnetico, da altri processi. La coda delle richieste alla periferica viene annessa al descrittore della periferica relativa (vedi figura 6.2) ed è gestita da un processo distinto denominato routine di gestione della periferica (device driver), che descriveremo nel prossimo paragrafo. La procedura di I/O segnala al gestore della periferica di aver sistemato nella coda una richiesta, svolgendo un'operazione di signal su un semaforo di richiesta in sospeso (o pendente) collegato alla periferica e contenuto nel descrittore della stessa. Analogamente, dopo il completamento dell'operazione di I/O, il gestore della periferica ne informa il processo utente tramite un semaforo di richiesta soddisfatta, il cui indirizzo, già parametro della procedura di I/O, è stato passato al gestore come elemento dello IORB. Tale semaforo può essere inizializzato sia dal processo utente che dalla procedura di I/O al momento della creazione dello IORB.

La routine completa di I/O è la seguente:

```
procedure DOIO (stream, modo, quantità, destinazione, semaforo);
begin cerca la periferica nel descrittore di processo;
          controlla i parametri a fronte delle caratteristiche della periferica;
        if errore then uscita per errore;
           raggruppa lo IORB;
        aggiungi lo IORB alla coda delle richieste alla periferica;
 signal (richiesta in sospeso)
 end;
```

Figura 54 - Descrittore della periferica e coda di richieste alla periferica



Come già ricordato nel paragrafo precedente, un gestore delle periferiche è costituito da un processo responsabile del soddisfacimento delle richieste presenti in una coda di richieste alla periferica e della segnalazione, al processo chiamante, del completamento dell'operazione relativa. Vi è un gestore distinto per ogni periferica, ma, poiché tutti i gestori funzionano in modo analogo, possono utilizzare programmi in condivisione. Le eventuali diversità di comportamento tra i singoli gestori vengono dedotte dalle caratteristiche memorizzate nei descrittori delle periferiche interessate.
Un gestore di periferica opera in ciclo continuo nel corso del quale toglie uno IORB dalla coda delle richieste, inizia l'operazione di I/O corrispondente, attende il completamento dell'operazione e segnala al processo chiamante l'avvenuto completamento. Il ciclo completo per un'operazione di input è il seguente:

```
repeat indefinitamente
     begin wait (richiesta in sospeso);
            prendi uno IORB dalla coda delle richieste;
            preleva gli estremi della richiesta;
            inizia l'operazione di I/O;
            wait (operazione completata);
            if errore then registra l'informazione d'errore;
            trascodifica carattere/i se necessario;
            trasferisci dati alla destinazione;
            signal (richiesta soddisfatta);
            cancella IORB
     end;
```

Le seguenti annotazioni, che andrebbero lette con riferimento alla figura 54, possono chiarire i particolari.
Il semaforo "richiesta in sospeso", contenuto nel descrittore della periferica, è soggetto ad un'operazione di signal da parte della procedura di I/O ogniqualvolta quest'ultima provvede a mettere uno IORB nella coda delle richieste della periferica in questione. Se la coda è vuota, il semaforo ha valore 0 ed il gestore della periferica viene messo in stato di sospensione.
Il gestore della periferica può prendere uno IORB dalla coda in base ad un qualunque algoritmo di priorità prescelto. Solitamente risulta adeguato un algoritmo di tipo FIFO, ma il gestore potrebbe essere influenzato dalle informazioni di priorità messe nello IORB da parte della procedura di I/O. Nel caso si tratti di un'unità a disco, il gestore della periferica potrebbe soddisfare le richieste secondo un ordine che rende minimi i movimenti della testina (6.1.5 Software del disco)
Le istruzioni richieste per avviare l'operazione di I/O possono essere prelevate dalle caratteristiche della periferica contenute nel suo descrittore.

Il semaforo operazione completata è soggetto ad un'operazione di signal da parte della routine di interrupt in seguito alla generazione di un interrupt per la periferica in questione. Il programma di massima per la routine di interrupt, in cui si entra da quella di gestione degli interrupt di primo livello, è il seguente:

Figura 55 - Schema del sistema di I/O

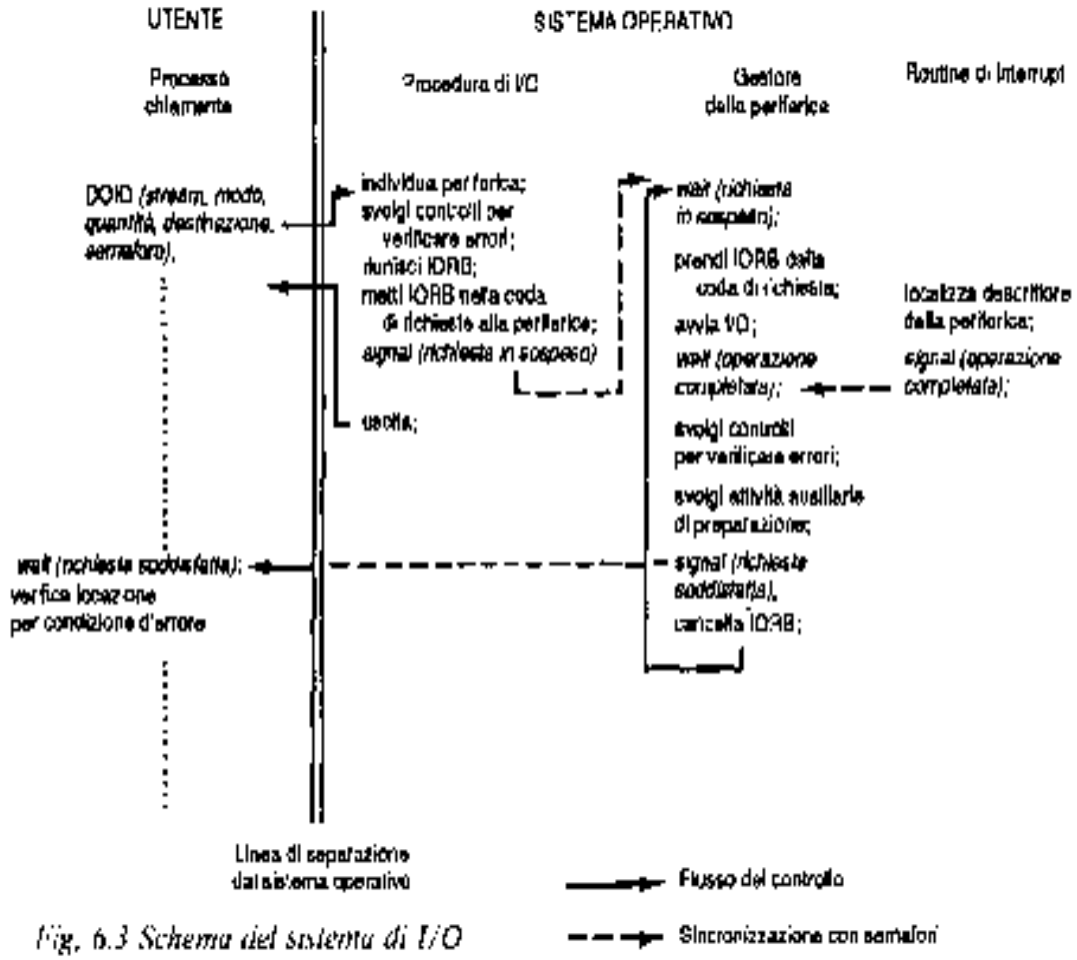


Fig. 6.3 Schema del sistema di I/O

Il semaforo operazione completata è contenuto nel descrittore di periferica. Si noti che la routine di interrupt è estremamente breve; le attività ausiliarie di preparazione nel loro insieme vengono svolte dal gestore della periferica, il quale opera nel modo non privilegiato, come qualsiasi normale processo.
Il riscontro degli errori si effettua consultando lo stato della periferica al completamento dell'operazione. Se si è verificata una situazione d'errore, quale un errore di parità o un inceppamento delle schede, oppure anche un fine file, l'informazione viene depositata in una locazione d'errore il cui indirizzo è inserito nello IORB ad opera della procedura di I/O.
La trascodifica dei caratteri viene effettuata in base al modo specificato nello IORB e le tabelle relative sono collegate tramite il puntatore del descrittore di periferica.
Il ciclo delineato in precedenza è valido per un'operazione di input. Nel caso di un'operazione di output, il prelievo dei dati dalla fonte relativa e l'eventuale trascodifica dei caratteri avvengono prima dell'operazione anziché dopo.
L'indirizzo del semaforo richiesta soddisfatta viene passato al gestore della periferica come componente dello IORB. Viene fornito sotto forma di parametro della procedura di I/O da parte del processo che richiede l'I/O.
La sincronizzazione ed il flusso del controllo tra i processi richiedenti l'I/O, la procedura di I/O, il gestore della periferica e la routine di interrupt relativi sono riepilogati in figura 55. Per quanto riguarda le frecce, le righe intere rappresentano i trasferimenti del controllo, mentre quelle tratteggiate rappresentano la sincronizzazione ottenuta tramite semafori. E importante prendere nota del fatto che, nello schema illustrato, il processo richiedente si evolve in modo asincrono rispetto al gestore della periferica, consentendogli di svolgere altre attività di calcolo o di effettuare altre richieste di I/O durante il soddisfacimento della richiesta originaria. E necessario conservare il processo richiedente solamente in quei casi in cui l'operazione di I/O sia incompleta al momento dell'esecuzione di wait (richiesta soddisfatta).
Lo svantaggio derivante da questa soluzione consiste nel fatto che il processo richiedente deve assumersi il compito di sincronizzare le sue attività con quelle del gestore della periferica. Per esempio, non deve tentare l'utilizzazione di dati di input che non sono ancora stati messi a disposizione. Ciò significa che l'autore del processo deve essere consapevole che le operazioni di I/O non sono istantanee e che devono venir messe in grado di raggiungere la sincronizzazione voluta impiegando adeguatamente il semaforo di richiesta soddisfatta.
Una soluzione alternativa consiste nell'inserire l'onere della sincronizzazione all'interno della procedura di I/O, che fa parte del sistema operativo. Lo scopo si può raggiungere rendendo il semaforo richiesta soddisfatta locale alla procedura di I/O

(il che implica che il suo indirizzo non venga più fornito dal processo richiedente) ed inserendo nella stessa, subito prima della sua uscita, le seguenti operazioni:

```
wait(richiesta soddisfatta);
verifica locazione d'errore;
```

Il ritardo implicito in un'operazione di I/O a questo punto viene mascherato entro il sistema operativo; per quanto riguarda il processo richiedente, l'operazione è istantanea, nel senso che si può supporre completata quando viene eseguita l'istruzione successiva alla richiesta. Per esprimere il concetto in altri termini, il processo richiedente è in esecuzione su una macchina virtuale in cui le operazioni di I/O vengono realizzate per mezzo di un'unica istruzione istantanea.
Si possono così riepilogare le due soluzioni possibili esposte in precedenza: con la prima l'utente ha la facoltà di continuare a lavorare parallelamente all'operazione di I/O, ma ricade su di lui il compito di rilevarne il completamento; con la seconda tale responsabilità gli viene sottratta, ma allo stesso tempo perde anche la possibilità summenzionata.

6.4 GESTIONE DEL BUFFER

La descrizione precedentemente fornita circa la procedura di I/O ed i gestori delle periferiche parte dal presupposto che tutti i trasferimenti di dati avvengano senza il buffer, cioè che ogni richiesta di I/O proveniente da un processo provochi il verificarsi di un trasferimento fisico sulla periferica appropriata. Se un processo sta eseguendo ripetuti trasferimenti sullo stesso stream sarà soggetto ad altrettante sospensioni ripetute al semaforo wait(richiesta soddisfatta) nel corso dei trasferimenti stessi. Per evitare gravosi appesantimenti del sistema dovuti alla commutazione di processi, a volte risulta conveniente eseguire i trasferimenti di I/O prima che vengano effettuate le richieste relative, assicurando in tal modo che i dati siano disponibili al momento necessario. Questa tecnica è nota sotto il nome di buffering.
I trasferimenti di input vengono effettuati dal sistema operativo su una zona di memoria denominata buffer di input. Il processo utente prende i propri dati dal buffer ed è obbligato ad attendere solo quando il buffer si svuota. Al verificarsi di quest'ultima situazione, il sistema operativo riempie nuovamente il buffer ed il processo può continuare. Analogamente, i trasferimenti di output provenienti da un processo vengono destinati ad un buffer di output che il sistema operativo provvede a vuotare completamente quando è pieno. Il processo deve mettersi in attesa solo se tenta di emettere un dato prima che il sistema abbia svuotato il buffer.
Questo procedimento può essere affinato con l'impiego di due buffer, invece che uno solo, ed in tal caso si parla di bufferizzazione doppia (double buffering). A questo punto un processo trasferisce i dati verso (oppure da) uno dei due buffer, mentre il sistema operativo svuota (oppure riempie) l'altro. In questo modo si garantisce che il processo deve attendere solo se ambedue i buffer sono pieni (o vuoti) prima dell'intervento da parte del sistema operativo. Quest'ultima situazione si verifica solamente quando un processo esegue un trasferimento di I/O a raffica (burst) ed in tal caso si può spesso risolvere il problema aggiungendo un numero ancora più elevato di buffer (bufferizzazione multipla). Ovviamente non vi è numero di buffer che tenga nei casi in cui un processo richiede I/O ad una velocità notevolmente maggiore di quella di esercizio delle periferiche. L'utilità dei buffer è limitata all'eliminazione dei picchi nelle richieste di I/O, in situazioni in cui la richiesta media non è superiore a quella che le periferiche di I/O sono in grado di soddisfare. In generale si può dire che tanto più alti sono questi valori di punta tanto maggiore è il numero di buffer necessari per ottenere l'uniformità di trasferimento.
L'inserimento, o meno, di buffer per un dato stream può costituire l'enunciato di uno statement opportuno previsto all'interno della descrizione del lavoro relativo. Se si prevede l'impiego di buffer, l'utente potrà specificarne il numero oppure lasciare che sia il sistema ad occuparsene; in quest'ultimo caso il valore assunto per difetto (default) solitamente è due. Il sistema operativo provvede all'allocazione dello spazio necessario ai buffer al momento dell'apertura dello stream e ne registra l'indirizzo nel descrittore dello stream stesso.
Per consentire lo svolgimento delle operazioni in presenza di buffer è necessaria una procedura di I/O leggermente diversa. Questa gestisce una richiesta di input prelevando il dato successivo dal buffer opportuno e passandolo direttamente al processo richiedente. Solo se un buffer si svuota, la procedura genera uno IORB e segnala al gestore della periferica che è necessario fornire più input. All'apertura dello stream, la procedura di I/O genera IORB sufficienti a provocare il riempimento di tutti i buffer. Il gestore della periferica agisce come in precedenza, avviando un trasferimento di dati sul buffer corrispondente alla locazione indicata nello IORB. Considerazioni analoghe sono valide, con le debite differenziazioni, anche per le richieste di output. In entrambi i casi, la procedura di I/O ed il gestore della periferica congiuntamente rappresentano una variante della coppia produttore/consumatore descritta nel capitolo 3.
La procedura di I/O per operazioni effettuate tramite buffer può essere richiamata da un processo utente tramite un'istruzione scritta nella forma generica:

```
DOBUFFIO (stream, modo, destinazione)
```

dove stream, modo e destinazione hanno lo stesso significato descritto a proposito della procedura DOIO nel paragrafo 6.3. La quantità di informazioni trasferite sarà pari ad un unico dato. Sì noti che, poiché eventuali ritardi derivanti da situazioni relative a buffer pieni o vuoti vengono mascherati all'interno della procedura di I/O, non risulta affatto necessario fornire un indirizzo di semaforo tra i parametri. Il tipo dell'eventuale tecnica di bufferizzazione impiegata, gli indirizzi dei buffer ed i semafori ad uso delle procedure di I/O sono tutti elementi cui si può avere accesso dalla lista dei descrittori di stream relativa al processo chiamante (vedi figura 56(b)). La prima di queste informazioni può servire a determinare il tipo di procedura di I/O da richiamare per effettuare un trasferimento.

6.5 LE UNITÀ DI SUPPORTO AI FILE

Nei paragrafi precedenti ci siamo basati implicitamente sul presupposto che il nome di un'unità periferica costituisca un'informazione sufficiente a determinare la sorgente o la destinazione esterna di un dato trasferimento. Questo risulta vero per periferiche operanti in modo sequenziale, per le quali non sussisterà nessuna ambiguità circa la zona del supporto esterno cui è diretto, o da cui proviene, un trasferimento di dati. A titolo d'esempio, un lettore di informazioni provenienti da tastiera può leggere solo il carattere immesso successivo, mentre una stampante parallela può stampare solo sulla riga emessa in quel momento. Può darsi che in alcuni casi sia possibile far avanzare in certa misura il supporto (poniamo, di una pagina), ma indubbiamente non si riuscirà a inviare un trasferimento, a proprio piacere, verso una parte specifica del supporto. Invece altre periferiche, quali i drive ed i tamburi magnetici, operanti ad accesso casuale, offrono la facilitazione di poter scegliere una particolare area del supporto (disco o nastro) sulla quale effettuare il trasferimento. In questo caso non è sufficiente dare un nome alla periferica interessata; bisogna anche specificare quale area dati del supporto va utilizzata.

Ogni area dati potenzialmente esistente sul supporto viene denominata file e solitamente ha una dimensione definita arbitrariamente all'atto della sua creazione o aggiornamento. Una periferica in grado di memorizzare un file viene denominata unità di supporto a file. Parleremo più ampiamente di questo argomento nel prossimo capitolo, riguardante l'organizzazione dei file; per il momento supporremo che ogni file sia caratterizzato da un nome univoco che può servire al sistema operativo per determinarne la locazione sul supporto relativo. A tale scopo il sistema operativo provvede a mantenere un indice (directory) contenente i nomi dei file e le corrispondenti locazioni.

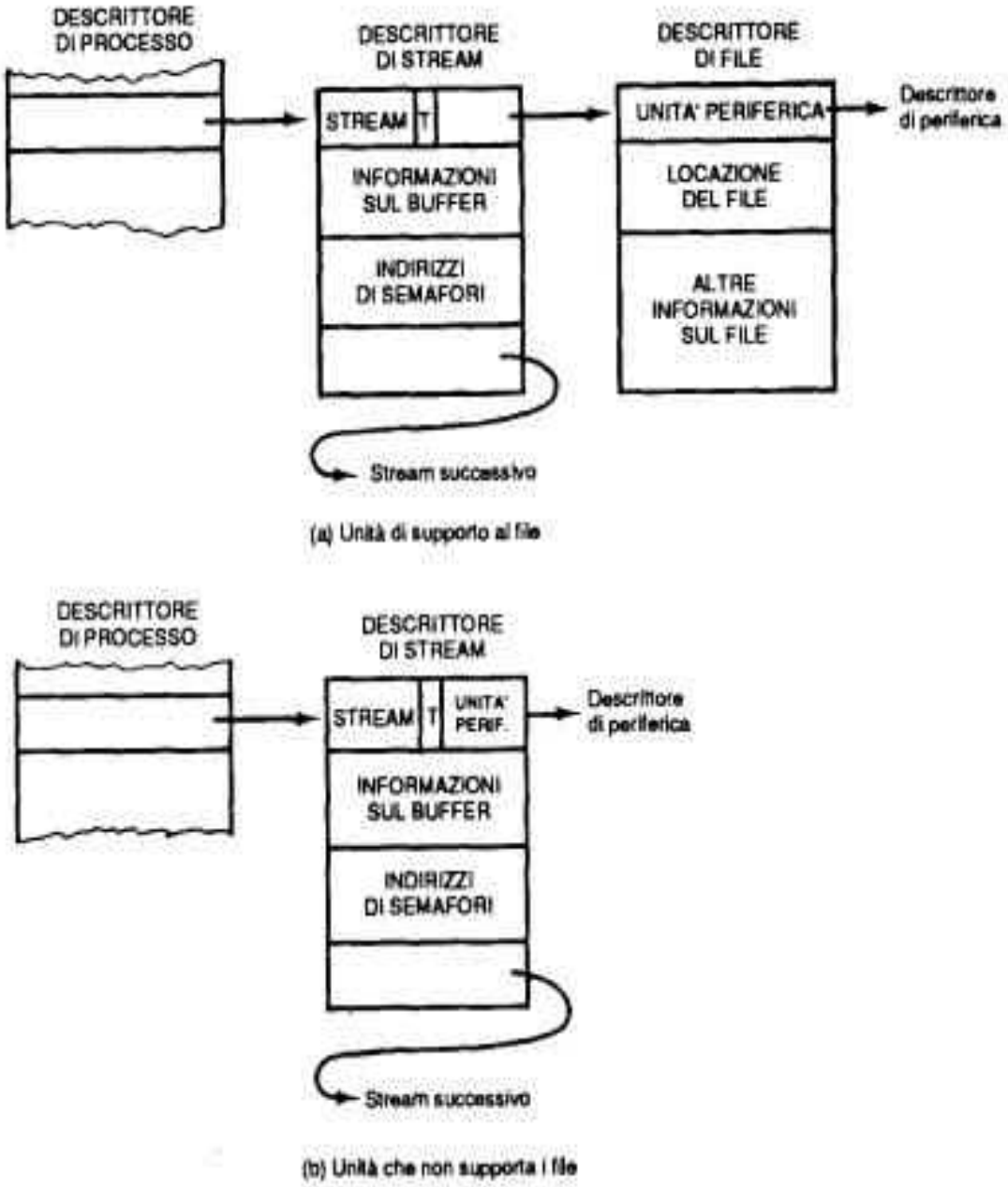


Figura 56 - Informazioni sullo stream per unità di supporto ai file e per unità che non supportano file

Quando si deve trasferire verso (o da) un'unità di supporto ai file un determinato stream di dati, quest'ultimo viene associato nella descrizione del lavoro al nome di un file specifico, anziché al nome dell'unità. Per esempio, uno statement caratteristico di descrizione del lavoro potrebbe essere il seguente:

```
INPUT 1 = "DATI_DI_TEST"
```

indicante che i dati dello stream N.1 devono provenire dal file denominato DATI_DI_TEST. All'apertura dello stream, il sistema operativo cerca nella directory il nome del file per trovare l'unità e la locazione in cui è memorizzato il file. Tale procedura, nota sotto il nome di apertura del file, comprende vari controlli dei quali ci occuperemo con maggior dovizia di particolari nel capitolo che segue. Poiché può trattarsi di un'operazione di lunga durata, non è consigliabile ripeterla per ogni trasferimento di dati. Di conseguenza, ogniqualvolta si apre un file, si crea anche un descrittore di file, per contenervi quelle informazioni necessarie per i trasferimenti che avverranno nell'immediato futuro. Tali informazioni comprendono l'indirizzo del descrittore della periferica su cui è memorizzato il file, la locazione del file su quella periferica, permessi di scrittura o lettura, nonché i particolari inerenti la sua organizzazione interna. Nell'opportuno descrittore di stream del processo che effettua l'apertura del file è inserito un puntatore al descrittore del file, come illustrato in figura 56(a). Ad un raffronto delle figure 56(a) e 56(b), il lettore vedrà che il descrittore del file aggiunge anche altre informazioni oltre a quelle necessarie per associare gli stream alle unità reali. Le procedure di I/O che effettuano l'associazione per ogni

trasferimento di dati possono venir modificate facilmente, cosi da tener conto di queste informazioni durante il raggruppamento degli IORB.

6.6 GESTIONE DELLO SPOOL

Consideriamo un tipico dispositivo gestito mediante tecniche di spooling: una stampante. Benché sia tecnicamente possibile consentire a ogni processo utente di aprire lo speciale file a caratteri corrispondente alla stampante, si può verificare il caso in cui un processo apra il file per poi bloccarsi per ore. Nessun altro processo potrebbe stampare più.
La soluzione consiste nel creare un processo, chiamato demone, e una directory speciale, chiamata la directory di spooling. Per stampare un file, il processo genera in anticipo l'intero file da stampare e lo mette nella directory di spooling. Il demone, che è il solo processo ad avere il permesso di usare il file speciale della stampante, ha il compito di stampare i file contenuti nella directory. In questo modo, proteggendo il file speciale dall'uso diretto da parte degli utenti, si elimina il rischio che ci sia qualcuno che lo tiene aperto inutilmente a lungo.
Lo spooling non viene usato solo per le stampanti, ma anche in altre situazioni. Per esempio, il trasferimento dei file sulla rete viene spesso effettuato da un demone di rete. Per inviare un file da qualche parte, l'utente lo inserisce in una directory di spooling di rete. Più tardi, il demone di rete lo prende e lo trasmette. Un uso particolare della trasmissione di file con spooling è la posta elettronica di Internet. Questa rete è costituita da milioni di macchine sparse per il mondo e comunicanti mediante un gran numero di reti di computer. Per inviare un messaggio a qualcuno, l'utente invoca un programma come send, che accetta la lettera da inviare e la deposita in una directory di spooling, da dove verrà successivamente trasmessa. L'intero sistema di posta elettronica viene eseguito al di fuori del sistema operativo.
Nei paragrafi precedenti si è fatta implicitamente una distinzione tra le unità condivisibili, quali i drive per dischi magnetici, in grado di gestire richieste successive provenienti da processi diversi, e le unità non condivisibili, quali le stampanti parallele, che necessariamente non possono essere allocate per più di un solo processo alla volta. Il funzionamento delle periferiche non condivisibili è tale che la loro assegnazione a più processi porterebbe ad un miscuglio inestricabile di transazioni di I/O. Come già sottolineato nel paragrafo 6.1, l'assegnazione di una periferica non condivisibile viene effettuata al momento dell'apertura dello stream ad essa relativo da parte di un processo; la periferica viene poi rilasciata solo quando si chiude lo stream oppure termina il processo. I processi che intendono utilizzare una periferica già allocata devono ottenerne il rilascio. Ciò comporta che, in periodi di forte richiesta, svariati processi possono essere tenuti sospesi in attesa di poter utilizzare risorse che scarseggiano, mentre in altri momenti le stesse risorse potrebbero rimanere inutilizzate. Al fine di suddividere il carico di lavoro e ridurre l'eventualità di situazioni d'ingorgo (bottleneck) coinvolgenti processi e periferiche risulta necessaria qualche altra strategia.
La soluzione adottata da molti sistemi consiste nell'organizzare tutto l'I/O relativo a periferiche soggette ad utilizzo intensivo tramite lo spool. Ciò significa che, invece di eseguire un trasferimento direttamente sulla periferica relativa allo stream, le procedure di I/O lo effettuano su qualche supporto intermedio (solitamente un disco). Il compito di spostare i dati tra il disco e la periferica necessaria viene attribuito ad un processo a parte, denominato spooler, il quale è associato a quella periferica. A titolo d'esempio, si consideri un sistema in cui tutto l'output relativo alla stampante parallela viene inviato allo spool. Ad ogni processo che apre uno stream della stampante parallela viene assegnato un file anonimo su disco e tutto l'output contenuto nello stream viene inviato a questo file da parte della procedura di I/O. In effetti, il file funge da stampante parallela virtuale. Quando si richiude lo stream, il file in questione viene aggiunto ad una coda contenente file analoghi creati da altri processi, tutti in attesa di essere stampati. La funzione dello spooler della stampante parallela consiste nel prelevare i file dalla coda ed inviarli alla stampante. Si presume, ovviamente, che la velocità della stampante sia sufficiente per gestire tutti i file di output generati entro un determinato intervallo di tempo. La struttura schematica di uno spooler è la seguente.

```
repeat indefinitamente
begin wait (dati da mettere in spool);
     preleva il file dalla coda;
     apri il file;
     repeat fino alla fine del file
     begin DOIO (parametri per lettura da disco);
        wait (richiesta disco soddisfatta);
        DOIO (parametri per output su stampante parallela);
        wait (richiesta stampante soddisfatta)
     end
end;
```

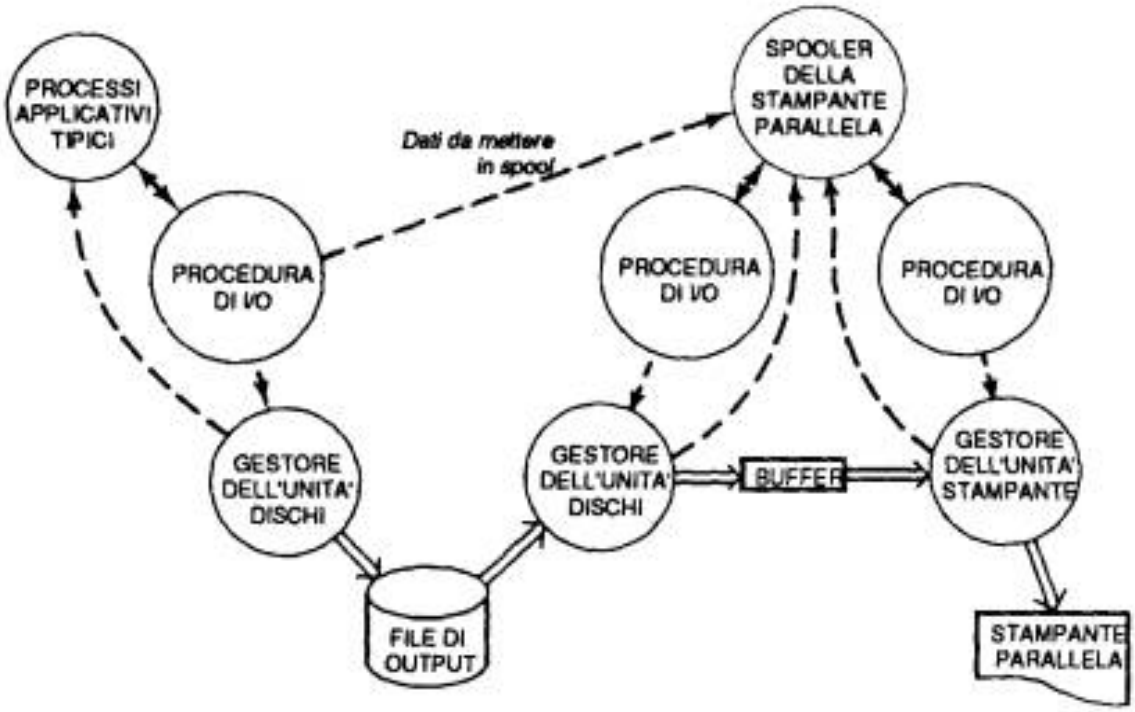


Figura 57 - I processi di spool

la struttura in pratica sarà modificata per consentire l'inserimento di un buffer per i dati tra disco e stampante;
sul semaforo dati da mettere in spool viene svolta un'operazione di signal da qualsiasi processo che chiuda uno stream della stampante parallela (che completa, cioè, un file di output);
i dati in coda di output non devono essere necessariamente elaborati col metodo FIFO; lo spooler potrebbe privilegiare, ad esempio, i file corti anziché quelli lunghi.
I rapporti esistenti tra spooler, gestori delle periferiche e processi che producono output sono riepilogati in figura 57. Uno schema analogo, basato su argomentazioni simili alle precedenti, potrebbe essere redatto anche per uno spooler di input.
Riassumendo, possiamo dire che la funzione di spool serve ad uniformare il carico delle richieste per quelle periferiche soggette ad utilizzo più consistente. Come vedremo nel capitolo 8, diminuisce inoltre la probabilità di situazioni di stallo dovute ad un'imprudente allocazione delle periferiche. Un vantaggio ulteriore è costituito dal fatto che è relativamente semplice produrre copie multiple dello stesso output senza dover rieseguire il lavoro. Per quanto riguarda gli aspetti negativi, invece, ricordiamo la grande quantità di spazio su disco necessario a contenere le code di input e di output, nonché il notevole movimento sul canale del disco, Infine, la funzione di spool non è naturalmente realizzabile in una situazione in cui si operi in tempo reale, dove le transazioni di I/O devono avvenire immediatamente.

7 IL FILE SYSTEM

7.1 OBIETTIVI

Abbiamo ricordato nel capitolo 2 che la memorizzazione a lungo termine è un requisito auspicabile in ogni sistema operativo. La motivazione di questo tipo di memorizzazione varia a seconda della natura del sistema, ma in generale deriva dalla valutazione di uno o entrambi i vantaggi esposti di seguito.

(1) Memoria in linea
Per certe applicazioni che comportano il ritrovamento di informazioni (come, ad esempio, nel caso dei sistemi informativi gestionali) risulta necessario memorizzare grosse quantità di dati in modo che siano sempre accessibili. Anche in un sistema universale, dove un vasto data base in linea può essere superfluo, è molto comodo per l'utente essere in grado di memorizzare programmi o dati all'interno dello stesso sistema di calcolo, anziché su qualche supporto esterno, quali le schede perforate. In particolare, per i sistemi ad accesso multiplo non è pensabile che l'utente riesca nei suoi compiti senza una memoria in linea, dal momento che l'unica periferica di I/O a sua disposizione è il terminale sul quale lavora. Pochi utenti riuscirebbero a sopportare un sistema nel quale debbano essere reinseriti di nuovo tutti i programmi ed i dati all'avvio di ogni sessione sul terminale. Perfino nei sistemi di tipo batch una memoria in linea può portare ad un aumento della produttività, in quanto riduce la dipendenza dalle periferiche più lente.

(2) Condivisione delle informazioni
Per alcuni sistemi è auspicabile che gli utenti siano in grado di condividere le informazioni. Per esempio, gli utenti di un sistema universale potrebbero voler utilizzare reciprocamente i programmi o i dati degli altri utenti ed in un sistema di trattazione delle transazioni molti processi distinti possono servirsi dello stesso data base. Nella maggior parte dei sistemi universali è consigliabile per l'installazione fornire un insieme di programmi di libreria, quali editor, compilatori o sottoprogrammi a carattere scientifico, comunemente disponibili a tutti gli utenti. Se le informazioni vanno condivise in questo modo, devono essere memorizzate in linea per lunghi periodi di tempo.

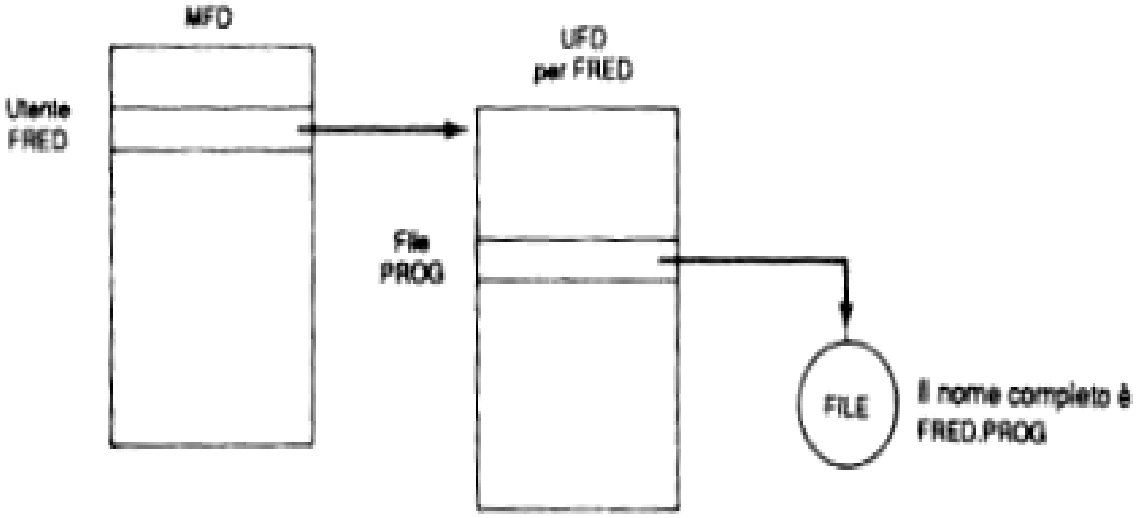
Per motivi economici, la memorizzazione a lungo termine viene effettuata su supporti secondari, quali dischi, nastri magnetici e CD-ROM, e spetta al file system fornire i mezzi idonei all'utente per poter organizzare ed accedere ai dati con comodità. Il modo in cui ciò viene messo in pratica naturalmente dipende dal tipo di dati e dall'utilizzo che se ne intende fare; è prevedibile che il file system per un sistema di trattazione delle transazioni sia piuttosto diverso da quello destinato ad un sistema di controllo dei processi. In questo capitolo ci atterremo allo spirito che pervade il libro, limitandoci ai sistemi universali nei quali i file system in generale sono su dischi; rimandiamo il lettore interessato ad altri settori all'ampia bibliografia esistente (per esempio IFIP, 1969; Judd, 1973; Martin, 1977).
L'utente di un sistema universale dispone i propri dati in file di dimensione arbitraria. Ogni file rappresenta così una raccolta di dati che l'utente considera come un'unica entità; si può trattare di un programma, di un insieme di procedure o dei risultati di un esperimento. Il file è l'unità logica memorizzata e manipolata dal file system. Il supporto su cui sono memorizzati i file viene solitamente suddiviso in blocchi di lunghezza fissa (sono caratteristiche le lunghezze comprese tra 512 e 4096 byte) che il file system deve allocare in numero opportuno per ciascun file. Per tornare utile, il file system deve:

- permettere la creazione e la cancellazione dei file;
- consentire l'accesso ai file per la lettura o la scrittura di dati;
- svolgere automaticamente la gestione della memoria secondaria: non dovrebbe interessare affatto all'utente dove sono ubicati i suoi file all'interno della memoria di massa;
- consentire di fare riferimento ai file tramite un nome simbolico: poiché l'utente non conosce, né vuole conoscere, la locazione fisica dei propri file, per farvi riferimento gli dovrebbe venir richiesto solamente di riportarne i nomi relativi;
- proteggere i file dai malfunzionamenti del sistema: gli utenti si affiderebbero malvolentieri al sistema, se non fossero convinti della sua integrità;
- consentire la condivisione dei file tra utenti cooperanti, ma proteggere i file dall'accesso di utenti non autorizzati.

Nei paragrafi che seguono esamineremo il modo di raggiungere gli obiettivi esposti in precedenza.

7.2 LE DIRECTORY

Il problema fondamentale dell'accesso ai file consiste nell'associare un nome simbolico alla locazione relativa su memoria secondaria. Il mapping si ottiene tramite una directory dei file, o catalogo, che essenzialmente consiste in una tabella contenente informazioni sulle locazioni dei file richiesti. Poiché la directory è il meccanismo tramite cui si accede ai file, è naturale includervi un qualche metodo di protezione dall'accesso di utenti non autorizzati. Tratteremo più approfonditamente questo argomento nel prossimo paragrafo; per il momento osserviamo che si può prevedere una misura di sicurezza immediata dividendo la directory in due livelli, come indicato in figura 7.1. Al livello superiore, una directory dei file principali (MFD = Master File Directory;) contiene per ogni utente del sistema un puntatore diretto ad una directory dei file dell'utente (UFD = User File Directory); al livello più basso, ciascuna UFD contiene i nomi e le locazioni dei file di ogni singolo utente. Siccome si può aver accesso ad una UFD solo tramite la MFD, sì riesce a garantire la riservatezza dei file di un utente tramite il semplice controllo della sua identità a livello della MFD. Inoltre, utenti diversi possono dare lo stesso nome ad un file senza creare confusione, poiché il nome completo di un file può essere considerato come la concatenazione del nome dell'utente relativo (o del suo numero) e del nome proprio del file. Per esempio, il nome proprio del file illustrato in figura 7.1 è PROG ed il suo nome completo è FRED.PROG. Un altro file con lo stesso nome proprio, ma appartenente ad un utente di nome BILL, verrebbe definito con il nome completo di BILL.PROG. In pratica non è sempre necessario specificare il nome completo di un file, poiché il file system è in grado di utilizzare l'identità di chi richiede di accedere al sistema come valore assunto per default relativamente al primo elemento. Solo quando un utente avanza la richiesta d'accesso ad un file di un altro utente è necessario che ne specifichi interamente il nome.



Le informazioni contenute in ogni voce della UFD generalmente sono formate da:

- il nome del file;
- la locazione fisica del file sulla memoria secondaria. La forma di questa informazione dipenderà dal modo in cui è memorizzato il file (vedi paragrafo 7.4);
- il tipo di file (caratteri, binario, rilocabile, ecc.). Queste informazioni in larga misura vengono conservate per comodità dell'utente ed anche perché possono risultare utili per quei programmi di sistema, quali i loader o gli editor, che generalmente vengono usati per operare sui file. Per quanto riguarda il file system in sé, ogni file è considerato semplicemente una stringa di bit;
- informazioni di controllo per l'accesso ad un file (per esempio sola lettura; vedi paragrafo 7.3);
- informazioni di normale amministrazione (per esempio l'ora dell'ultimo aggiornamento).

Tali informazioni servono a fornire dati a quelle attività di sistema che si occupano della conservazione di duplicati, quali garanzia a fronte di eventuali malfunzionamenti dell'hardware (vedi paragrafo 7.5).
Molti sistemi (per esempio il DEC System-10) adottano la struttura di directory a due livelli descritta in precedenza. Altri, invece, come UNIX, allargano il concetto ad una struttura a più livelli in cui gli elementi di una directory possono essere puntatori diretti sia a file che ad altre directory (vedi figura 7.2). Una struttura a più livelli torna utile per quelle situazioni in cui le informazioni memorizzate presentano una classificazione simile ad un albero, oppure laddove gli utenti sono raggruppati in gerarchie come, per esempio, gli individui nell'ambito di équipe di progetto che fanno capo a dipartimenti. in quest'ultimo caso si può applicare un meccanismo gerarchico di protezione effettuando verifiche sempre più selettive mano a mano che si va verso i nodi terminali dell'albero. Come nel sistema a due livelli, si possono risolvere situazioni di eventuale contrasto tra i nomi dei file considerando come nome completo la concatenazione dei nomi propri dei file con i nomi delle directory esistenti sul suo percorso d'accesso.

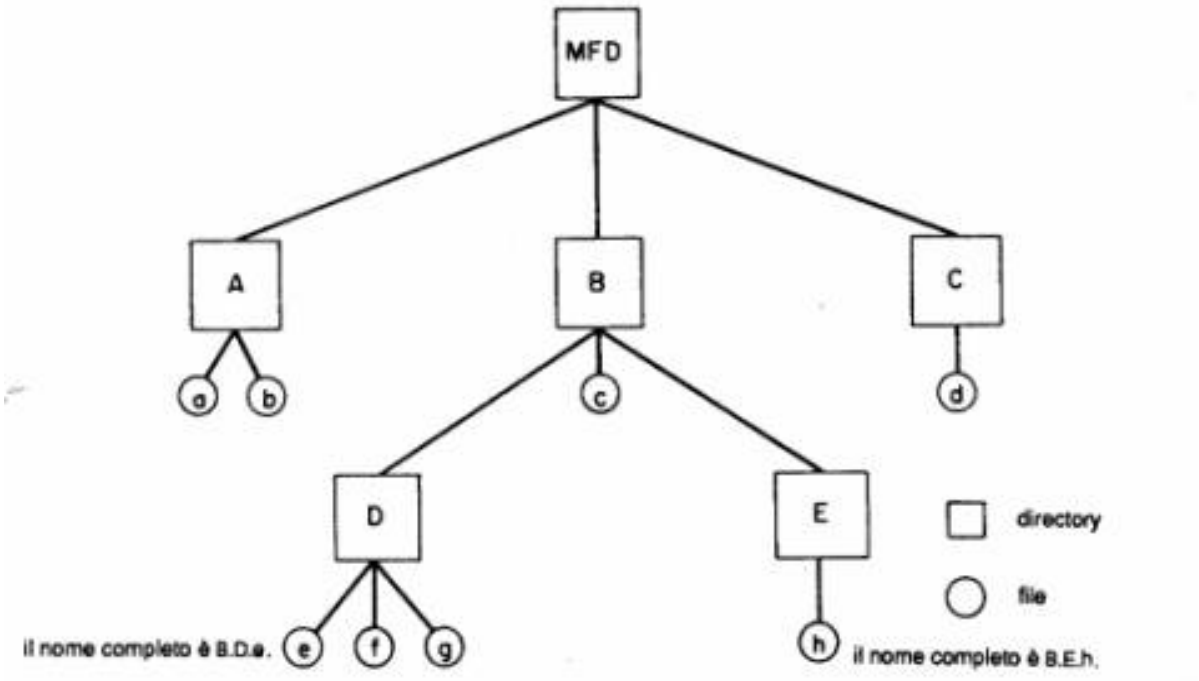


Figura 58 - Struttura di directory a più livelli

Lo svantaggio presente in un sistema a più livelli consiste nella lunghezza del percorso per arrivare ad un qualsiasi file specifico e nel numero di accessi su disco che si devono effettuare per seguire il percorso attraverso le varie directory. Si può rendere in certa misura più agevole quest'operazione sfruttando la tendenza degli accessi successivi ad essere indirizzati verso i file presenti nella stessa directory. Una volta stabilito il percorso per arrivare ad una directory specifica attraverso la struttura ad albero, si può designare quella directory come directory corrente: diviene così necessario, per i successivi riferimenti ai file, specificare solamente il nome proprio del file desiderato. Si effettua il cambiamento di directory corrente specificando il nome completo del file. Questa tecnica è impiegata su UNIX e nel MULTICS (Daley e Neumann, 1965)

7.3 CONDIVISIONE E SICUREZZA

Il problema della sicurezza dei file nasce direttamente dal desiderio di condividerli. In un ambiente senza condivisione, nel quale l'unica persona cui è concesso accedere ad un file è il proprietario stesso, si può raggiungere l'obiettivo relativo alla sicurezza effettuando un controllo di identità a livello della MFD. Tuttavia, quando i file vanno condivisi, il proprietario preferisce disporre di un metodo per specificare a quali utenti va consentito l'accesso ai suoi file ed a quali, invece, no. Inoltre, per il proprietario risulta comodo poter specificare quale tipo di accesso è consentito; potrebbe desiderare che ad alcuni suoi amici sia permesso aggiornare i suoi file, ad altri solo leggerli ed ad altri ancora semplicemente caricarli per metterli in esecuzione. Possiamo riassumere il tutto affermando che il proprietario deve essere in grado di specificare di quale tipo di accesso privilegiato possono disporre gli altri utenti.
Un modo estremamente semplice per realizzare questo obiettivo consiste nell'associare a ciascun file un insieme di accessi privilegiati validi per le varie classi di utenza. Queste ultime potrebbero essere suddivise come segue:

1. il proprietario
2. il gruppo
3. gli altri

mentre gli accessi privilegiati comunemente sono i seguenti:

- nessun accesso (N)
- solo esecuzione (E)
- solo lettura (R)
- solo append (A)
- aggiornamento (U)
- cambiamento di protezione (P)
- cancellazione (D)

Ciascun accesso privilegiato solitamente implica anche l'assegnazione di quelli ad esso precedenti. Una tipica chiave di protezione assegnata dal proprietario al momento della creazione di un file potrebbe essere DAR, indicante che il proprietario può fare qualunque operazione, il gruppo solamente quella di append e tutti gli altri la sola lettura. La chiave di protezione del file risiede nella UFD del proprietario e fa parte delle informazioni relative a quel file specifico. Si noti che

in alcuni casi può risultare pratico per il proprietario concedere solo un accesso di basso privilegio anche a se stesso, al fine di tutelarsi dai propri errori. Ciò nonostante, il file system deve sempre permettere di poter cambiare il livello di privilegio al proprietario che, altrimenti, non avrebbe modo di modificare o cancellare i suoi file.
Questo metodo rende necessario che il proprietario elenchi nella propria UFD quegli utenti designati come soci e questo elenco deve essere verificato per tutti gli accessi, tranne che nel caso del proprietario stesso. In alcuni sistemi è possibile evitare quest'operazione definendo i soci in modo implicito: l'identificazione tramite cui è noto al sistema il proprietario può venire strutturata in modo che il suo primo elemento indichi la condivisione da parte del gruppo ed il secondo l'identità del singolo utente. Per esempio, sul DEC System-10 l'identificazione dell'utente è formata dalla coppia (numero progetto, numero utente) e si presume che tutti gli utenti con identico numero di progetto siano soci, per quanto riguarda le modalità d'accesso ai file. Chiaramente questa tecnica risulta inadeguata quando per file diversi sono necessari diversi gruppi di soci che si sovrappongono.
Il metodo di protezione basato sulle classi di utenza è stato sviluppato inizialmente verso la metà degli anni '60 per l'Atlas 2, che possedeva sette diversi accessi privilegiati e quattro classi di utenza (i soci venivano definiti in modo esplicito dall'utente); ora viene impiegato in svariati sistemi, come il DEC System-10, il quale ha otto accessi privilegiati e tre classi di utenza, con i soci definiti in modo implicito.
Una metodica più generica, in grado di superare le limitazioni di tipo gerarchico delle classi di utenza, consiste nel far elencare dal proprietario delta UFD tutti coloro cui ha assegnato determinati accessi privilegiati, assieme ad una specifica relativa al tipo di privilegi concessi. Se una singola lista viene associata ad un'intera UFD, i privilegi accordati sono necessariamente identici per tutti i file; alternativamente, si possono associare liste distinte ad ogni voce della UFD, così da poter concedere diversi privilegi a gruppi differenti di utenza per file diversi. In quest'ultimo caso, le liste possono occupare una quantità considerevole di spazio, per cui forse non si dovrebbe usare tale metodo se non quando l'aumento di elasticità del sistema si rivela indispensabile.

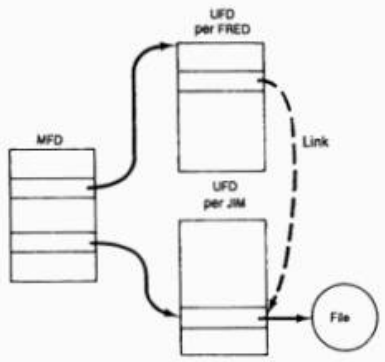


Fig. 7.3: Link tra varie UFD

Una variante di questa tecnica, impiegata nel MULTICS (Daley e Neumann, 1965), è quella in cui il proprietario concede agli altri utenti di creare, nell'ambito dette relative UFD, dei link con le voci della sua UFD. La figura 7.3 mostra un utente Fred che dispone di un link ad un file presente nell'UFD di un altro utente, Jim. Questo link consente a Fred di accedere al file corrispondente che appartiene a Jim. L'UFD di quest'ultimo conterrà una lista con tutti gli utenti cui egli concede l'effettuazione dei link. Il notevole svantaggio derivante dal metodo consiste nel fatto che, quando si cancella un file, devono venir cancellati anche tutti i link ad esso relativi, il che significa che il sistema deve aver modo di individuarli. In alternativa, come nel caso di UNIX, si può gestire un contatore del numero dei Link, cancellando il file solo quando il contatore relativo è a zero.
Riassumendo, le tecniche descritte finora offrono varie soluzioni al problema della condivisione e della sicurezza. In linea di massima, tanto più elastico è il metodo adottato, tanto maggiore è lo spazio occupato dalla UFD, con relativo aumento del tempo necessario ad eseguire un accesso.

7.4 L'ORGANIZZAZIONE DELLA MEMORIA SECONDARIA

Come già ricordato nel paragrafo 7.1, lo spazio di memoria per i file viene allocato in blocchi di dimensione fissa e, poiché tutti i file sono di dimensione variabile, è necessaria una tecnica dinamica di allocazione, sia per i file che per lo spazio libero. Vi sono svariati modi di organizzazione dei blocchi all'interno di un file, tre dei quali vengono da noi descritti qui di seguito.

(1) Interconnessione dei blocchi

Una singola word in ciascun blocco del file viene utilizzata come puntatore al blocco successivo (vedi figura 7.4). Nella UFD per ogni file un puntatore punta al primo blocco della catena. L'appesantimento conseguente in termini di spazio occupato consiste in una word per blocco in ogni file.
Uno svantaggio di questo metodo a concatenazione consiste nel vasto numero di accessi su disco necessari per trovare la fine del file. Questo può rivelarsi particolarmente poco pratico se il file va cancellato e lo spazio, occupato in precedenza, restituito ad una lista libera; le modifiche da apportare ai puntatori, necessarie per la restituzione dello spazio, presuppongono l'individuazione della fine del file. Per questo motivo l'UFD spesso viene estesa, inserendovi un puntatore anche per l'ultimo oltre che per il primo blocco del file.

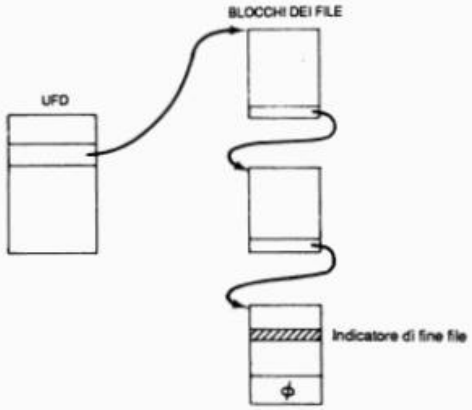


Fig. 7.4: Blocchi concatenati

Il lettore noterà che l'accesso al file è necessariamente sequenziale, poiché si possono raggiungere tutti i blocchi solo percorrendo l'intero concatenamento. Tale metodo, pertanto, è maggiormente adatto a quelle situazioni in cui i file vengono elaborati sequenzialmente. In questi casi l'appesantimento relativo all'accesso al file consiste solo nella lettura dei blocchi consecutivi.

(2) Mappa del file

Per questo metodo di interconnessione la situazione del disco viene registrata in una mappa del file, nella quale ogni blocco su disco è rappresentato da una singola word. L'elemento dell'UFD relativo ad un determinato file punta alla posizione della mappa che rappresenta il primo blocco del file. Questa locazione a sua volta punta alla locazione che denota il blocco successivo nel file e così via (vedi figura 7.5). L'ultimo blocco del file è rappresentato da un puntatore nullo. Così, il file illustrato in figura 7.5 occupa su disco i blocchi 3, 6, 4 ed 8. L'appesantimento conseguente a questo metodo, in termini di spazio occupato, consiste in una word per ogni blocco su disco.

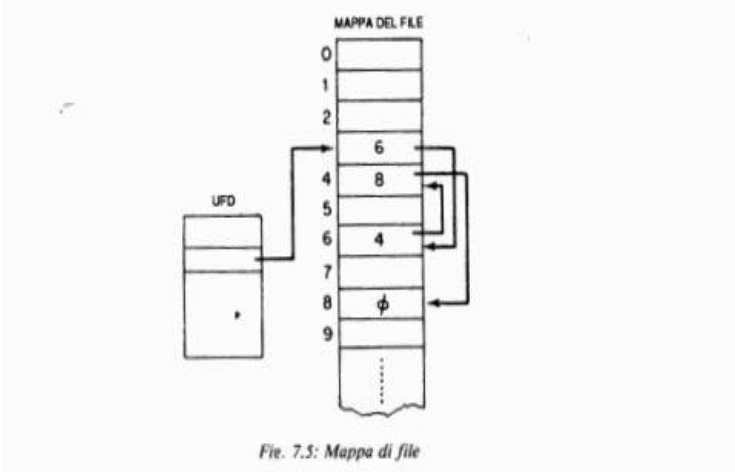


Fig. 7.5: Mappa di file

Come nel caso del metodo a interconnessione dei blocchi, l'accesso al file è necessariamente sequenziale. Come accorgimento utile all'ampliamento e alla cancellazione di un file, l'elemento corrispondente dell'UFD può contenere anche un puntatore alla posizione che, nella mappa del file, rappresenta l'ultimo blocco.

Poiché in generale è troppo vasta per poter essere mantenuta in memoria centrale, la mappa del file deve essere memorizzata su disco, a sua volta sotto forma di file, per essere poi riportata in memoria un blocco alla volta, a seconda delle esigenze. Ciò significa che la lettura di un file di N blocchi può rendere necessari N accessi in più al disco per leggere le sezioni più opportune della mappa del file. L'appesantimento sarà inferiore solo se si dà il caso che alcune posizioni rappresentanti blocchi consecutivi del file si trovino nello stesso blocco della mappa del file. Per questa ragione risulta chiaramente vantaggioso mantenere il più contiguo possibile lo spazio occupato da ogni file, anziché lasciare che il file sia frammentato per tutto il disco.

(3) Blocchi indice

I puntatori per i link relativi ad ogni file sono memorizzati in un blocco indice separato, residente su disco. Se il file è di dimensioni considerevoli, possono essere necessari svariati blocchi indice, concatenati l'un con l'altro (vedi figura 7.6). L'elemento della UFD relativo al file punta al primo blocco indice della catena. Poiché è improbabile che l'ultimo blocco indice di un file sia utilizzato completamente, in media l'appesantimento in termini di spazio occupato è leggermente superiore ad una word a blocco per ciascun file. Per i file più ridotti l'appesantimento è proporzionalmente maggiore rispetto a quelli più estesi.

Il grosso vantaggio dei blocchi indice consiste nel fatto che non è necessario accedere al file sequenzialmente; qualsiasi blocco può essere oggetto di accesso casuale semplicemente specificando il nome del file e lo spostamento (offset) relativo sul blocco indice. Questo metodo di memorizzazione, pertanto, si presta a quelle situazioni in cui i file possiedono una struttura interna con elementi cui si deve accedere separatamente.

L'appesantimento in termini di tempo d'accesso al file è limitato: il primo accesso richiede la lettura del blocco indice, ma gli accessi successivi non comportano appesantimento, sempre che non siano necessari blocchi indice differenti. Tuttavia, l'ampliamento o la cancellazione dei blocchi intermedi di un file comportano un riassestamento dei puntatori all'interno dei blocchi indice. Tale operazione può durare a lungo, per cui questo metodo di memorizzazione non risulta idoneo per i file soggetti a cambiamenti frequenti.

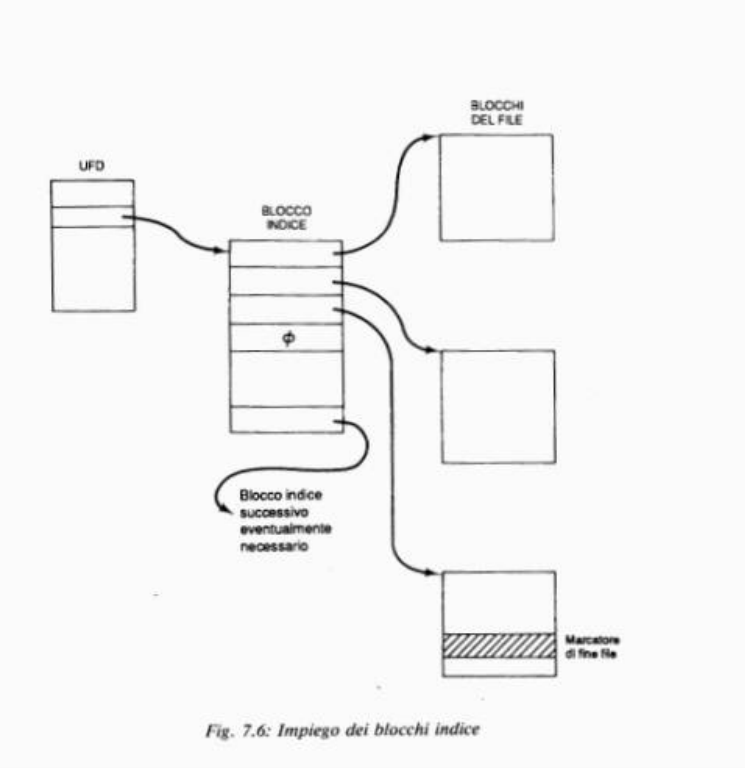


Fig. 7.6: Impiego dei blocchi indice

Osserviamo per inciso che in ognuno dei metodi esposti precedentemente è possibile contrassegnare la fine del file con uno speciale indicatore situato all'interno dell'ultimo blocco, nella posizione più opportuna. Si può realizzare un simile accorgimento per qualsiasi tipo di file in cui sia possibile definire una marca univoca, non interpretabile come dato utile. Per esempio una marca di fine file, nel caso di file contenenti caratteri, potrebbe essere costituita da bit configurati in modo da non rappresentare nessun carattere noto. Non è possibile comunque definire una marca di tal genere per i file binari, in quanto tutte le configurazioni di bit costituiscono dati validi. In tal caso è necessario registrare la lunghezza del file (in word oppure byte) quale elemento aggiuntivo della relativa voce nella UFD. Si tratta di un'idea valida in ogni caso, poiché si fornisce così una verifica ulteriore a scanso di manipolazioni improprie dei file.

Come osservazione finale sull'interconnessione dei blocchi dei file, notiamo che un sistema può consentirne diverse forme, in particolare laddove risulta probabile che ad alcuni file si acceda sequenzialmente, mentre in altri si entra in modo non sequenziale. Il tipo di interconnessione richiesta può essere specificato dall'utente al momento della creazione di un file, con inoltre un eventuale valore assunto per default da parte del sistema.

Un aspetto altrettanto importante dell'organizzazione della memoria di massa consiste nella gestione dello spazio disponibile. Uno dei metodi consta nel considerare i blocchi disponibili come se a loro volta costituissero un file, collegandoli tramite una delle tecniche descritte in precedenza. Se si ricorre alla tecnica del blocco indice, tutte le operazioni da effettuarsi su una catena formata da spazio libero (cioè l'allocazione e la restituzione dei blocchi disponibili) devono aver luogo in coda alla stessa, in quanto se fossero eseguite al suo inizio sarebbe necessario il riassestamento di

un vasto numero di puntatori. Se si impiega la tecnica dell'interconnessione dei blocchi o della mappa del file, le operazioni possono aver luogo indifferentemente ad una delle due estremità della catena. Il metodo del blocco indice si dimostra anche svantaggioso per la necessità di dover cancellare o inserire nell'ultimo blocco indice gli N puntatori relativi agli N blocchi eventualmente allocati o resi disponibili. Per esempio, quando vengono reinseriti dei blocchi liberi alla fine di una catena, gli unici due puntatori che devono cambiare sono quello dell'ultimo blocco della vecchia catena al primo blocco aggiunto e quello della directory di sistema alla fine della nuova catena.
Una metodologia diversa per registrare le disponibilità di spazio consiste nell'impiego di una mappa binaria, costituita da una zona di memoria in cui ogni bit rappresenta uno dei blocchi su disco. Se un blocco è libero, il bit corrispondente è settato a zero; se invece il blocco è utilizzato, il bit è settato ad uno. Alfine di reperire N blocchi liberi è necessario soltanto ricercare all'interno della mappa binaria i primi N bit uguali a zero, eseguendo quindi un semplice calcolo per ottenere gli indirizzi corrispondenti dei blocchi.
In determinati contesti, la mappa binaria può risultare troppo estesa, tanto da non poter essere contenuta comodamente in memoria centrale. In tali casi, la mappa può essere memorizzata su memoria di massa, tenendone in memoria centrale solo una parte che può essere usata per tutte le allocazioni; quando sarà piena (tutti i bit, cioè, saranno ad uno) la si potrà scaricare dalla memoria centrale, ricaricandone un'altra sezione. La restituzione dello spazio disponibile comporta il reperimento della sezione di mappa corrispondente ai blocchi restituiti e l'azzeramento dei bit appropriati. Ciò può causare un movimento intenso nella mappa su disco, che si può, però, ridurre mantenendo una lista di tutti i blocchi restituiti ed utilizzandola per aggiornare la mappa ogniqualvolta ne giunge in memoria centrale una sezione. Bisogna fare attenzione nell'applicazione di questa tecnica, in modo da evitare che, nell'eventualità di un guasto al disco, lo stato della mappa binaria perda la corrispondenza con lo spazio allocato.
Solitamente la scelta dell'estensione dei blocchi è determinata dal progettista hardware (eccezione degna di nota è la serie IBM 370, in cui l'utente può effettuare scelte distinte per ogni file) e viene effettuata in base ai criteri che seguono:
lo spazio sprecato a causa del mancato riempimento di alcuni blocchi, che aumenta di pari passo con l'aumentare della dimensione degli stessi;

- lo spazio sprecato per via dei puntatori della catena, che devono essere tanto più numerosi quanto minore è la dimensione dei blocchi;
- l'unità impiegata dal supporto di informazione per il trasferimento di dati in memoria centrale; la dimensione dei blocchi dovrebbe essere un multiplo di questa unità, così da utilizzare appieno ogni trasferimento di dati;
- lo spazio di memoria necessario per ogni operazione di lettura o scrittura su file.

A seconda della modalità di organizzazione del disco, può risultare necessario reperire spazio in memoria per il blocco indice in uso, ovvero per il blocco contenente la mappa del file in uso, nonché il blocco oggetto di lettura o scrittura.
Le dimensioni caratteristiche dei blocchi sono comprese tra 512 e 4096 byte.
Le strategie di allocazione possono variare dalle più lineari (impiego del primo blocco presente nella catena dello spazio disponibile o del blocco della mappa binaria) alle più sofisticate (impiego del blocco che rende minima l'entità del movimento della testina sul disco durante la lettura del file). Infatti, l'appesantimento in termini di tempo di ricerca, nel corso della lettura di un file i cui blocchi sono distribuiti per tutto il disco, può rivelarsi notevole e portare alla saturazione del canale del drive. Lo si può ridurre tramite il ricorso alla strategia suesposta oppure sottoponendo periodicamente la lista dei file al concatenamento del numero più alto possibile di file. Quest'ultima tecnica può essere incorporata nelle procedure di salvataggio descritte nel prossimo paragrafo.

7.5 L'INTEGRITÀ DEL FILE SYSTEM

Poiché il contenuto dei file può rappresentare il lavoro di molti mesi o essere costituito da dati insostituibili, è indispensabile che un file system preveda gli opportuni meccanismi per il salvataggio ed il recupero dei dati nella non improbabile evenienza di un malfunzionamento dell'hardware o del software. Ciò significa che il sistema deve conservare copie di tutti i file in modo da poterli ripristinare successivamente al verificarsi di un inconveniente.
Esistono essenzialmente due modi per fare la copia di un file. Il primo, che è anche il più semplice, è noto sotto il nome di dump (backup) periodico (o massiccio). Ad intervalli prestabiliti, l'intera memoria d'archivio viene copiata su qualche supporto, solitamente un nastro magnetico, e nel caso di malfunzionamento tutti i file possono essere ricreati nello stesso stato in cui erano scritti al momento del dump. File singoli inavvertitamente cancellati possono essere ripristinati a prezzo di esaminare l'intero nastro del dump. Gli svantaggi inerenti al dump periodico sono i seguenti:
può essere necessario mettere fuori uso il file system durante il dump (l'alternativa consiste nel non sottoporre a dump quei file eventualmente aperti ed in corso di scrittura);
di solito il dump impiega molto tempo (da 20 minuti a 2 ore, a seconda delle dimensioni del sistema e della velocità delle unità nastro); ciò significa che non lo si può effettuare molto spesso e che, pertanto, un file ripristinato potrebbe facilmente risultare non più aggiornato.
La seconda tecnica, più raffinata della precedente, è il dump incrementale (Fraser, 1969). Con tale metodo si fa il dump solamente di quelle informazioni che sono state modificate dopo il dump precedente. Ciò significa che in ogni operazione sono soggetti a dump solo i file che sono stati creati o modificati e gli elementi delle directory che sono stati parimenti cambiati. L'entità ditali informazioni sarà limitata e quindi il dump può essere effettuato ad intervalli più frequenti. Al fine di determinare quali file ricopiare, viene settato a 1 un flag nell'elemento della directory relativo a ciascuno dei file, ogni volta che questi ultimi vengono modificati; il flag viene poi azzerato in seguito al dump. Affinché il processo di dump non debba esaminare tutte le voci della directory, in ogni UFD si può usare un flag globale che indichi se sono stati eventualmente settati ad 1 dei singoli flag all'interno della UFD in questione. Poiché il processo di dump non prende in considerazione i file i cui flag non sono settati a 1, includendo quindi tutti i file in corso di aggiornamento, può essere eseguito parallelamente alle normali attività. E' anche possibile, come avviene nel MULTICS, che il processo di dump abbia una priorità bassa e sia sempre presente in macchina, alla continua ricerca entro la directory di file aggiornati.
Gli svantaggi del dump incrementale sono costituiti dalla grande quantità di dati generati e dalla complessità della procedura di recupero. in seguito ad ogni inconveniente, le informazioni devono essere ricostruite dalla sequenza di eventi registrata sui nastri di dump. I nastri vengono montati in ordine cronologico inverso ed il file system viene ripristinato così com'era al momento dello svolgimento dell'ultimo dump periodico. (i dump periodici saranno effettuati ad intervalli di tempo non frequenti, solitamente una volta alla settimana). Durante questo procedimento, il processo di ripristino preleva dai nastri di dump solo quei file che non sono già stati ripristinati, cosicché quelli più recenti non vengono soppiantati dalle loro versioni precedenti. Infine viene usato l'ultimo nastro di dump periodico per completare l'operazione di recupero. Questo modo di procedere è migliore rispetto a quello inverso, consistente nell'avviare il nastro di dump periodico a partire dalla prima operazione registrata in ordine di tempo, in quanto non vengono ripristinate le informazioni ridondanti.

Vale la pena osservare che, qualsiasi metodo di dump si impieghi, il recupero del sistema rappresenta una buona occasione per compattare i file in blocchi contigui. Così facendo si diminuisce l'appesantimento relativo all'accesso ai file, a scapito di un lieve prolungamento della procedura di ripristino.

7.6 APERTURA E CHIUSURA DEI FILE

Nel capitolo precedente abbiamo osservato che, quando si associa uno stream di I/O ad un file, il sistema operativo deve cercare la periferica e la locazione in cui è memorizzato il file. La procedura è nota come apertura del file; la procedura inversa di chiusura del file viene svolta sia esplicitamente, quanto è stato completato tutto l'I/O relativo al file, oppure implicitamente, quando termina il processo che fa uso del file.
Per implementare queste operazioni abbiamo bisogno di due procedure, facenti parte del sistema operativo, della forma:

```
open (filename, modo)
close (filename)
```

dove filename è il nome del file da aprire o chiudere e modo è il tipo d'accesso richiesto (per esempio lettura, scrittura o creazione). Le operazioni svolte da open sono le seguenti:

- trova il riferimento del file nella directory;
- verifica che il processo chiamante sia stato attivato da un utente con i privilegi necessari per l'accesso nel modo specificato;
- effettua controlli per assicurare che, se il file è già aperto in lettura (eventualmente da parte di un altro processo), non si possa ora aprire per scrivervi e che, se è già aperto in scrittura, non si possa aprire affatto (queste verifiche saranno descritte tra un attimo);
- specifica la periferica e la locazione in cui è memorizzato il file (se il file deve essere creato ex novo, la locazione viene specificata dalla routine di allocazione);
- crea un descrittore di file contenente tutte le informazioni sul file inerenti i prossimi trasferimenti di dati.

Come già accennato nel paragrafo 6.5, il descrittore di file viene utilizzato dalle procedure di I/O quale fonte di informazioni per la costruzione delle richieste di I/O, rendendo così superfluo il passaggio attraverso la procedura di apertura per ogni trasferimento di dati che interessa il file.
Le informazioni necessarie nel descrittore di file comprendono:

- il nome del file;
- l'indirizzo del descrittore relativo alla periferica in cui è memorizzato il file;
- la locazione del primo blocco del file;
- la locazione del blocco successivo da leggere o scrivere (nell'ipotesi che l'accesso sia sequenziale);
- il modo d'accesso.

Il descrittore di file viene individuato tramite un puntatore del relativo descrittore di stream inerente al processo che ha aperto il file, come illustrato in figura 56(a).

I meccanismi di lettura/scrittura, citati al punto (3) della descrizione di open, si possono implementare includendo fra le informazioni delle directory relative a ciascun file due elementi aggiuntivi. Il primo è un “write bit” (bit di scrittura) che viene settato a 1 ogniqualvolta si apre il file per scopi di scrittura; il secondo è uno “use count” (conteggio di utilizzo) relativo al numero dei processi che dispongono del file aperto per fini di lettura. Un processo può aprire un file per la sola lettura se il write bit è uguale a zero e lo può aprire per la sola scrittura se sia il write bit che lo use count sono a zero. Purtroppo, questa tecnica presenta un problema, in quanto più processi potrebbero voler aprire o chiudere lo stesso file contemporaneamente, apportando così cambiamenti simultanei alla relativa voce nella directory. È possibile superare questa situazione effettuando la stesura delle procedure di open e close sotto forma di sezioni critiche, ciascuna racchiusa dallo stesso semaforo di mutua esclusione, anche se tale accorgimento presenta lo svantaggio che, se un processo viene interrotto mentre apre o chiude un file, queste operazioni non sono più disponibili per nessun altro processo.
Un metodo alternativo consiste nel formare una lista dei descrittori di tutti i file che sono aperti su ogni specifica periferica. La lista può essere collegata tramite un puntatore del descrittore della periferica interessata. Quando si deve aprire un file, la procedura di open controlla la lista per vedere se eventualmente esistono già descrittori per il file e, in caso affermativo, qual è il modo d'accesso da essi indicato. La procedura non consente l'apertura del file, a meno che non siano soddisfatte le limitazioni di lettura/scrittura. Si può impiegare un semaforo di mutua esclusione per proteggere la lista dall'accesso contemporaneo di quei processi che vogliono aprire o chiudere lo stesso file nello stesso momento. Lo svantaggio di questa tecnica consiste nel tempo necessario a controllare la lista che, nei sistemi ad accesso multiplo di dimensioni considerevoli, può facilmente contenere un centinaio di descrittori. Il tempo diventa un fattore particolarmente critico, in quanto la lista può essere sottoposta a verifica da un solo processo per volta e, nel caso il processo in questione venisse interrotto, questa diverrebbe inaccessibile per gli altri processi.
Una soluzione parziale di questi problemi consiste nel dividere ogni descrittore di file in due strutture distinte: un descrittore centrale di file per ogni file aperto ed un descrittore locale di file per ogni processo che utilizza un file. Si crea un descrittore locale ogniqualvolta un processo apre un file e tutti i descrittori locall di un determinato file puntano all'unico descrittore centrale relativo al file in questione. I descrittori centrali sono collegati con gli opportuni descrittori di periferica (vedi figura 7.7). Un descrittore centrale contiene informazioni identiche per tutti i processi che usano il file; più precisamente:

- il nome del file;
- l'indirizzo del descrittore della periferica in cui è memorizzato il file;
- la locazione del primo blocco del file;
- lo use count del file;
- il write bit del file.
- Un descrittore locale contiene informazioni pertinenti al processo che fa uso del file e più precisamente:
- la locazione del blocco successivo da leggere o scrivere;
- il modo d'accesso;
- nonché un puntatore rivolto al descrittore centrale di file.

L'introduzione dei descrittori centrali di file elimina la doppia memorizzazione delle informazioni in descrittori separati e diminuisce la lunghezza della lista dei descrittori per ogni periferica. Inoltre, diminuisce l'entità dell'attività svolta dalle procedure di open e close inerente la ricerca all'interno della lista dei descrittori e, cosa ancor più significativa, consente di applicare la mutua esclusione singolarmente a ciascun processo, anziché alla lista nel complesso, evitando così il rischio riguardante il verificarsi di strozzature con conseguente caduta dell'efficienza del sistema (bottleneck), come già ricordato nei capitoli che precedono. Il semaforo di mutua esclusione può essere contenuto nel descrittore centrale di ciascun file.

Una variante di questa tecnica (Courtois et al., 1971)consiste nel sostituire il write bit del descrittore centrale di file con un altro semaforo w. Le procedure di open e close quindi vengono redatte in modo da includervi i seguenti segmenti di codifica:

Open

```
        if modo = read then
                begin wait (mutex);
                        usecount:=usecount + 1;
                        if usecount I then wait (w);
                        signal (mutex)
                end
        else wait (w)
```

Close

```
        if modo = read then
                begin wait (mutex);
                        usecount:=usecount - 1;
                        if usecount = 0 then signal (w);
                        signal (mutex)
                end
        else signal (w)
```

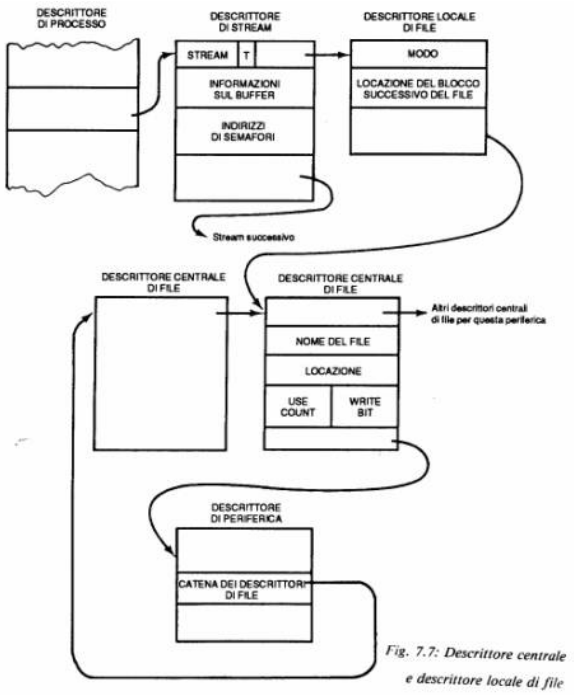


Fig. 7.7: Descrittore centrale e descrittore locale di file

In questa variante, le richieste d'accesso respinte vengono implicitamente inserite nella coda d'attesa al semaforo w, mentre nella versione originaria il progettista di sistema conserva la facoltà di mettere o meno in coda le richieste, in base all'applicazione.
Il funzionamento della procedura close è relativamente semplice. Consiste nel cancellare il descrittore locale di file e nell'effettuare il decremento dello use count del descrittore centrale. Se lo use count è uguale a zero, viene cancellato anche il descrittore centrale e si aggiorna la directory conformemente alle necessità del caso (se, per esempio, il file è appena stato creato).
La cancellazione di un file può essere gestita come una chiamata della open con un parametro modo opportuno. Si presenta comunque lo svantaggio che, se il file è già aperto, come spesso avviene per i file di libreria, la richiesta di cancellazione viene respinta. Tutto ciò rende alquanto problematica la sostituzione dei file di libreria. Un modo per aggirare il problema consiste nell'inserire in ogni descrittore centrale di file un “delete pending bit” (bit di cancellazione pendente) che viene settato ogniqualvolta viene effettuata una richiesta di cancellazione per un file aperto. La procedura di close verifica lo stato di questo bit: se è settato a 1 e lo use count va a zero, si può cancellare il file. La cancellazione viene accelerata se la open respinge una richiesta d'accesso ad un file il cui bit di cancellazione pendente è settato a 1.

7.7 CONCLUSIONI

In questo capitolo abbiamo trattato vari aspetti dei file system per applicazioni universali. Il passo successivo nella costruzione del nostro sistema operativo ipotetico consiste nell'adozione di un sistema particolare basato sulle tecniche che abbiamo descritto. Otterremo così un altro strato da aggiungere alla nostra “cipolla”, costituito dalla codifica per l'implementazione di particolari meccanismi volti a garantire protezione ed integrità e finalizzati anche ad amministrare lo spazio disponibile e ad aprire e chiudere i file. Le ulteriori strutture dati permanenti sono le directory ed una qualsiasi struttura che serve per registrare l'allocazione dello spazio di memoria. Invece, le strutture dati temporanee esistenti solo mentre è aperto un file sono costituite dai descrittori locall e centrali di file. Lo stadio attuale del nostro sistema è illustrato in figura 7.8.

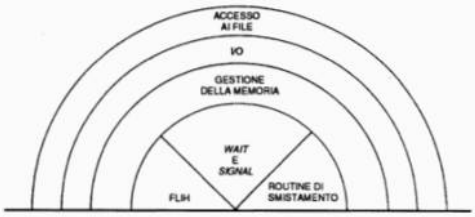


Fig. 7.8: Stato attuale del sistema operativo ipotetico

8 SCHEDULAZIONE E ALLOCAZIONE DELLE RISORSE

Fino a questo punto, nello sviluppare il nostro sistema operativo ipotetico siamo partiti dal presupposto che tutti i processi dispongano delle risorse di cui necessitano. È giunto ormai il momento di prendere in considerazione le modalità secondo le quali i processi acquisiscono le proprie risorse e riescono a condividerne un insieme limitato in modo efficiente. La nostra trattazione inerente l'allocazione delle risorse includerà anche la schedulazione, in quanto le due funzioni sono strettamente correlate: le scelte relative alle priorità dei processi possono dipendere dal tipo di impegno delle risorse e l'inserimento di nuovi processi nel sistema risente chiaramente dell'entità del potenziale di risorse residue. Infatti, la schedulazione, in quanto riguardante l'allocazione delle CPU, può essere considerata come una delle attività relative all'allocazione delle risorse presa nel suo complesso.

8.1 OSSERVAZIONI DI CARATTERE GENERALE

All'interno di un ambiente con risorse illimitate sarebbe perfettamente accettabile un metodo di acquisizione basato semplicemente sulle necessità del momento. Purtroppo, si riesce raramente a mettere a disposizione risorse in grado di soddisfare le richieste concorrenti di tutti i processi presenti nel sistema e bisogna, quindi, concepire tecniche miranti alla condivisione di una serie limitata di risorse tra più processi che competono per il loro utilizzo. Gli obiettivi di queste tecniche sono i seguenti:

1. attuare la mutua esclusione dei processi rispetto a risorse non condivisibili;
2. prevenire situazioni di stallo (vedi paragrafo 3.2) derivanti dalle richieste di utilizzo delle risorse;
3. assicurare un alto livello di utilizzazione delle risorse;
4. dare modo a tutti i processi di acquisire, entro un arco di tempo "ragionevole", le risorse di cui hanno necessità.

Il lettore noterà che questi obiettivi non sono necessariamente in armonia l'uno con l'altro. In particolare, l'obiettivo (4), in cui è implicito il soddisfacimento dell'utente, generalmente è raggiungibile solo a scapito dell'obiettivo (3). Ciò è dovuto al fatto che, con l'innalzamento del livello di utilizzazione delle risorse, aumenta anche il tempo medio di attesa per l'accoglimento delle richieste ad esse relative. L'alternarsi di situazioni soddisfacenti per l'utente con l'utilizzo ottimale delle risorse rappresenta uno dei criteri su cui si può basare la valutazione delle metodiche di schedulazione e di allocazione delle risorse. In un sistema operante in tempo reale con un tempo di risposta garantito si può prevedere che tale avvicendamento sia favorevole all'utente, mentre un sistema di tipo batch potrebbe privilegiare l'utilizzazione ottimale delle risorse. Questo crea un problema gestionale di difficile soluzione in quei sistemi che tentano di fornire sia il funzionamento di tipo batch che quello ad accesso multiplo.

È utile esaminare l'allocazione delle risorse sotto due punti di vista: i meccanismi e le metodiche. Per meccanismi intendiamo l'aspetto più pratico relativo al modo in cui viene effettuata l'allocazione, includendo particolari quali la struttura dati per la descrizione dello stato delle risorse, le tecniche per garantire l'uso esclusivo di risorse non condivisibili ed i metodi per mettere in code d'attesa quelle richieste di utilizzo che non possono essere soddisfatte immediatamente. Le metodiche regolano le modalità di applicazione dei meccanismi. Si occupano del modo più accorto per soddisfare le richieste, anche quando sono disponibili risorse adeguate, comprendendo quindi anche le questioni relative alle situazioni di stallo ed al bilanciamento del sistema, in quanto un'allocazione poco prudente potrebbe portare a situazioni in cui alcuni processi non riescono ad evolversi o in cui il sistema è sovraccarico di richieste rispetto ad una determinata categoria di risorse. E' proprio in relazione a quest'ultimo punto che l'allocazione delle risorse viene a collegarsi con la schedulazione, poiché si possono ridurre le probabilità di sovraccarico prendendo decisioni accurate di schedulazione inerenti le priorità dei processi e l'introduzione di nuovi processi nel sistema. Nei paragrafi che seguono tratteremo sia dei meccanismi che delle metodiche.

8.2 I MECCANISMI DI ALLOCAZIONE

Dalle osservazioni fatte in precedenza dovrebbe risultare chiaro che si può considerare risorsa qualsiasi elemento di un sistema di elaborazione presente in disponibilità limitate e con la necessità di essere condiviso. In linea generale, infatti, le risorse rientrano in una delle categorie citate in seguito. Passeremo in rassegna i meccanismi di allocazione per ciascuna categoria, dando il dovuto peso alla parte del sistema che effettua l'allocazione, alle strutture dati impiegate per descrivere le risorse ed ai metodi di accodamento dei processi in attesa dell'allocazione richiesta.

(1) CPU

Abbiamo già esaminato nel capitolo 4 il modo in cui una CPU viene assegnata, da parte della routine di smistamento, al primo processo non ancora in esecuzione presente nella coda d'attesa al processore. I dati che definiscono una CPU possono essere contenuti in un descrittore di processore, il quale è simile al descrittore di periferica utilizzato per i dati relativi ad un'unità periferica (vedi capitolo 6). Il contenuto caratteristico di un descrittore di processore potrebbe essere il seguente:

a. l'identificazione del processore;
b. lo stato attuale (modo privilegiato o non privilegiato);
c. un puntatore relativo al descrittore del processo corrente.

I descrittori di processore possono essere contenuti a loro volta nella tabella centrale, oppure essere individuati da un suo puntatore.

In una configurazione con CPU di tipi diversi, i descrittori di processore possono essere ampliati includendovi un'indicazione circa le caratteristiche della singola unità (per esempio se è dotata o meno di dispositivi hardware per la virgola mobile). In tal caso, ciascun processore potrebbe essere più idoneo all'esecuzione di un tipo particolare di processo ed avere quindi la propria coda d'attesa a cui punterebbe il suo descrittore. Le code d'attesa al processore, quindi, sono analoghe alle code di richieste delle periferiche. L'assegnazione dei processi a code d'attesa al processore diverse riguarda un settore in larga misura non ancora esplorato e, quindi, non ce ne occuperemo dettagliatamente in questa sede.

(2) Memoria

Nel capitolo 5 abbiamo visto come può essere allocata la memoria dalla componente del nostro sistema operativo che si occupa della sua gestione, sia nel caso di paginazione che in assenza di essa. Le strutture dati che definiscono lo stato della memoria sono le page table, le segment table o le liste di blocchi disponibili, a seconda dell'architettura della macchina. Un processo, in attesa del trasferimento di una nuova pagina o segmento dalla memoria secondaria, viene messo in attesa d'eventi, ed i bit di stato nel suo descrittore ne indicano il motivo. Le richieste per il trasferimento di pagine o segmenti vengono accodate dal sistema di gestione della memoria sotto forma di blocchi di richiesta (IORB), gestiti dal gestore relativo alla periferica interessata su cui risiede la memoria secondaria.

(3) Periferiche

Abbiamo descritto nel capitolo 6 il modo in cui, nei sistemi con I/O basato su stream, ad un processo viene assegnata una periferica non condivisibile al momento dell'apertura dello stream corrispondente. Nei sistemi che non fanno uso di stream, una periferica può essere assegnata a seguito di una richiesta inviata direttamente al sistema operativo. In entrambi i casi il meccanismo di assegnazione è lo stesso. La struttura dati che definisce una periferica è il relativo descrittore ed i processi in attesa dell'assegnazione di una periferica possono essere accodati ad un semaforo compreso in tale descrittore. La mutua esclusione dall'utilizzo della periferica è garantita tramite l'inizializzazione ad uno del valore del semaforo.

(4) Memoria ausiliaria

La memoria ausiliaria utilizzata nell'implementazione della memoria virtuale viene allocata dal sistema di gestione della memoria, mentre quella impiegata per i file è allocata dal file system. Alcuni sistemi operativi (in particolare il MULTICS) non fanno distinzione tra file e segmenti, integrando così lo spazio di memorizzazione dei file nella memoria virtuale, mentre il compito di effettuare tutte le allocazioni viene svolto dal file system. La struttura dati utilizzata per l'allocazione consiste in un qualche tipo di lista dei blocchi disponibili o di mappa binaria, come descritto nel paragrafo 7.4.

Le richieste di spazio per i file vengono respinte solo se il singolo utente interessato ha superato la sua quota, oppure se l'intera memoria ausiliaria è piena. In nessuno dei due casi risulta pratico accodare le richieste: la prima delle due situazioni è risolvibile solo tramite l'intervento dell'utente stesso, mentre, per quanto riguarda la seconda, non si può garantire un mutamento entro un breve arco di tempo.

(5) File
Si può considerare come risorsa anche un singolo file, nel senso che più processi possono mirare a condividerlo. Fintantoché tutti i processi operano nel modo di "solo lettura", la risorsa è condivisibile; se un processo intende scrivere dentro al file, la risorsa diventa non condivisibile.
L'allocazione di un file per un processo viene effettuata dal file system al momento dell'apertura del file stesso e i meccanismi di interlock per le operazioni di scrittura vengono messi in pratica tramite i metodi descritti nel paragrafo 7.6. Le strutture dati che definiscono i file naturalmente sono le directory.
Di solito i sistemi di tipo batch non accodano le richieste di accesso ai file in quanto richieste contemporanee (di non lettura) solitamente indicano che l'utente ha commesso un errore. Nella trattazione delle transazioni e nel controllo dei processi, comunque, può essere conveniente fissare delle code d'accesso all'interno del file system. Tali code possono essere associate a semafori, come si è visto nel paragrafo 7.6.
Risulta chiaro dal precedente quadro d'insieme che i meccanismi di allocazione vengono implementati ai vari livelli del sistema operativo, in quanto ogni meccanismo si trova al livello più opportuno per la risorsa allocata. Vedremo, comunque, che le metodiche di impiego dei meccanismi devono essere intese globalmente per il sistema nel suo complesso.

8.3 LE SITUAZIONI DI STALLO

I primi metodi di allocazione delle risorse presi in considerazione si riferiscono al problema delle situazioni di stallo, descritte nel paragrafo 3.2.
Se le risorse vengono assegnate ai processi esclusivamente sulla base della loro disponibilità, è facile che si verifichino situazioni di stallo. Nella sua forma più semplice, una situazione di stallo avviene nelle seguenti circostanze: il processo A, cui è concessa la risorsa X, in seguito richiede anche la risorsa Y; il processo B, cui è assegnata la risorsa Y, in seguito richiede anche la risorsa X. Se entrambe le risorse non sono condivisibili e se nessun processo rilascia la risorsa impegnata, sorge una situazione di stallo. In generale, le condizioni necessarie e sufficienti per una situazione di stallo (Coffman et al., 1971) sono le seguenti:

1. le risorse interessate non sono condivisibili;
2. i processi impegnano le risorse già loro assegnate, mentre ne attendono di nuove;
3. le risorse non possono essere interdette durante l'uso;
4. esiste una catena circolare di processi tale che ogni processo impegna le risorse che al momento vengono richieste da parte del processo che segue nella catena.

Si può risolvere il problema delle situazioni di stallo adottando una delle seguenti strategie:

1. prevenire situazioni di stallo garantendo sempre che almeno una delle quattro condizioni precedenti non sia valida;
2. rilevare situazioni di stallo quando si verificano e tentare quindi di porvi rimedio;
3. evitare situazioni di stallo tramite opportuni interventi preventivi.

Prendiamole in considerazione una alla volta.

(1) Tecniche di prevenzione delle situazioni di stallo
Per prevenire le situazioni di stallo bisogna invalidare almeno una delle quattro condizioni necessarie e sufficienti esposte in precedenza.
La condizione (1) è difficile da invalidare, poiché alcune risorse (per esempio un lettore di schede o un file modificabile) per loro stessa natura non sono condivisibili. Tuttavia, l'impiego della funzione di spool (vedi paragrafo 6.6) può eliminare potenziali situazioni di stallo insite nelle periferiche non condivisibili.
La condizione (2) può essere invalidata stabilendo che i processi richiedano tutte assieme le risorse loro necessarie e che non possano evolversi fino a che tutte le richieste non siano state soddisfatte. Tale accorgimento presenta lo svantaggio che risorse utilizzate solo per poco tempo vengono, ciò nondimeno, allocate e diventano, pertanto, inaccessibili per lunghi periodi. È comunque facile da implementare e, a lunga scadenza, può rivelarsi più economico di certi algoritmi maggiormente complessi.
La condizione (3) viene invalidata semplicemente, imponendo la regola che, se ad un processo viene respinta la richiesta di utilizzo di una risorsa, questo deve rilasciare tutte le risorse impegnate al momento e, se necessario, richiederle nuovamente assieme a quelle aggiuntive. Purtroppo, questa strategia draconiana all'atto pratico può dimostrarsi poco funzionale, in quanto l'interdizione, poniamo, di una stampante parallela potrebbe dar luogo all'interfoliazione dell'output proveniente da più lavori. Oltre a ciò, anche se una risorsa può essere interdetta in modo conveniente, si può incorrere in un appesantimento piuttosto considerevole dovuto alla memorizzazione ed al ripristino del suo stato al momento dell'interdizione. Nel caso di una sola CPU, comunque, l'appesantimento del sistema è relativamente limitato (più precisamente, alla memorizzazione dell'ambiente volatile del processo corrente) e l'interdizione del processore si può considerare sempre possibile.
Si può effettuare l'invalidazione della condizione (4) imponendo ai vari tipi di risorse un ordinamento tale per cui, se ad un processo sono state assegnate risorse del tipo k, può richiedere solo quelle risorse corrispondenti ai tipi successivi a k. In questo modo si assicura che non si verifichi mai una condizione di attesa in circolo (circular wait). Lo svantaggio relativo, però, è la restrizione imposta alla sequenza naturale delle richieste di risorse, benché sia possibile moderare tale effetto negativo mettendo le risorse di uso più comune in posizione più avanzata all'interno della sequenza.
Un esempio interessante inerente a questi tre metodi di prevenzione delle situazioni di stallo è presente nell'OS/360 (Havender, 1968). In questo sistema i lavori sono suddivisi in passi (job steps) e una routine di inizializzazione dei passi stessi è responsabile dell'ottenimento delle risorse necessarie ad ogni passo. L'attivazione concorrente di più passi relativi a lavori diversi è ottenuta tenendo a disposizione varie copie della routine d'inizializzazione. Si previene l'insorgere di situazioni di stallo tra le copie della routine garantendo che ognuna di esse acquisisca le risorse sempre nel seguente ordine: file, memoria, periferiche. Si previene d'altra parte l'insorgere di situazioni di stallo tra i diversi passi di lavoro che intendono accedere agli stessi file facendo sì che la routine di inizializzazione acquisisca in una volta sola tutti i file necessari ad un intero lavoro. Si noti che in questo caso l'interdizione non sarebbe opportuna, in quanto un lavoro non vorrebbe vedersi manomettere i propri file da parte di passi ad esso estranei, mentre si trova esso stesso in un qualche passo intermedio tra l'inizio ed il suo completamento. Non è nemmeno possibile stabilire una sequenza sensata per le richieste d'accesso ai file. Si previene l'insorgere di situazioni di stallo per quanto riguarda l'allocazione delle periferiche costringendo i job a rilasciare tutte le risorse, tranne il file, dopo ogni passo ed a richiederle nuovamente per il passo successivo, se necessario.

(2) Tecniche di riconoscimento e recupero delle situazioni di stallo

Se si ritengono troppo restrittive le metodiche di prevenzione esposte dettagliatamente in precedenza, può forse dimostrarsi soddisfacente un'impostazione alternativa, la quale consenta sì l'eventuale verificarsi di situazioni di stallo, ma faccia affidamento sulla loro individuazione e sulla capacità di promuoverne il recupero. La velocità di tale impostazione è funzione della frequenza con cui insorgono situazioni di stallo e del tipo di recupero che è possibile effettuare.

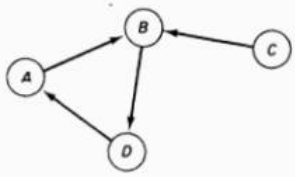


Fig. 8.1: Esempio di grafico di stato (le risorse A, B e D sono interessate da una situazione di stallo)

Gli algoritmi di riconoscimento si basano sul rilevamento delle situazioni di attesa in circolo espresse dalla precedente condizione (4). Si può rappresentare lo stato del sistema in ogni momento dato tramite un grafico di stato, i cui nodi indicano le risorse ed un eventuale arco tra i nodi A e B comporta l'esistenza di un processo che impegna la risorsa A, mentre richiede nel contempo la risorsa B (vedi figura 8.1).

Nel grafico di stato la condizione di attesa in circolo è rappresentata da una curva chiusa (nell'esempio di figura 8.1, quella che unisce A, B e D). L'algoritmo di riconoscimento mantiene una rappresentazione del grafico di stato nella maniera più opportuna e la controlla di tanto in tanto per verificare l'esistenza di curve chiuse. Il controllo può avvenire a seguito di ogni allocazione oppure, poiché l'appesantimento di questa operazione probabilmente risulterebbe troppo oneroso, ad intervalli prefissati

Il riconoscimento di situazioni di stallo è utile solo se si può effettuare un tentativo soddisfacente per recuperarle. La definizione di "soddisfacente" può essere ampliata a seconda delle circostanze, includendovi una qualsiasi delle tecniche di seguito elencate in ordine di complessità crescente.

a. interruzione di tutti i processi in situazioni di stallo, il che equivale all'incirca a ciò che succede negli uffici postali dove, di tanto in tanto, in presenza di un sovraccarico di lavoro, si brucia un sacco di corrispondenza; questo è però il metodo adottato nella maggioranza dei sistemi universali.
b. Riattivazione dei processi in situazione di stallo a partire da qualche punto di controllo intermedio, sempre che esista. Se si applica questo metodo in modo semplicistico, però, può ripresentarsi tale e quale la situazione originaria di stallo, anche se la non-determinatezza del sistema solitamente garantisce che ciò non avvenga.
c. Interruzione successiva di tutti i processi in situazioni di stallo fino a che queste ultime non sussistano più. L'ordine delle interruzioni può essere tale da minimizzare i costi relativi alla perdita dell'investimento fatto in termini dì risorse già utilizzate. Tale metodica comporta il richiamo dell'algoritmo di riconoscimento a seguito di ogni singola interruzione, per verificare se persistono ancora situazioni di stallo.
d. Interdizione successiva delle risorse dai processi in situazioni di stallo, fino a che queste ultime non sussistano più. Come nel punto (c), l'ordine delle interdizioni può essere tale da minimizzare qualche funzione di costo. Il richiamo dell'algoritmo di riconoscimento è necessario dopo ogni interdizione. Un processo che ha subito l'interdizione di una risorsa deve effettuare successivamente una richiesta per la riallocazione della stessa.

Vale la pena osservare, prima di abbandonare l'argomento, che il riconoscimento di situazioni di stallo spesso viene affidato agli operatori di sistema, anziché essere eseguito dal sistema stesso. Un operatore dotato di spirito d'osservazione noterà alla fine che certi processi sembrano essere bloccati e, ad un più attento esame, si renderà conto che si è verificata una situazione di stallo. L'intervento consueto volto al recupero di tale condizione consiste nell'interruzione e nella riattivazione (se possibile) dei processi in stallo.

(3) Tecniche per evitare situazioni di stallo

Con questa definizione intendiamo l'impiego di un algoritmo che avverta anticipatamente del probabile verificarsi di una situazione di stallo e che, pertanto, respinga una richiesta di risorse, altrimenti accettata. Questo concetto è diverso da quello di prevenzione, la quale garantisce aprioristicamente l'impossibilità del verificarsi di situazioni di stallo, invalidando una delle condizioni necessarie.

Un interessante criterio di impostazione del problema, mirante all'ideazione di un algoritmo volto ad evitare situazioni di stallo, è il seguente. Prima dell'accettazione di una richiesta d'utilizzo di risorse, si provi sperimentalmente a cambiare il grafico di stato per vedere cosa accadrebbe se la richiesta fosse accettata, applicando quindi un algoritmo di riconoscimento delle situazioni di stallo. Se l'algoritmo dà il "cessato allarme", la richiesta viene accolta; in caso contrario, viene respinta ed il grafico di stato viene riportato al suo aspetto originario. Purtroppo, questa tecnica non è sempre efficace, in quanto si basa sull'ipotesi che, se un'allocazione porta ad una situazione di stallo, questa si verificherà immediatamente. Che quest'ipotesi sia falsa lo si può dimostrare tramite lo studio della figura 8.2 (dovuto a Dijkstra), la quale illustra un sistema fondato su due processi e due risorse.

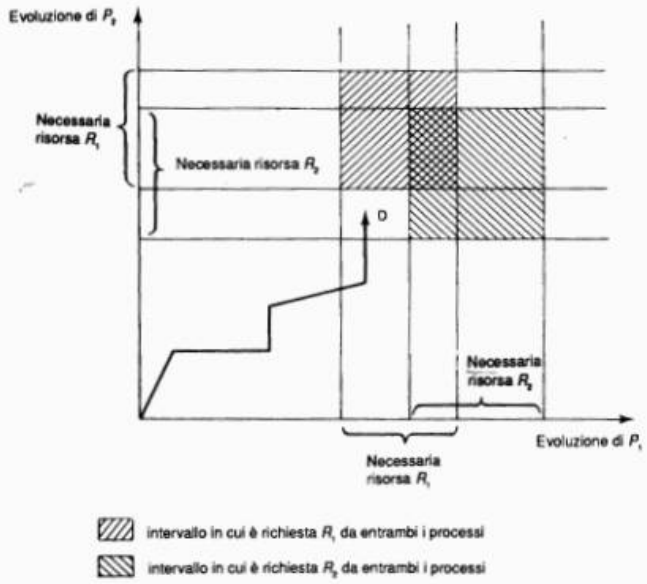


Fig. 8.2: Illustrazione di una situazione di stallo

L'evoluzione contemporanea dei processi P, e P2 si rappresenta tracciando una traiettoria, che nel diagramma è indicata con la spezzata in neretto. I segmenti orizzontali della traiettoria rappresentano i periodi in cui è in esecuzione P1, mentre i segmenti inclinati rappresentano quei periodi in cui sono in esecuzione simultanea sia P1 che P2. (In una configurazione ad una sola CPU potranno esistere solamente segmenti verticall ed orizzontali.) La traiettoria viene condotta in modo da

non entrare nell'area tratteggiata del diagramma, poiché si tratta di un intervallo in cui è richiesta da entrambi i processi almeno una delle risorse, R1 ed R2. Il nocciolo della questione è che, se la traiettoria entra nell'area D, è inevitabile finire in una situazione di stallo, poiché in quel caso è impossibile far sì che la traiettoria eviti di incontrare la zona tratteggiata, in quanto si può muovere solo verso l'alto e verso destra (cioè nei due sensi corrispondenti all'evoluzione positiva dei processi). Situazioni di stallo, tuttavia, non esistono di per sé nell'area D e, quindi, l'entrarvi non solleverebbe obiezioni da parte di nessuno fra gli algoritmi di riconoscimento delle situazioni di stallo. In questo caso, pertanto, l'impiego di uno di tali algoritmi non eviterebbe il verificarsi di una situazione di stallo. Per esprimere la questione in altri termini, un algoritmo di riconoscimento riesce a prevedere solo eventi relativi al passo immediatamente successivo, mentre situazioni di stallo possono essere inevitabili anche parecchi passi più avanti.
Risulta chiaro da quanto detto finora che un algoritmo volto ad evitare situazioni di stallo, per essere efficace, deve avere in anticipo qualche cognizione sul possibile schema evolutivo degli eventi futuri, così da essere in grado di distinguere situazioni in cui è insito un potenziale di stallo, già prima del verificarsi del fenomeno. il tipo di conoscenza a priori rappresenta la caratteristica distintiva dei vari algoritmi. Di seguito prendiamo in esame quello noto come banker's algorithm (algoritmo del banchiere) che, con tutta probabilità, è il più conosciuto.
Il tipo di conoscenza a priori richiesta dal banker's algorithm riguarda la disponibilità massima di ogni risorsa che richiederà un processo durante il suo ciclo di vita. Denominiamo questa quantità la claim (rivendicazione) di un processo su una risorsa. Nei sistemi di tipo batch, tale claim è piuttosto razionale, in quanto si può far uso delle descrizioni dei job per specificare anticipatamente il fabbisogno massimo di risorse. L'algoritmo accoglie una richiesta di risorse solo se:

- la richiesta, sommata alla quantità relativa all'uso corrente, è inferiore alla claim;
- in seguito a ciò, sussiste una sequenza secondo la quale tutti i processi sono in grado di essere eseguiti fino al completamento, anche se tutti avanzano le relative claim per intero.

Prima di effettuare un'allocazione, pertanto, l'algoritmo verifica che in seguito alla stessa rimangano risorse a sufficienza per consentire la costruzione di una simile sequenza. È stato dimostrato (Habermann, 1969) che l'attività concernente l'effettuazione delle verifiche è proporzionale al quadrato del numero dei processi esistenti nel sistema.
Per esemplificare la modalità operativa dell'algoritmo si prenda nuovamente in considerazione la fig. 8.2. Le claim di P1 e P2 sono costituite per entrambi da R1 ed R2. Prima dell'ingresso nell'area D, viene assegnata a P1 la risorsa R1, mentre R2 è disponibile. Esistono così risorse residue sufficienti ad eseguire fino al completamento sia P1 che P2, anche se entrambi richiedessero per intero le relative claim. La sequenza più idonea in tal caso consiste in P1 seguito da P2. All'interno dell'area D, R1 è destinata a P1 ed R2 a P2, cosicché, se entrambi i processi richiedessero per intero le rispettive claim, non potrebbe esistere nessuna sequenza (ne P1 prima di P2, ne viceversa) in base alla quale poterli eseguire fino al completamento. Il banker's algorithm, pertanto, evita l'ingresso nell'area D respingendo la richiesta di utilizzo di R2 da parte di P2. Quest'ultimo verrà sospeso e soltanto P1 potrà essere eseguito. La traiettoria seguirà, pertanto, un andamento orizzontale lungo il bordo inferiore dell'area tratteggiata fino a che P1 rilascia R1. A questo punto può essere soddisfatta la richiesta relativa ad R2 avanzata originariamente da P2, in quanto vi sono risorse sufficienti a disposizione per portare a termine ambedue i processi.
Si deve notare che il banker's algorithm ammette sempre il peggior caso possibile, quello in cui tutti i processi avanzano interamente le rispettive claim. A volte, pertanto, può respingere una richiesta, perché potrebbe sorgere una situazione di stallo, anche se di fatto non se ne sarebbe verificata nessuna nel caso dell'eventuale concessione della richiesta. Continuando sull'esempio di figura 8.2, si osserva che l'ingresso nell'area D è vietato a causa dell'eventualità che P1 possa esercitare la sua claim su R2 proprio quando P2 esercita la sua su R1. Se di fatto nessuno dei due processi scegliesse di avanzare la rispettiva claim in un momento rischioso, l'area D risulterebbe del tutto sicura (in quanto le aree tratteggiate non occuperebbero più gli spazi indicati) e la mancata concessione a P2 della richiesta di utilizzo di R2 si rivelerebbe una limitazione superflua. I principali inconvenienti relativi all'impiego di questo algoritmo sono dati dalla sua tendenza all'eccessiva prudenza, nonché dall'appesantimento del sistema conseguente alla sua applicazione.
Concludiamo la nostra trattazione inerente le situazioni di stallo, prendendo in considerazione il livello del sistema operativo al quale si dovrebbero implementare le metodiche per la loro gestione.
Nel caso si faccia uso di una strategia volta alla prevenzione, tutte le parti del sistema destinate all'allocazione delle risorse sottostanno ad una o più delle limitazioni descritte in precedenza, le quali si possono implementare come elementi dei meccanismi di allocazione delle risorse ai livelli più idonei.
Se si fa uso dei metodi di riconoscimento, non è necessaria alcuna modifica dei meccanismi di allocazione. L'algoritmo di riconoscimento che deve poter accedere a tutte le strutture dati concernenti l'allocazione, può essere implementato ad alto livello, se possibile nello stesso ambito del gestore dei processi (descritto nei paragrafi successivi).
Nel caso si miri ad evitare situazioni di stallo, tutte le richieste di risorse devono essere sottoposte a verifica da parte dell'algoritmo relativo. Nel prossimo paragrafo vedremo come sia possibile effettuare tutte le richieste di risorse attraverso il gestore dei processi, il quale diviene così la sede naturale per l'implementazione dell'algoritmo.

8.4 IL GESTORE DEI PROCESSI

Con il termine schedulazione in generale si contemplano le questioni inerenti la determinazione dei tempi di introduzione dei nuovi processi nel sistema e l'ordine di esecuzione che dovrebbero seguire quelli esistenti. Come già detto in precedenza, questi argomenti sono intimamente connessi all'allocazione delle risorse. Infatti, le decisioni riguardanti la schedulazione e quelle relative all'allocazione delle risorse sono correlate in maniera tanto stretta da rendere spesso consigliabile l'affidamento della loro messa in pratica ad un unico processo di sistema. Chiameremo tale processo il gestore dei processi e daremo di seguito una descrizione dei suoi compiti.

(1) Introduzione di nuovi processi
In un sistema di tipo batch, i job in attesa di esecuzione vengono immagazzinati in un pool dei lavori che è tenuto su memoria di massa. Il gestore dei processi avvia l'esecuzione di un job dando inizio al processo opportuno, quale la compilazione. La scelta relativa al lavoro successivo da avviare dipende dalle risorse richieste da ogni singolo job (in base a quanto definito nella relativa descrizione) nonché dallo schema attuale complessivo di allocazione delle risorse entro il sistema. Al fine di raggiungere una produttività elevata, il gestore dei processi dovrebbe avviare quelli nuovi non appena lo consente la disponibilità di risorse.
In un sistema ad accesso multiplo, i processi vengono creati nel momento in cui gli utenti effettuano il login. Ogni nuovo utente rende maggiore la domanda complessiva di risorse e l'accesso può essere respinto quando il numero di utenti collegati al sistema è tale da aumentare il tempo di risposta fino a limiti non più tollerabili.

(2) Assegnazione delle priorità ai processi
L'ordine concernente l'esecuzione dei vari processi viene determinato sia dalla sequenza presente nella coda d'attesa al processore sia dalla successione con cui la routine di smistamento seleziona per il prelievo i processi della coda. Nel

capitolo 4 abbiamo suggerito che, per contenere al massimo gli appesantimenti dovuti allo smistamento, la routine che si occupa di tale attività dovrebbe prelevare dalla coda sempre il primo processo avente i requisiti necessari; l'ordine della coda diviene così implicitamente il fattore determinante. Il gestore dei processi è responsabile dell'assegnazione delle priorità ai processi, così da inserirli in coda nella posizione più opportuna allorché vengono messi in stato di pronto. Prenderemo in esame gli algoritmi che determinano le opportune priorità nel prossimo capitolo.

(3) Implementazione dei metodi di allocazione delle risorse

I metodi cui si fa riferimento in questa sede sono quelli relativi a come evitare situazioni di stallo e bilanciare il sistema: come garantire, cioè, che nessun genere di risorsa sia né sottoutilizzata, né sovrautilizzata. i criteri per la valutazione del bilanciamento del sistema saranno esaminati nel paragrafo che segue.

Poiché il comportamento del sistema è determinato in larga misura dalle attività del gestore dei processi, è importante che quest'ultimo abbia una priorità più alta rispetto agli altri processi. Se il gestore dei processi possiede la priorità massima, la routine di smistamento lo manderà in esecuzione ogniqualvolta si trova in stato di pronto, dandogli la precedenza rispetto agli altri processi. Ciò assicura che il sistema reagisca prontamente al mutare delle situazioni, in particolare ai cambiamenti della domanda complessiva di risorse. Gli eventi che possono causare l'attivazione del gestore dei processi sono le seguenti:

- viene richiesta una risorsa;
- viene rilasciata una risorsa;
- termina un processo;
- giunge nel pool dei lavori in attesa d'esecuzione un nuovo lavoro (oppure un nuovo utente prova ad effettuare il login).

È istruttivo notare l'analogia esistente tra gli interrupt e gli eventi di schedulazione precedentemente riportati. Questi fenomeni si verificano ad intervalli imprevedibili ed entrambi possono provocare un cambiamento nel comportamento del sistema. Nel caso degli eventi di schedulazione, le modifiche avvengono ad alto livello, influenzando parametri globali, quali le priorità dei processi ed il loro numero entro il sistema. Le modifiche dovute agli interrupt, invece, si verificano a basso livello.

Gli algoritmi di schedulazione cambiano a seconda dell'aspetto del sistema operativo che si vuole privilegiare. In un ambiente di tipo batch l'aspetto da privilegiare potrebbe consistere nella produttività complessiva o nel tempo medio d'esecuzione per una certa categoria di lavori; in un sistema ad accesso multiplo potrebbe, invece, essere data dal tempo medio di risposta agli utenti o dal tempo di risposta presente per certi tipi di interazione.

Per cominciare, prendiamo in considerazione gli algoritmi basati sul presupposto che le CPU siano risorse di primaria importanza. Abbiamo già detto implicitamente all'inizio del presente capitolo che in molti casi tale ipotesi risulta falsa e che anche altre risorse possono essere molto importanti. Comunque, studiando i sistemi limitati dal processore possiamo riuscire a vedere a fondo anche in quei casi più complessi che esamineremo oltre.

Un modello generico di sistema limitato dalla CPU è illustrato in figura 8.3. I nuovi processi giungono nella coda d'attesa al processore e vengono gestiti da svariati processori (avanziamo l'ipotesi che siano identici). Dopo essere stato in certa misura soggetto ad interventi gestionali, un processo potrebbe essere completato, nel qual caso lascia il sistema; in caso contrario, potrebbe essere reimmesso nella coda per ottenere di venir ripreso in una fase successiva. Il modello rappresenta solamente quei processi che si trovano in stato di pronto ed in corso d'esecuzione: l'assenza dei processi in stato di blocco (a tale proposito si veda il capitolo 4) rispecchia il fatto che, in un sistema limitato dalla CPU, gli unici ritardi significativi eventualmente sono dovuti all'attesa nella coda al processore. Ogni particolare algoritmo di schedulazione è caratterizzato dall'ordine che hanno i processi in coda d'attesa e dalle condizioni in cui vi vengono reimmessi. Di seguito prenderemo in considerazione vari algoritmi tra i più noti.

1. Il primo lavoro è il più breve

Come rivela il nome stesso, l'ordine della coda d'attesa è disposto in base al tempo di esecuzione richiesto da ogni processo. Questo algoritmo è idoneo solamente ai sistemi di tipo batch, nei quali si può ricavare una stima del tempo d'esecuzione derivandola dalla descrizione del job. Il suo scopo consiste nel contenere al massimo il tempo d'esecuzione (turnaround time) dei lavori più brevi e, pertanto, se ne consiglia l'impiego in quegli ambienti in cui il grosso dell'attività è formato da job limitati. Nella sua forma più semplice, questo algoritmo consente l'esecuzione di tutti i processi fino alloro completamento, anche se è possibile realizzarne una variante in cui un nuovo processo può interdire un processore nel caso il suo tempo d'esecuzione sia inferiore a quello necessario per portare a termine il processo corrente. Tale versione, conosciuta come pre-emptive shortest jobfirst, è ancora più favorevole ai job brevi. Un'altra modifica, che garantisce che i lavori più lunghi non vengano ritardati all'infinito, consiste nell'aumento graduale della priorità di un processo in base al suo tempo d'attesa in coda.

2. Round robin

Questo algoritmo è stato sviluppato quale metodo per fornire rapidamente risposta a richieste di gestione contenuta nel tempo, quando non siano noti in anticipo i tempi d'esecuzione. Ogni processo presente nel sistema gode di una determinata quantità di attenzione prima di essere reimmesso in coda, in ultima posizione. La coda è quindi circolare ed il suo ordine si basa sul tempo trascorso dall'ultima esecuzione, i processi che necessitano di gestioni di lunga durata rimarranno in coda e ripeteranno il ciclo varie volte prima di essere completati, mentre i processi con tempi d'esecuzione inferiori alla quantità prevista saranno portati a termine durante il primo ciclo. L'algoritmo round robin è stato usato nel primo sistema operante in time-sharing, il CTSS (Corbatò et al., 1962) ed è stato successivamente implementato in parecchie varianti su molti altri sistemi. La maggior parte delle varianti rispondono allo scopo di ridurre la tendenza del sistema a collassare se sottoposto a carico troppo consistente. L'analisi e l'esperienza mostrano che, se il carico si appesantisce eccessivamente rispetto ad una data capacità, le prestazioni del sistema si degradano immediatamente. Un modo per risolvere il problema consiste nell'aumentare la capacità con il crescere del numero dei processi. Ciò fa aumentare anche le probabilità di portare a termine un processo entro l'intervallo di tempo prestabilito, riducendo l'appesantimento dovuto alla commutazione dei processi.

3. Coda a due livelli

Si tratta di un altra variante dell'algoritmo round robin volta a superare il problema del degrado immediato dovuto al carico. I processi non completati entro un numero prefissato di cicli vengono dirottati su una coda secondaria che viene presa in considerazione solo se nel sistema non ci sono altri processi. La coda secondaria stessa può poi essere trattata sulla base dell'algoritmo round robin, però con un intervallo di tempo maggiore, oppure può operare sul principio FIFO. Il sistema a due livelli viene impiegato spesso negli ambienti misti operanti nel modo batch e ad accesso multiplo, dove i processi di tipo batch, essendo più lunghi, tendono a cedere la precedenza e vengono eseguiti solo quando non sono presenti processi avviati da terminale.

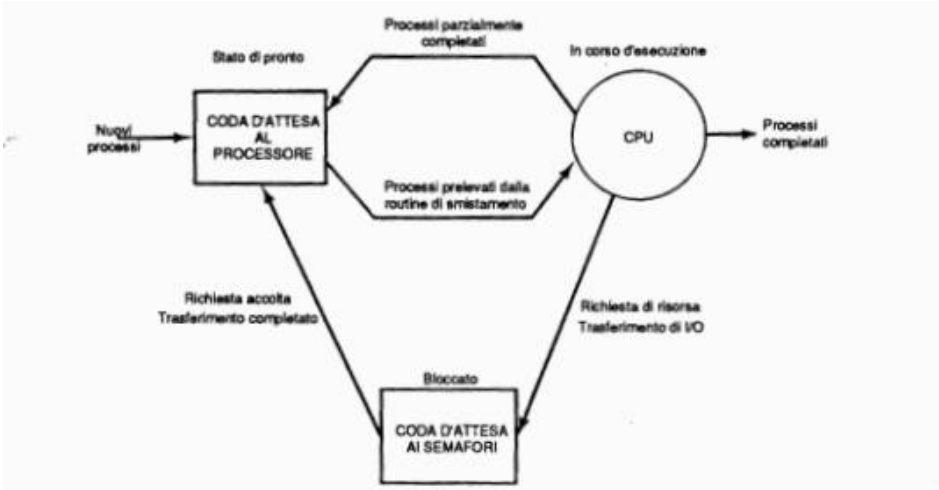


Fig. 8.4: Modello generico di schedulazione

Se si ritiene opportuno, l'algoritmo può essere generalizzato ed adattato ad un sistema a livelli multipli. Si ha un esempio di ciò nel DEC System-10 Monitor, in cui ci sono tre code basate sul principio round robin, con intervalli impostati rispettivamente su 0,02 secondi, 0,25 secondi e 2 secondi (valori che in realtà possono essere cambiati, se si preferisce).

È difficile determinare analiticamente i valori ottimali dei parametri dell'algoritmo round robin e dei relativi derivati, quali la misura massima di carico ed i tempi di inserimento dei processi in coda secondaria, anche se sono stati effettuati tentativi verso questa direzione (Coffman e Denning, 1973). Tramite simulazione sono ricavabili delle approssimazioni che si possono poi mettere a punto in base alle condizioni reali di funzionamento esperite.
Gli algoritmi riportati in precedenza costituiscono la base delle più diffuse metodiche di schedulazione. Si possono trovare altri algoritmi e varianti nel validissimo studio ad opera di Coffman e Kleinrock (1968). Una variante particolare degna di nota consiste nel consentire l'inserimento di priorità specifiche dall'esterno. Il risultato che ne consegue fa sì che i processi con priorità esterna più alta vengano gestiti in modo migliore di quanto accadrebbe con la forma pura di un algoritmo. Ciò mette gli utenti con scadenze strette o di rango superiore in condizioni di ottenere dal sistema una risposta migliore, benché per godere di questo privilegio possano subire maggiori addebiti.
i sistemi in cui le CPU non sono le sole risorse importanti si possono descrivere con l'ausilio del modello illustrato in figura 8.4, il quale differisce da quello di figura 8.3 per via dell'inclusione dello stato di blocco, durante il quale i processi sono in attesa di richieste di risorse o di trasferimento di I/O. L'aggiunta di questo stato aumenta enormemente la complessità del modello. In primo luogo, lo stato di blocco è caratterizzato non da un'unica coda, bensì da molteplici code d'attesa associate ai vari semafori che provocano l'insorgere del blocco. In secondo luogo, i processi passano dallo stato di blocco a quello di pronto a seguito di eventi, quali le operazioni di signal su semafori, che si verificano ad intervalli non prevedibili. In relazione alle diverse strategie, il movimento dei processi tra le code d'attesa si può studiare sia tramite analisi formale che con la simulazione (Coffman e Denning, 1973; Svobodova, 1976), nonostante la complessità del modello renda difficile applicare entrambe le tecniche.
Dalla figura 8.4 il lettore noterà che uno degli scopi di un algoritmo di schedulazione potrebbe consistere nel limitare al massimo il numero delle transizioni dallo stato di esecuzione a quello di blocco. Poiché le transizioni, però, se dovute ai trasferimenti di I/O, sono indipendenti da esso, ne risulta che l'algoritmo dovrebbe solamente rendere minimo il numero delle richieste di risorse respinte. Ciò comporta che si dovrebbero introdurre nuovi processi nel sistema solo se i loro probabili fabbisogni di risorse potranno essere soddisfatti tramite quelle disponibili in quel momento. È più facile implementare questo metodo nei sistemi di tipo batch, dove la domanda complessiva di risorse può essere specificata anticipatamente, rispetto ai sistemi ad accesso multiplo, nei quali le domande non sono praticamente prevedibili. Un esempio di implementazione in ambiente batch era il sistema operativo dell'Atlas, nel quale i job venivano divisi in tre classi: job corti, che richiedevano tempi brevi dì CPU e nessuna periferica esterna; job da nastro magnetico, il cui nome si spiega da sé; job lunghi, che comprendevano tutto il resto. Il gestore dei processi tentava di mantenere sempre nel sistema almeno un job per ogni classe.
È opportuno notare che metodi simili a questo non danno necessariamente luogo ad un'utilizzazione elevata delle risorse. Il motivo sta nel fatto che è improbabile che durante la sua vita un processo faccia uso di tutte quelle risorse che ha indicato. Le risorse oggetto di claim e concesse dal gestore dei processi possono restare, pertanto, inattive per periodi considerevoli. Se è richiesta un'utilizzazione più elevata, il gestore dei processi deve prendere decisioni di metodo circa l'allocazione delle risorse non solo al momento dell'introduzione di nuovi processi, ma anche al verificarsi di ciascuna richiesta o rilascio di risorsa, I criteri alla base di tale processo decisionale e dell'assegnazione delle priorità ai processi sono generalmente empirici. Ne prendiamo in esame alcuni qui di seguito.

1. Nel tentativo di accelerarne il completamento, si può dare una priorità alta a quei processi che detengono risorse numerose, rendendole così nuovamente disponibili ad essere sfruttate diversamente. il pericolo insito nell'applicazione di questa strategia consiste nel fatto che i job più consistenti possono monopolizzare la macchina o che gli utenti possono generare richieste fittizie di risorse al fine di ricevere un trattamento privilegiato. in un sistema batch si può evitare quest'ultimo rischio con la garanzia che verranno mandati in esecuzione i job con fabbisogni limitati di risorse, preferendoli agli altri.
2. Ogni volta che è possibile, si potrebbero accogliere ulteriori richieste provenienti da processi che detengono risorse già numerose. La giustificazione è analoga alla precedente ed anche qui si presenta il pericolo che i job più consistenti possano monopolizzare il sistema.
3. Nelle macchine dotate di paginazione, si può effettuare l'allocazione della memoria conformemente al principio dell'area di lavoro, descritto nel capitolo 5.
4. Ai processi del sistema operativo dovrebbero essere assegnate priorità che riflettono il carattere d'urgenza delle funzioni da essi svolte. La scala vede al primo posto lo stesso gestore dei processi, per finire con processi meno

impellenti, come quello che effettua il dump incrementale dei file. La maggior parte dei processi di sistema avrà così priorità maggiori rispetto ai processi degli utenti.

5. Come caso particolare del punto (4), i gestori delle periferiche dovrebbero avere priorità elevata. In generale, tanto più veloce è la periferica, tanto più alta dovrebbe essere la priorità del gestore relativo. Ciò assicura la maggiore occupazione possibile delle periferiche, riducendo conseguentemente il potenziale verificarsi di situazioni di ingorgo relative all'I/O. Se si impiega la funzione di spool, anche gli spooler dovrebbero avere una priorità alta.
6. Salvo che non siano previste misure relative alla prevenzione di situazioni di stallo, tutte le richieste di risorse dovrebbero essere assoggettate ad un algoritmo volto ad evitare situazioni di stallo.
7. L'appesantimento inerente il processo decisionale di schedulazione non dovrebbe essere sproporzionato rispetto ai vantaggi che ne derivano.

Si possono applicare le considerazioni suesposte per modificare uno degli algoritmi descritti in precedenza e volti alla schedulazione delle attività di CPU. Ne risulterà un algoritmo in cui le priorità dei processi sono aggiornate con continuità in base a come si configurano i fabbisogni di risorse e lo stato del sistema. È difficile poter fare una valutazione del rendimento di un tale algoritmo, se non tramite l'osservazione diretta; spesso lo si può migliorare considerevolmente apportando ai suoi parametri solo lievi cambiamenti. Un esempio interessante di questo fenomeno si è avuto sul DEC System-10 Monitor. In una delle prime versioni dell'algoritmo di schedulazione, i processi che rientravano nello stato di pronto, dopo essere stati bloccati in attesa di I/O da terminale, venivano messi alla fine della coda d'attesa al processore di primo livello. Successivamente tale procedimento fu modificato in modo da mettere i processi in questione alla sommità della coda d'attesa di primo livello. Il miglioramento della risposta del sistema agli utenti operanti in accesso multiplo risultò eccezionale.
Possiamo riassumere questo paragrafo dicendo che il numero dei possibili algoritmi di schedulazione è vastissimo, ma che al momento non abbiamo modo di scegliere l'algoritmo ottimale per un caso specifico, tranne che per mezzo della sperimentazione. L'analisi e la simulazione possono essere utili in certa misura, ma la loro validità è limitata a causa della complessità della maggioranza dei sistemi.

8.6 LE GERARCHIE DEI PROCESSI

Abbiamo visto nel paragrafo 8.4 che il gestore dei processi è responsabile dell'avvio dei nuovi processi. Per certi versi, quindi, il gestore dei processi rappresenta il padre di tutti i processi introdotti nel sistema. Inoltre, è responsabile del benessere della sua prole, in quanto presiede ai metodi d'allocazione delle risorse ed influisce sull'ordine secondo il quale vengono selezionati i processi dalla routine di smistamento. Questo ruolo paterno non deve essere per forza limitato al solo gestore dei processi; lo si può allargare anche ad altri processi, concedendo loro la possibilità di:

1. creare sottoprocessi;
2. allocare per i propri sottoprocessi un sottoinsieme delle proprie risorse (le quali vengono restituite al termine del sottoprocesso);
3. determinare la priorità relativa dei propri processori.

Lo scopo consiste nel consentire la creazione di una gerarchia dei processi, con il gestore dei processi alla sommità. La forma naturale della struttura dei processi non è più, quindi, la semplice lista esposta nel capitolo 4, bensì è costituita da un albero, come mostra la figura 8.5. Nell'albero si insinua in parte la coda d'attesa al processore, la quale collega tutti i nodi (processi) che correntemente si trovano in stato di pronto.

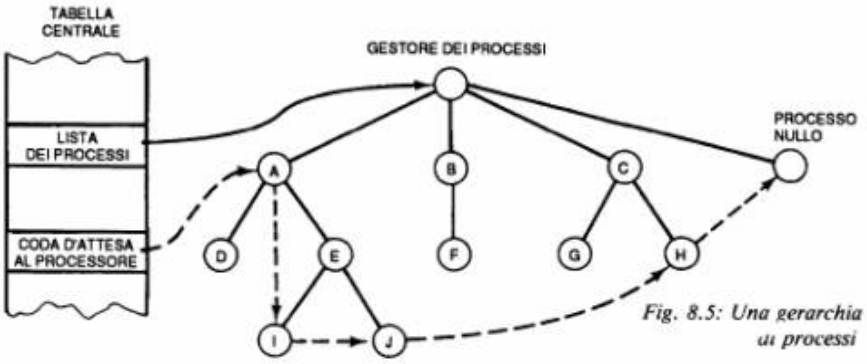


Fig. 8.5: Una gerarchia di processi

I vantaggi di un'organizzazione gerarchica sono di duplice natura. In primo luogo, consente ad un processo che svolge vari compiti indipendenti di creare un processo ausiliario per ogni singolo compito, rendendo possibile l'esecuzione parallela. Naturalmente, non si guadagna nulla nel caso di una macchina ad una sola CPU, in quanto si può ottenere solamente concorrenza virtuale; ma quando sono a disposizione svariati processori ne possono conseguire vantaggi reali.
Secondariamente, consente di far girare autonomamente ed in parallelo varie versioni del sistema operativo, ciascuna finalizzata ad una diversa applicazione. Ciascuna delle diverse versioni, o sottosistemi, si baserà sulle prestazioni fornite dal nucleo del sistema (vedi capitolo 4), ma può servirsi di algoritmi diversi per l'implementazione delle funzioni di sistema di più alto livello che compaiono negli strati esterni. Ogni sottosistema rappresenta così una versione particolare della nostra “cipolla”, benché tutti i sottosistemi abbiano lo stesso nucleo. Ciascuno di essi implementa una macchina virtuale idonea ad una particolare categoria di utenti e tutte le macchine virtuali coesistono nella stessa configurazione fisica.
A titolo d'esempio, si consideri la figura 8.6 che illustra una situazione con due sottosistemi diversi, responsabili rispettivamente delle attività di tipo batch ed in linea. Il loro rendimento subisce l'influenza delle risorse e le priorità relative sono assegnate dal gestore dei processi; fatte salve queste restrizioni, ogni sottosistema può controllare i processi ad esso subordinati in base agli algoritmi più opportuni. Nell'ambito del sottosistema di tipo batch, ad esempio, tutto l'I/O potrebbe essere svolto tramite spool, con i processi degli utenti ordinati in base al principio del job più breve. All'interno del sottosistema in linea, invece, l'I/O potrebbe avvenire senza spool ed i processi potrebbero essere ordinati in base all'algoritmo round robin. Si può immaginare un ampliamento della gerarchia nella quale, ad esempio, il sottosistema di tipo batch sia suddiviso in ulteriori sottosistemi adibiti alla gestione di diverse categorie di job.

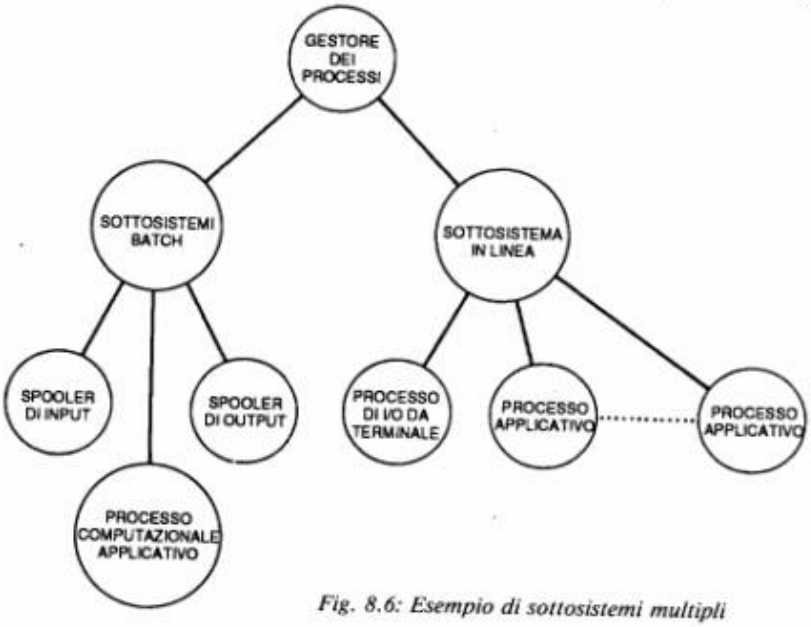


Fig. 8.6: Esempio di sottosistemi multipli

Un vantaggio della soluzione basata sui sottosistemi consiste nella coesistenza di diverse macchine virtuali. Un altro deriva dalla capacità di sviluppare nuove versioni di un sistema operativo mentre la vecchia versione gira in parallelo. A fronte di tutto ciò bisogna porre la perdita del controllo globale sull'allocazione delle risorse, conseguenza del modo in cui le stesse vengono spostate su e giù nella struttura dei processi. Le risorse allocate per un sottosistema o per un processo subordinato, e che non vengono usate, non possono essere trasferite altrove nel sistema, a meno che non vengano prima restituite al progenitore comune. Poiché i processi hanno la facoltà di assegnare a proprio piacimento sottoinsiemi delle loro risorse, non vi è garanzia che lo schema generale dell'utilizzazione delle stesse sia comunque ottimale.
Inoltre, anche l'assegnazione delle priorità da parte di un processo ai propri figli può creare dei problemi. Salvo che non siano imposte restrizioni, è possibile che un processo ottenga slealmente vantaggi inerenti la schedulazione delegando il proprio compito ad un sottoprocesso cui assegni una priorità molto alta. Un tipo di limitazione volta a risolvere tale problema consiste nel far sì che tutte le priorità assegnate siano relative a quella del processo d'origine e che il campo delle possibili variazioni per le priorità relative diminuisca successivamente mano a mano che si passa ai livelli inferiori della gerarchia. Per esempio, supponiamo che nella figura 8.5, al massimo livello, l'intervallo delle priorità relative vada da 0 a 99, mentre al secondo livello vari da 0 a 9, e così via. Quindi, se la priorità del processo B è 20, nessun sottoprocesso di B può avere una priorità che vada oltre l'intervallo 20 ± 9. Ciò comporta che, se la priorità del processo A è superiore a 29, suddividendosi in sottoprocessi, B non è in grado di ottenere nessun vantaggio rispetto ad A. Se la priorità di A è maggiore di 39, a tutti i suoi sottoprocessi sarà garantita la precedenza rispetto ai sotto-processi di B.
Nella situazione illustrata in figura 8.6, le priorità relative, assegnate dal gestore dei processi ai sottosistemi per attività di tipo batch ed in linea, determinano effettivamente la risposta data dal sistema per le due categorie di lavori. Una differenza notevole di priorità tra i due sottosistemi assicurerà che una categoria di utenza abbia sempre la precedenza sull'altra, mentre una differenziazione limitata darà luogo a competizione per l'utilizzo della (o delle) CPU su basi più paritetiche. La priorità relativa dei processi nell'ambito di ciascun sottosistema, come già detto, è di pertinenza del sottosistema interessato.
Uno dei primi sistemi operativi con una struttura gerarchica dei processi lo ritroviamo sull'elaboratore RC-4000 (Hansen, 1970). 11 suo nucleo veniva denominato Monitor ed alla sommità della gerarchia si trovava quello che veniva chiamato il Basic Operating System, corrispondente al nostro gestore dei processi. La struttura dei processi poteva comprenderne fino a 23 (la limitatezza era dovuta a ragioni di spazio) e le risorse venivano trasferite dai processi padre a quelli derivati, così come descritto in precedenza. Una differenza tra il sistema RC-4000 e quello da noi delineato consiste nel fatto che i processi progenitori non potevano assegnare priorità ai propri figli; tutti i processi avevano uguale priorità, se si esclude il fatto che potevano essere selezionati dalla routine di smistamento sulla base di un algoritmo round robin.
Una gerarchia limitata è implementata anche sul sistema operativo IBM VM/370 (VM sta per il termine inglese corrispondente a macchina virtuale). La lista in questo caso è limitata a 3 livelli: alla base vi è proprio il VM/370; al livello successivo si può trovare uno qualsiasi dei vari altri sistemi operativi standard dell'IBM (che fungono da sottosistemi); mentre al livello più basso vi sono i task degli utenti. I sistemi operativi standard del secondo livello presentano varie macchine virtuali sulla stessa configurazione. La routine di smistamento opera in base ad un algoritmo round robin a due livelli che può essere modificato assegnando delle priorità relative a ciascun sottosistema.
L'implementazione di una struttura gerarchica dei processi richiede la disponibilità delle seguenti procedure di sistema:

- crea processo (nome, priorità relativa)
- cancella processo (nome)
- avvia processo (nome)
- arresta processo (nome)

La funzione di crea processo consiste nell'impostare un descrittore per il nuovo processo e nell'aggregarlo alla struttura dei processi. Avvia processo aggiunge il descrittore di processo alla coda d'attesa al processore, cioè provvede a contraddistinguere il processo in quanto dotato dei requisiti per essere messo in esecuzione. Arresta processo e cancella processo svolgono invece funzioni inverse e consentono ad un processo padre di controllare i propri processi figli, i quali naturalmente possono terminare spontaneamente, nel qual caso l'impiego di arresta processo diventa superfluo. Tutte le quattro procedure manipolano i descrittori e le code d'attesa dei processi e, pertanto, dovrebbero essere implementate all'interno del nucleo del sistema. Le procedure richiedi risorsa e rilascia risorsa, descritte nel paragrafo 8.4, vengono modificate in modo da richiamare il processo padre invece del gestore delle risorse. Il processo padre diviene così responsabile delle allocazioni per i suoi derivati.

8.7 CONTROLLO E CONTABILITÀ DELLE RISORSE

Nei paragrafi precedenti ci siamo occupati dell'allocazione delle risorse sulla base di una scala cronologica paragonabile ai tempi d'esecuzione dei lavori. Ora volgiamo la nostra attenzione agli aspetti a più lungo termine riguardanti l'allocazione delle risorse, cioè a questioni quali l'addebito relativo all'utilizzo delle risorse, la valutazione e l'influsso sull'articolarsi delle richieste e la garanzia che tutti gli utenti ottengano una quota adeguata delle risorse di calcolo disponibili. Tali questioni solitamente vengono definite con il nome di controllo delle risorse. Nel senso stretto del termine, il controllo delle risorse riguarda più il responsabile dell'elaboratore che il progettista del sistema operativo, ma il primo può far ben poco se il secondo non prevede gli strumenti opportuni. In particolare, il progettista deve fornire strumenti atti alla misurazione, poiché senza di questa non è possibile alcun controllo.
Gli obiettivi del controllo delle risorse dipendono dalla configurazione interessata. Parlando in generale, le configurazioni rientrano in due categorie: per conto terzi e private. Lo scopo di un sistema che svolge lavoro per conto terzi, come le macchine dei centri di elaborazione dati, consiste nel fornire servizi di calcolo a svariati utenti dietro il pagamento di un corrispettivo. Sull'utenza non sono imposte limitazioni, a parte il fatto che i servizi ottenuti vanno pagati: anzi, è nell'interesse del proprietario vendere il maggior numero possibile di servizi al più vasto numero possibile di utenti. Il controllo delle risorse, pertanto, si riduce alla contabilità, ovvero alla registrazione delle quantità di ogni risorsa addebitabile effettivamente utilizzata ed al conseguente addebito ai clienti.
D'altra parte, una configurazione privata è di proprietà di una singola società, università o altro ente e fornisce servizi soltanto a chi ne fa parte. A causa dei vincoli imposti al bilancio, l'amministratore si trova ad affrontare il problema riguardante il soddisfacimento di una domanda di servizi che quasi sempre supera la capacità delle configurazioni. Contrariamente al centro di elaborazione dati, non si trae nessun profitto dalla fornitura dei servizi e, quindi, solitamente non è aperta la via che porta all'aumento delle potenzialità della macchina, al fine di far fronte alla domanda. L'amministratore dovrà così razionare le risorse disponibili in base all'importanza o ai fabbisogni relativi ai diversi utenti, in tal caso il controllo delle risorse rappresenta uno strumento per differenziare gli utenti: deve adempiere alla duplice funzione di contabilizzare le risorse, garantendo che nessun utente superi la propria quota.
Dovrebbe risultare chiaro che i meccanismi di controllo necessari per le configurazioni adibite al servizio per conto terzi sono un sottoinsieme di quelle necessarie per le configurazioni private. Pertanto, concentreremo la nostra attenzione su queste ultime.

(1) Restrizione dell'accesso
L'accesso a quei servizi di sistema che comportano un utilizzo notevole di risorse può essere negato a determinate categorie di utenza, quali i programmatori più giovani o gli studenti. D'altronde, si potrebbe limitare l'attività di utenti operanti in accesso multiplo all'editing ed a prove su piccola scala, mentre agli utenti batch potrebbe essere concesso di spaziare per l'intera gamma delle risorse.

(2) Razionamento
Un metodo di razionamento può essere strutturato sia a lungo che a breve termine, oppure con una soluzione intermedia. il razionamento a breve termine applica un limite alla quantità di risorse, quali il tempo di CPU o lo spazio di memoria, che si possono utilizzare per un singolo lavoro. Nel razionamento a lungo termine viene destinata ad ogni utente una determinata riserva (budget) indicante la quantità di risorse di cui può fare uso entro un dato periodo, per esempio in una settimana o in un mese. Qualora l'utente tenti di andare oltre la propria quota si determina l'interruzione del lavoro o il mancato consenso all'accesso al sistema.

Entrambi questi metodi di controllo si possono implementare mantenendo un file di contabilità contenente le seguenti informazioni per ogni utente accreditato presso il sistema:

- L'identificazione dell'utente. Si tratta di un “numero di conto” citato dall'utente al momento del login o all'inizio della descrizione di un lavoro. Serve al sistema per identificare l'utente.
- Una sigla di riconoscimento nota solo al sistema ed all'utente, il quale la presenta a riprova della propria identità. (Normalmente nei sistemi di tipo batch non sono necessarie sigle di riconoscimento, in quanto è difficile mantenerne la riservatezza se sono perforate su schede o supporti analoghi.)
- Un “vettore di permessi” in cui ogni bit rappresenta il permesso ad usare una certa risorsa, il quale è accordato solo se il bit relativo è settato a 1.
- I limiti delle risorse per ogni lavoro.
- Il budget delle risorse per il periodo attuale di contabilizzazione.
- Le risorse ancora da utilizzare presenti nel budget per il periodo corrente.

I particolari applicativi del controllo delle risorse variano da sistema a sistema. Nei sistemi orientati verso applicazioni di tipo batch il saldo delle risorse residue disponibili viene addebitato alla fine di ciascun job ed i lavori sono accettati dal gestore dei processi solo se tale saldo è positivo. In alternativa, si può prevedere l'accettazione di un lavoro solo se i relativi fabbisogni dichiarati di risorse sono inferiori al saldo attuale; con ciò si impedisce che un utente superi la sua razione mettendo in esecuzione un lavoro di entità notevole quando il saldo delle risorse a sua disposizione è quasi nullo, In un sistema ad accesso multiplo il saldo può essere addebitato ad intervalli regolari durante una sessione di lavoro al terminale, costringendo l'utente al log-off (a seguito di un avviso in tal senso) nel caso in cui il suo saldo diventi nullo.
Si può applicare il principio del razionamento singolarmente per ciascuna risorsa oppure, come avviene più comunemente, l'utilizzo di ogni risorsa può essere calcolato in modo composto tramite una formula specifica di addebito, espresso in un tipo di unità riferita al consumo globale di risorse. In tal caso, il peso relativo assegnato ad ogni risorsa presente nella formula viene prescelto in modo da riflettere la scarsa rilevanza o l'importanza della risorsa stessa. Un esempio di una formula di addebito simile, presa dal DEC System-10 presso l'Università dell'Essex, è il seguente:

```
numero di unità utilizzate = (133,0P + 6,0C + 13,3K + 5,5W)
```

dove

- P = tempo di CPU in secondi
- C = tempo di collegamento del terminale in secondi
- K = occupazione di memoria, cioè l'intero relativo al numero dei blocchi di 1024 word utilizzati nell'intervallo del tempo di CPU
- W = numero di trasferimenti da disco

È opportuno osservare che i parametri di una formula d'addebito possono essere modificati al fine di influenzare il modo in cui lavorano gli utenti. Per esempio, un aumento considerevole dell'addebito per il tempo di collegamento del terminale

spingerebbe gli utenti a fare maggiormente uso delle risorse batch, mentre un aumento dell'addebito per l'occupazione di memoria potrebbe incentivare la riduzione delle dimensioni dei lavori.
Un'altra caratteristica importante dei metodi di razionamento consiste nel porre in relazione la quantità totale allocata per ogni risorsa con quella che il sistema riesce a fornire entro un dato periodo di tempo. Non è necessario che queste due quantità siano uguali, poiché vi sono buone probabilità che la maggior parte degli utenti non consumi tutte le proprie disponibilità; grazie all'esperienza si potrà determinare con buon margine di sicurezza l'entità di un'allocazione eventualmente superiore al dovuto. Nel caso delle formule di addebito globale è più difficile far combaciare le allocazioni con le disponibilità, in quanto, se tutti gli utenti decidessero di impiegare la propria quota di allocazione sotto forma della stessa risorsa, il sistema non riuscirebbe più a farvi fronte. In pratica, però, l'esistenza di un vasto numero di utenti rende improbabile questa evenienza.
Metodi di razionamento più raffinati (per esempio, Hartley, 1970) possono impiegare tassi d'addebito differenziati a seconda del momento della giornata. Ciò incoraggia l'uso della macchina negli orari meno richiesti e dà luogo ad una distribuzione più uniforme del carico di lavoro. Analogamente, gli addebiti possono variare in base alla priorità esterna richiesta da un utente per il proprio lavoro.
Compete al progettista di sistema fornire gli strumenti adeguati per l'implementazione di metodiche quali quelle appena descritte. Tra di essi si annoverano:

1. routine per la gestione del file di contabilizzazione;
2. meccanismi per misurare l'utilizzo delle risorse;
3. meccanismi per negare le risorse a quegli utenti che hanno esaurito il budget loro assegnato.

La misurazione dell'utilizzo delle risorse è una funzione distribuita in tutto il sistema, in quei punti dove le risorse sono allocate. Può essere facilitata enormemente da un hardware opportuno, quale, ad esempio, un contatore nel controllore del disco per registrare il numero dei trasferimenti. Per un'accurata misurazione è ovviamente indispensabile un orologio in tempo reale. Le misurazioni relative ad un singolo processo possono venir memorizzate come parte del descrittore relativo ed i dati statistici sull'impiego complessivo del sistema possono essere raccolti in un file centrale. Questi ultimi possono essere ritenuti indicativi delle prestazioni del sistema, ed usati come base di raffronto e di potenziamento.
Si dovrebbe forse tener conto anche della questione riguardante l'eventuale addebito all'utente degli appesantimenti sul sistema operativo. Da un lato si potrebbe sostenere che il sistema operativo agisce per conto dell'utente e che, pertanto, quest'ultimo dovrebbe essere assoggettato ai relativi addebiti. In tal caso, si otterrebbe pure l'effetto marginale, ma vantaggioso, di stimolare gli utenti a redigere i programmi in modo da contenere al massimo l'aggravio del sistema, per esempio evitando accessi superflui ai file. D'altra parte, gli appesantimenti variano in base al carico di lavoro istantaneo e si potrebbe ritenere ingiusto aumentare l'addebito ad un utente, semplicemente perché la macchina ha molto da fare o perché il suo lavoro interagisce negativamente con qualche altro che si dà il caso sia presente contemporaneamente nel sistema. Secondo un'altra argomentazione a sostegno di quest'ultima ipotesi, non è pensabile che l'utente comune acquisisca una conoscenza approfondita del sistema operativo tale da consentirgli di modificare le proprie attività in modo da ridurre gli appesantimenti sul sistema. Un compromesso possibile consiste nell'addebitare all'utente tutte quelle attività di sistema, come la gestione dell'I/O, svolte specificamente per lui, tralasciando, invece, le attività indipendenti dalla sua volontà, come, ad esempio, la commutazione dei processi. Anche questa soluzione, comunque, lascia margini di discussione.
Per esempio, il movimento dovuto alla paginazione dipende sia dall'utente, il quale può aumentare la locallzzazione dei suoi riferimenti in memoria, sia dall'entità delle richieste di memoria. In ultima analisi, le decisioni prese in merito sono necessariamente arbitrarie e vanno lasciate al singolo progettista.

8.8 RIASSUNTO

In questo capitolo abbiamo visto come si possono implementare i metodi di schedulazione ed allocazione delle risorse nell'ambito di un unico processo, denominato gestore dei processi. Abbiamo descritto come è possibile assegnare Le funzioni di quest'ultimo anche ad altri processi, stabilendo così una lista gerarchica dei processi. Il nucleo del sistema operativo ipotetico è stato ampliato inserendovi le procedure per creare e cancellare processi. i meccanismi per l'allocazione delle risorse e per la misurazione del loro utilizzo vengono distribuiti in tutto il sistema, ai livelli più opportuni rispetto alle risorse interessate. Lo stato attuale del nostro sistema è illustrato in figura 8.7.

Fig. 8.7: Stato attuale del sistema operativo ipotetico

Le strutture dati aggiuntive sono:

1. un record delle risorse disponibili;
2. per ogni processo, una lista delle risorse allocate, cui fa riferimento un puntatore nel descrittore relativo;
3. se si impiegano metodi per il riconoscimento di situazioni di stallo, una rappresentazione del grafico di stato;
4. se si impiegano metodi volti ad evitare situazioni di stallo, un record con le richieste e le allocazioni delle risorse;
5. un'eventuale elaborazione della coda d'attesa al processore con l'inclusione di svariati livelli;
6. un'eventuale modifica della lista dei processi per trasformarla in un albero;
7. un file di contabilizzazione.