Capitolo 5

Scheduling della CPU

OBIETTIVI

- **Introduzione allo scheduling della CPU, base dei sistemi operativi multiprogrammati.**
- **Descrizione di vari algoritmi di scheduling della CPU.**
- **Analisi dei criteri di valutazione nella scelta degli algoritmi di scheduling della CPU per particolari sistemi.**

Lo scheduling della CPU è alla base dei sistemi operativi multiprogrammati: attraverso la commutazione del controllo della CPU tra i vari processi, il sistema operativo può rendere più produttivo il calcolatore. In questo capitolo s'introducono i concetti fondamentali dello scheduling e si descrivono vari algoritmi di scheduling della CPU. Si affronta inoltre il problema della scelta dell'algoritmo da impiegare per un dato sistema.

Nel Capitolo 4 abbiamo arricchito il concetto di processo introducendo i thread. Nei sistemi operativi che li contemplano, sono loro, in effetti, l'oggetto dell'attività di scheduling, e non i processi. Ciononostante, le locuzioni **scheduling dei processi** e **scheduling dei thread** sono spesso considerate equivalenti. In questo capitolo useremo la prima nell'analizzare i principi generali dello scheduling, e la seconda nel trattare idee specificamente inerenti ai thread.

5.1 Concetti fondamentali

In un sistema monoprocessore si può eseguire al massimo un processo alla volta; gli altri processi, se ci sono, devono attendere che la CPU sia libera e possa essere nuovamente sottoposta a scheduling.

L'idea della multiprogrammazione è relativamente semplice. Un processo è in esecuzione finché non deve attendere un evento, generalmente il completamento di qualche richiesta di I/O; durante l'attesa, in un sistema di calcolo semplice, la CPU resterebbe inattiva, e tutto il tempo d'attesa sarebbe sprecato. Con la multiprogrammazione si cerca d'impiegare questi tempi d'attesa in modo produttivo: si tengono contemporaneamente più processi in memoria, e quando un processo deve attendere un evento, il sistema operativo gli sottrae il controllo della CPU per cederlo a un altro processo.

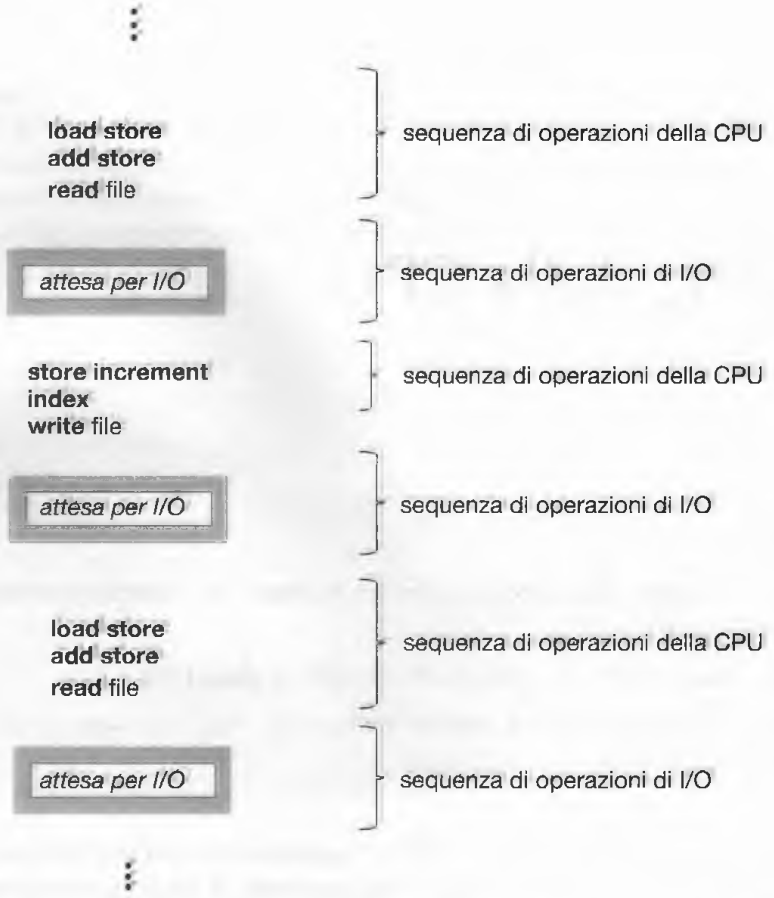


Figura 5.1 Serie alternata di sequenze di operazioni della CPU e di sequenze di operazioni di I/O.

Lo scheduling è una funzione fondamentale dei sistemi operativi; si sottopongono a scheduling quasi tutte le risorse di un calcolatore. Naturalmente la CPU è una delle risorse principali, e il suo scheduling è alla base della progettazione dei sistemi operativi.

5.1.1 Ciclicità delle fasi d'elaborazione e di I/O

Il successo dello scheduling della CPU dipende dall'osservazione della seguente proprietà dei processi: l'esecuzione del processo consiste in un **ciclo** d'elaborazione (svolta dalla CPU) e d'attesa del completamento delle operazioni di I/O. I processi si alternano tra questi due stati. L'esecuzione di un processo comincia con una sequenza (una "raffica") di operazioni d'elaborazione svolte dalla CPU (*CPU burst*), seguita da una sequenza di operazioni di I/O (*I/O burst*), quindi un'altra sequenza di operazioni della CPU, di nuovo una sequenza di operazioni di I/O, e così via. L'ultima sequenza di operazioni della CPU si conclude con una richiesta al sistema di terminare l'esecuzione (Figura 5.1).

Le durate delle sequenze di operazioni della CPU sono state misurate, e sebbene varino molto secondo il processo e secondo il calcolatore, la loro curva di frequenza è simile a quella illustrata nella Figura 5.2. La curva è generalmente di tipo esponenziale o iperesponenziale, con molte brevi sequenze di operazioni della CPU, e poche sequenze di operazioni della CPU molto lunghe. Un programma con prevalenza di I/O (*I/O bound*) produce generalmente molte sequenze di operazioni della CPU di breve durata. Un programma con prevalenza d'elaborazione (*CPU bound*), invece, produce poche sequenze di operazioni della CPU molto lunghe. Queste caratteristiche possono essere utili nella scelta di un appropriato algoritmo di scheduling della CPU.

Figura 5.2 Diagramma delle durate delle sequenze di operazioni della CPU.

5.1.2 Scheduler della CPU

Ogniqualvolta la CPU passa nello stato d'inattività, il sistema operativo sceglie per l'esecuzione uno dei processi presenti nella coda dei processi pronti. In particolare, è lo **scheduler a breve termine**, o scheduler della CPU che, tra i processi in memoria pronti per l'esecuzione, sceglie quello cui assegnare la CPU.

La coda dei processi pronti non è necessariamente una coda in ordine d'arrivo (*first-in, first-out*, FIFO). Come si nota analizzando i diversi algoritmi di scheduling, una coda dei processi pronti si può realizzare come una coda FIFO, una coda con priorità, un albero o semplicemente una lista concatenata non ordinata. Tuttavia, concettualmente tutti i processi della coda dei processi pronti sono posti nella lista d'attesa per accedere alla CPU. Generalmente gli elementi delle code sono i *process control block* (PCB) dei processi.

5.1.3 Scheduling con diritto di prelazione

Le decisioni riguardanti lo scheduling della CPU si possono prendere nelle seguenti circostanze:

1. un processo passa dallo stato di esecuzione allo stato di attesa (per esempio, richiesta di I/O o richiesta di attesa (`wait`) per la terminazione di uno dei processi figli);
2. un processo passa dallo stato di esecuzione allo stato pronto (per esempio, quando si verifica un segnale d'interruzione);
3. un processo passa dallo stato di attesa allo stato pronto (per esempio, al completamento di un'operazione di I/O);
4. un processo termina.

I casi 1 e 4 in quanto tali non comportano alcuna scelta di scheduling; a essi segue la scelta di un nuovo processo da eseguire, sempre che ce ne sia almeno uno nella coda dei processi pronti per l'esecuzione. Una scelta si deve invece fare nei casi 2 e 3.

Quando lo scheduling interviene solo nelle condizioni 1 e 4, si dice che lo schema di scheduling è **senza diritto di prelazione** (*nonpreemptive*) o **cooperativo** (*cooperative*); altrimenti, lo schema di scheduling è **con diritto di prelazione** (*preemptive*). Nel caso dello scheduling senza diritto di prelazione, quando si assegna la CPU a un processo, questo rimane in possesso della CPU fino al momento del suo rilascio, dovuto al termine dell'esecuzione o al

passaggio nello stato di attesa. Questo metodo di scheduling è stato adottato da Microsoft Windows 3.x; lo scheduling con diritto di prelazione è stato introdotto a partire da Windows 95 e in tutte le versioni successive. È adottato anche dal sistema Mac OS X; le versioni precedenti del sistema operativo Macintosh si basavano sullo scheduling cooperativo. Poiché, diversamente dallo scheduling con diritto di prelazione, non richiede dispositivi particolari (per esempio i timer), lo scheduling cooperativo è l'unico metodo utilizzabile su alcune piattaforme.

Sfortunatamente lo scheduling con diritto di prelazione presenta un inconveniente. Si consideri il caso in cui due processi condividono dati; mentre uno di questi aggiorna i dati si ha la sua prelazione in favore dell'esecuzione dell'altro. Il secondo processo può, a questo punto, tentare di leggere i dati che sono stati lasciati in uno stato incoerente dal primo processo. Sono quindi necessari nuovi meccanismi per coordinare l'accesso ai dati condivisi; questo argomento è trattato nel Capitolo 6.

La capacità di prelazione si ripercuote anche sulla progettazione del kernel del sistema operativo. Durante l'elaborazione di una chiamata di sistema, il kernel può essere impegnato in attività in favore di un processo; tali attività possono comportare la necessità di modifiche a importanti dati del kernel, come le code di I/O. Se si ha la prelazione del processo durante tali modifiche e il kernel (o un driver di dispositivo) deve leggere o modificare gli stessi dati, si può avere il caos. Alcuni sistemi operativi, tra cui la maggior parte delle versioni dello UNIX, affrontano questo problema attendendo il completamento della chiamata di sistema o che si abbia il blocco dell'I/O prima di eseguire un cambio di contesto. Quindi, il kernel non può esercitare la prelazione su un processo mentre le strutture dati del kernel si trovano in uno stato incoerente. Sfortunatamente, questo modello d'esecuzione del kernel non è adeguato alle computazioni in tempo reale e alla multielaborazione. Questi problemi e le relative soluzioni sono descritti nei Paragrafi 5.5 e 19.5.

Poiché le interruzioni si possono, per definizione, verificare in ogni istante e il kernel non può sempre ignorale, le sezioni di codice eseguite per effetto delle interruzioni devono essere protette da un uso simultaneo. Il sistema operativo deve ignorare raramente le interruzioni, altrimenti si potrebbero perdere dati in ingresso, o si potrebbero sovrascrivere dati in uscita. Per evitare che più processi accedano in modo concorrente a tali sezioni di codice, queste disattivano le interruzioni al loro inizio e le riattivano alla fine. Le sezioni di codice che disabilitano le interruzioni si verificano tuttavia raramente e, in genere, non contengono molte istruzioni.

5.1.4 Dispatcher

Un altro elemento coinvolto nella funzione di scheduling della CPU è il **dispatcher**; si tratta del modulo che passa effettivamente il controllo della CPU ai processi scelti dallo scheduler a breve termine. Questa funzione riguarda quel che segue:

- il cambio di contesto;
- il passaggio alla modalità utente;
- il salto alla giusta posizione del programma utente per riavviarne l'esecuzione.

Poiché si attiva a ogni cambio di contesto, il dispatcher dovrebbe essere quanto più rapido è possibile. Il tempo richiesto dal dispatcher per fermare un processo e avviare l'esecuzione di un altro è nota come **latenza di dispatch**.

5.2 Criteri di scheduling

Diversi algoritmi di scheduling della CPU hanno proprietà differenti e possono favorire una particolare classe di processi. Prima di scegliere l'algoritmo da usare in una specifica situazione occorre considerare le caratteristiche dei diversi algoritmi.

Per il confronto tra gli algoritmi di scheduling della CPU sono stati suggeriti molti criteri. Le caratteristiche usate per tale confronto possono incidere in modo rilevante sulla scelta dell'algoritmo migliore. Di seguito si riportano alcuni criteri.

- **Utilizzo della CPU.** La CPU deve essere più attiva possibile. Teoricamente, l'utilizzo della CPU può variare dallo 0 al 100 per cento. In un sistema reale può variare dal 40 per cento, per un sistema con poco carico, al 90 per cento, per un sistema con utilizzo intenso.
- **Produttività.** La CPU è attiva quando si svolge del lavoro. Una misura del lavoro svolto è data dal numero dei processi completati nell'unità di tempo: tale misura è detta **produttività** (*throughput*). Per processi di lunga durata questo rapporto può essere di un processo all'ora, mentre per brevi transazioni è possibile avere una produttività di 10 processi al secondo.
- **Tempo di completamento.** Considerando un processo particolare, un criterio importante può essere relativo al tempo necessario per eseguire il processo stesso. L'intervallo che intercorre tra la sottomissione del processo e il completamento dell'esecuzione è chiamato **tempo di completamento** (*turnaround time*), ed è la somma dei tempi passati nell'attesa dell'ingresso in memoria, nella coda dei processi pronti, durante l'esecuzione nella CPU e nelle operazioni di I/O.
- **Tempo d'attesa.** L'algoritmo di scheduling della CPU non influisce sul tempo impiegato per l'esecuzione di un processo o di un'operazione di I/O; influisce solo sul tempo d'attesa nella coda dei processi pronti. Il **tempo d'attesa** è la somma degli intervalli d'attesa passati nella coda dei processi pronti.
- **Tempo di risposta.** In un sistema interattivo il tempo di completamento può non essere il miglior criterio di valutazione: spesso accade che un processo emetta dati abbastanza presto, e continui a calcolare i nuovi risultati mentre quelli precedenti sono in fase d'emissione. Quindi, un'altra misura di confronto è data dal tempo che intercorre tra la sottomissione di una richiesta e la prima risposta prodotta. Questa misura è chiamata **tempo di risposta**, ed è data dal tempo necessario per iniziare la risposta, ma non dal suo tempo d'emissione. Generalmente il tempo di completamento è limitato dalla velocità del dispositivo d'emissione dei dati.

È auspicabile aumentare al massimo utilizzo e produttività della CPU, mentre il tempo di completamento, il tempo d'attesa e il tempo di risposta si devono ridurre al minimo. Nella maggior parte dei casi si ottimizzano i valori medi; in alcune circostanze è più opportuno ottimizzare i valori minimi o massimi, anziché i valori medi; per esempio, per garantire che tutti gli utenti ottengano un buon servizio, può essere utile ridurre il massimo tempo di risposta.

Per i sistemi interattivi, come i sistemi a tempo ripartito, alcuni analisti suggeriscono che sia più importante ridurre al minimo la **varianza** del tempo di risposta anziché il tempo medio di risposta. Un sistema il cui tempo di risposta sia ragionevole e **prevedibile** può essere considerato migliore di un sistema mediamente più rapido, ma molto variabile. Tuttavia, è stato fatto poco sugli algoritmi di scheduling della CPU al fine di ridurre al minimo la varianza.

Poiché si analizzano diversi algoritmi di scheduling della CPU, è opportuno che se ne illustri anche il funzionamento. Una spiegazione approfondita richiederebbe il ricorso a molti processi, ognuno dei quali costituito da parecchie centinaia di sequenze di operazioni della CPU e di sequenze di operazioni di I/O. Per motivi di semplicità, negli esempi si considera una sola sequenza di operazioni della CPU (la cui durata è espressa in millisecondi) per ogni processo. La misura di confronto adottata è il tempo d'attesa medio. Meccanismi di valutazione più raffinati sono trattati nel Paragrafo 5.7.

5.3 Algoritmi di scheduling

Lo scheduling della CPU riguarda la scelta dei processi presenti nella coda dei processi pronti cui assegnare la CPU. In questo paragrafo si descrivono alcuni fra i tanti algoritmi di scheduling della CPU.

5.3.1 Scheduling in ordine d'arrivo

Il più semplice algoritmo di scheduling della CPU è l'algoritmo di **scheduling in ordine d'arrivo** (*scheduling first-come, first-served*, FCFS). Con questo schema la CPU si assegna al processo che la richiede per primo. La realizzazione del criterio FCFS si fonda su una coda FIFO. Quando un processo entra nella coda dei processi pronti, si collega il suo PCB all'ultimo elemento della coda. Quando è libera, si assegna la CPU al processo che si trova alla testa della coda dei processi pronti, rimuovendolo da essa. Il codice per lo scheduling FCFS è semplice sia da scrivere sia da capire.

Il tempo medio d'attesa per l'algoritmo FCFS è spesso abbastanza lungo. Si consideri il seguente insieme di processi, che si presenta al momento 0, con la durata della sequenza di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza
P_1	24
P_2	3
P_3	3

Se i processi arrivano nell'ordine P_1, P_2, P_3 e sono serviti in ordine FCFS, si ottiene il risultato illustrato dal seguente nel seguente **diagramma di Gantt**, un istogramma che illustra una data pianificazione includendo i tempi d'inizio e fine di ogni processo partecipante.

Il tempo d'attesa è 0 millisecondi per il processo P_1, 24 millisecondi per il processo P_2 e 27 millisecondi per il processo P_3. Quindi, il tempo d'attesa medio è (0 + 24 + 27)/3 = 17 mil-

lisecondi. Se i processi arrivassero nell'ordine P_2, P_3, P_1, i risultati sarebbero quelli illustrati nel seguente diagramma di Gantt:

Il tempo di attesa medio è ora di (6 + 0 + 3)/3 = 3 millisecondi. Si tratta di una notevole riduzione. Quindi, il tempo medio d'attesa in condizioni di FCFS non è in genere minimo, e può variare sostanzialmente al variare della durata delle sequenze di operazioni della CPU dei processi.

Si considerino inoltre le prestazioni dello scheduling FCFS in una situazione dinamica. Si supponga di avere un processo con prevalenza d'elaborazione e molti processi con prevalenza di I/O. Via via che i processi fluiscono nel sistema si può riscontrare come il processo con prevalenza d'elaborazione occupi la CPU. Durante questo periodo tutti gli altri processi terminano le proprie operazioni di I/O e si spostano nella coda dei processi pronti, nell'attesa della CPU. Mentre i processi si trovano nella coda dei processi pronti, i dispositivi di I/O sono inattivi. Successivamente il processo con prevalenza d'elaborazione termina la propria sequenza di operazioni della CPU e passa a una fase di I/O. Tutti i processi con prevalenza di I/O, caratterizzati da sequenze di operazioni della CPU molto brevi, sono eseguiti rapidamente e tornano alle code di I/O, lasciando inattiva la CPU. Il processo con prevalenza d'elaborazione torna nella coda dei processi pronti e riceve il controllo della CPU; così, finché non termina l'esecuzione del processo con prevalenza d'elaborazione, tutti i processi con prevalenza di I/O si trovano nuovamente ad attendere nella coda dei processi pronti. Si ha un **effetto convoglio**, tutti i processi attendono che un lungo processo liberi la CPU, che causa una riduzione dell'utilizzo della CPU e dei dispositivi rispetto a quella che si avrebbe se si eseguissero per primi i processi più brevi.

L'algoritmo di scheduling FCFS è senza prelazione; una volta che la CPU è assegnata a un processo, questo la trattiene fino al momento del rilascio, che può essere dovuto al termine dell'esecuzione o alla richiesta di un'operazione di I/O. L'algoritmo FCFS risulta particolarmente problematico nei sistemi a tempo ripartito, dove è importante che ogni utente disponga della CPU a intervalli regolari. Permettere a un solo processo di occupare la CPU per un lungo periodo condurrebbe a risultati disastrosi.

5.3.2 Scheduling per brevità

Un criterio diverso di scheduling della CPU si può ottenere con l'algoritmo di **scheduling per brevità** (*shortest-job-first*, SJF). Questo algoritmo associa a ogni processo la lunghezza della successiva sequenza di operazioni della CPU. Quando è disponibile, si assegna la CPU al processo che ha la più breve lunghezza della successiva sequenza di operazioni della CPU. Se due processi hanno le successive sequenze di operazioni della CPU della stessa lunghezza si applica lo scheduling FCFS. Si noti che sarebbe più appropriato il termine *shortest next CPU burst*, infatti lo scheduling si esegue esaminando la lunghezza della successiva sequenza di operazioni della CPU del processo e non la sua lunghezza totale. Tuttavia, poiché è usato nella maggior parte dei libri di testo, anche qui si fa uso del termine SJF.

Come esempio si consideri il seguente insieme di processi, con la durata della sequenza di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza
P_1	6
P_2	8
P_3	7
P_4	3

Con lo scheduling SJF questi processi si ordinerebbero secondo il seguente diagramma di Gantt.

0 3 9 16 24

Il tempo d'attesa è di 3 millisecondi per il processo P_1, di 16 millisecondi per il processo P_2, di 9 millisecondi per il processo P_3 e di 0 millisecondi per il processo P_4. Quindi, il tempo d'attesa medio è di (3 + 16 + 9 + 0)/4 = 7 millisecondi. Usando lo scheduling FCFS, il tempo d'attesa medio sarebbe di 10,25 millisecondi.

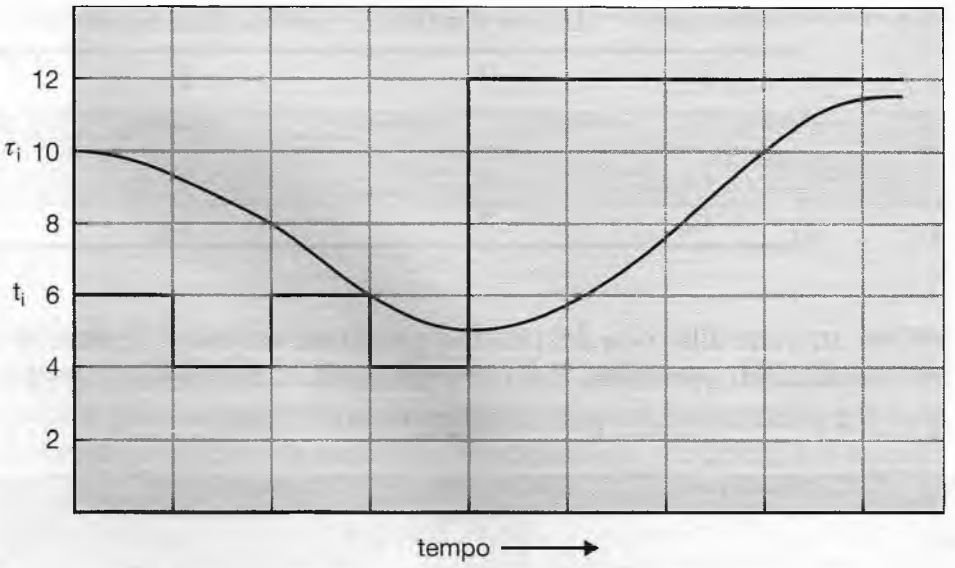
Si può dimostrare che l'algoritmo di scheduling SJF è *ottimale*, nel senso che rende minimo il tempo d'attesa medio per un dato insieme di processi. Spostando un processo breve prima di un processo lungo, il tempo d'attesa per il processo breve diminuisce più di quanto aumenti il tempo d'attesa per il processo lungo. Di conseguenza, il tempo d'attesa *medio* diminuisce.

La difficoltà reale implicita nell'algoritmo SJF consiste nel conoscere la durata della successiva richiesta della CPU. Per lo scheduling a lungo termine (*job scheduling*) di un sistema a lotti si può usare come durata il tempo limite d'elaborazione che gli utenti specificano nel sottoporre il processo. Gli utenti sono quindi motivati a stabilire con precisione tale limite, poiché un valore inferiore può significare una risposta più rapida. Occorre però notare che un valore troppo basso causa un errore di superamento del tempo limite e richiede una nuova esecuzione. Lo scheduling SJF si usa spesso nello scheduling a lungo termine.

Sebbene sia ottimale, l'algoritmo SJF non si può realizzare a livello dello scheduling della CPU a breve termine, poiché non esiste alcun modo per conoscere la lunghezza della successiva sequenza di operazioni della CPU. Un possibile metodo consiste nel tentare di approssimare lo scheduling SJF: se non è possibile *conoscere* la lunghezza della prossima sequenza di operazioni della CPU, si può cercare di *predire* il suo valore; è probabile, infatti, che sia simile ai precedenti. Quindi, calcolando un valore approssimato della lunghezza, si può scegliere il processo con la più breve fra tali lunghezze previste.

La lunghezza della successiva sequenza di operazioni della CPU generalmente si ottiene calcolando la **media esponenziale** delle effettive lunghezze delle precedenti sequenze di operazioni della CPU. La media esponenziale si definisce con la formula seguente. Denotando con t_n la lunghezza dell'n-esima sequenza di operazioni della CPU e con τ_{n+1} il valore previsto per la successiva sequenza di operazioni della CPU, con α tale che $0 \leq \alpha \leq 1$, si definisce

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n$$



CPU burst (t_i)	6	4	6	4	13	13	13	...
predizione (τ_i)	10	8	6	6	5	9	11	12	...

Figura 5.3 Predizione della lunghezza della successiva sequenza di operazioni della CPU (CPU burst).

Il valore di t_n contiene le informazioni più recenti; τ_n registra la storia passata. Il parametro α controlla il peso relativo sulla predizione della storia recente e di quella passata. Se $\alpha = 0$, allora, $\tau_{n+1} = \tau_n$, e la storia recente non ha effetto; si suppone, cioè, che le condizioni attuali siano transitorie; se $\alpha = 1$, allora $\tau_{n+1} = t_n$, e ha significato solo la più recente sequenza di operazioni della CPU: si suppone, cioè, che la storia sia vecchia e irrilevante. Più comune è la condizione in cui $\alpha = 1/2$, valore che indica che la storia recente e la storia passata hanno lo stesso peso. Il τ_0 iniziale si può definire come una costante o come una media complessiva del sistema. Nella Figura 5.3 è illustrata una media esponenziale con $\alpha = 1/2$ e $\tau_0 = 10$.

Per comprendere il comportamento della media esponenziale, si può sviluppare la formula per τ_{n+1} sostituendo in τ_n, in modo da ottenere

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots + (1 - \alpha)^j \alpha t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0$$

Poiché sia α sia $(1 - \alpha)$ sono minori o uguali a 1, ogni termine ha peso inferiore a quello del suo predecessore.

L'algoritmo SJF può essere sia *con prelazione* sia *senza prelazione*. La scelta si presenta quando alla coda dei processi pronti arriva un nuovo processo mentre un altro processo è ancora in esecuzione. Il nuovo processo può avere una successiva sequenza di operazioni della CPU più breve di quella che resta al processo correntemente in esecuzione. Un algoritmo SJF con prelazione sostituisce il processo attualmente in esecuzione, mentre un algoritmo SJF senza prelazione permette al processo correntemente in esecuzione di portare a termine la propria sequenza di operazioni della CPU. (Lo scheduling SJF con prelazione è talvolta chiamato scheduling *shortest-remaining-time-first*.)

Come esempio, si considerino i quattro processi seguenti, dove la durata delle sequenze di operazioni della CPU è data in millisecondi:

Processo	Istante d'arrivo	Durata della sequenza
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Se i processi arrivano alla coda dei processi pronti nei momenti indicati e richiedono le durate delle sequenze di operazioni della CPU illustrate, dallo scheduling SJF con prelazione risulta quel che è indicato dal seguente diagramma di Gantt.

All'istante 0 si avvia il processo P_1, poiché è l'unico che si trova nella coda. All'istante 1 arriva il processo P_2. Il tempo necessario per completare il processo P_1 (7 millisecondi) è maggiore del tempo richiesto dal processo P_2 (4 millisecondi), perciò si ha la prelazione sul processo P_1 sostituendolo col processo P_2. Il tempo d'attesa medio per questo esempio è $((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6{,}5$ millisecondi. Con uno scheduling SJF senza prelazione si otterrebbe un tempo d'attesa medio di 7,75 millisecondi.

5.3.3 Scheduling per priorità

L'algoritmo SJF è un caso particolare del più generale **algoritmo di scheduling per priorità**: si associa una priorità a ogni processo e si assegna la CPU al processo con priorità più alta; i processi con priorità uguali si ordinano secondo uno schema FCFS. Un algoritmo SJF è semplicemente un algoritmo con priorità in cui la priorità (p) è l'inverso della lunghezza (prevista) della successiva sequenza di operazioni della CPU. A una maggiore lunghezza corrisponde una minore priorità, e viceversa.

Occorre notare che la discussione si svolge nei termini di priorità *alta* e priorità *bassa*. Generalmente le priorità sono indicate da un intervallo fisso di numeri, come da 0 a 7, oppure da 0 a 4.095. Tuttavia, non si è ancora stabilito se attribuire allo 0 la priorità più alta o quella più bassa; alcuni sistemi usano numeri bassi per rappresentare priorità basse, altri usano numeri bassi per priorità alte. In questo testo i numeri bassi indicano priorità alte.

Come esempio, si consideri il seguente insieme di processi, che si suppone siano arrivati al tempo 0, nell'ordine P_1, P_2 ... P_5, e dove la durata delle sequenze di operazioni della CPU è espressa in millisecondi:

Processo	Durata della sequenza	Priorità
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Usando lo scheduling per priorità, questi processi sarebbero ordinati secondo il seguente diagramma di Gantt.

Il tempo d'attesa medio è di 8,2 millisecondi.

Le priorità si possono definire sia internamente sia esternamente. Quelle definite internamente usano una o più quantità misurabili per calcolare la priorità del processo; per esempio, i limiti di tempo, i requisiti di memoria, il numero dei file aperti e il rapporto tra la lunghezza media delle sequenze di operazioni di I/O e la lunghezza media delle sequenze di operazioni della CPU. Le priorità esterne si definiscono secondo criteri esterni al sistema operativo, come l'importanza del processo, il tipo e la quantità dei fondi pagati per l'uso del calcolatore, il dipartimento che promuove il lavoro e altri fattori, spesso di ordine politico.

Lo scheduling per priorità può essere sia con prelazione sia senza prelazione. Quando un processo arriva alla coda dei processi pronti, si confronta la sua priorità con quella del processo attualmente in esecuzione. Un algoritmo di scheduling per priorità e con diritto di prelazione sottrae la CPU al processo attualmente in esecuzione se la priorità dell'ultimo processo arrivato è superiore. Un algoritmo di scheduling senza diritto di prelazione si limita a porre l'ultimo processo arrivato alla testa della coda dei processi pronti.

Un problema importante relativo agli algoritmi di scheduling per priorità è l'**attesa indefinita** (*starvation*, letteralmente, "inedia"). Un processo pronto per l'esecuzione, ma che non dispone della CPU, si può considerare bloccato nell'attesa della CPU. Un algoritmo di scheduling per priorità può lasciare processi con bassa priorità nell'attesa indefinita della CPU. Un flusso costante di processi con priorità maggiore può impedire a un processo con bassa priorità di accedere alla CPU. Generalmente accade che o il processo è eseguito, alle ore 2 del mattino della domenica, quando il sistema ha finalmente ridotto il proprio carico, oppure il calcolatore si sovraccarica al punto da perdere tutti i processi con bassa priorità non terminati. Corre voce che, quando fu fermato l'IBM 7094 al MIT, nel 1973, si scoprì che un processo con bassa priorità sottoposto nel 1967 non era ancora stato eseguito.

Una soluzione al problema dell'attesa indefinita dei processi con bassa priorità è costituita dall'**invecchiamento** (*aging*); si tratta di una tecnica di aumento graduale delle priorità dei processi che attendono nel sistema da parecchio tempo. Per esempio, se le priorità variano da 127 (bassa) a 0 (alta), si potrebbe decrementare di 1 ogni 15 minuti la priorità di un processo in attesa. Anche un processo con priorità iniziale 127 può ottenere la priorità massima nel sistema e quindi essere eseguito: un processo con priorità 127 non impiega più di 32 ore per raggiungere la priorità 0.

5.3.4 Scheduling circolare

L'**algoritmo di scheduling circolare** (*round-robin*, RR) è stato progettato appositamente per i sistemi a tempo ripartito; simile allo scheduling FCFS, ha tuttavia in più la capacità di prelazione per la commutazione dei processi. Ciascun processo riceve una piccola quantità fissata del tempo della CPU, chiamata **quanto di tempo** o **porzione di tempo** (*time slice*), che varia generalmente da 10 a 100 millisecondi; la coda dei processi pronti è trattata come una coda circolare. Lo scheduler della CPU scorre la coda dei processi pronti, assegnando la CPU a ciascun processo per un intervallo di tempo della durata massima di un quanto di tempo.

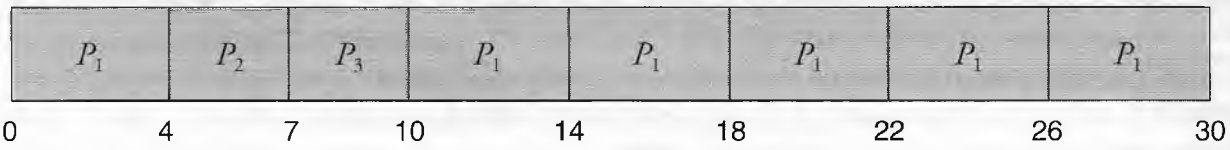
Per realizzare lo scheduling RR si gestisce la coda dei processi pronti come una coda FIFO. I nuovi processi si aggiungono alla fine della coda dei processi pronti. Lo scheduler della CPU individua il primo processo dalla coda dei processi pronti, imposta un timer in modo che invii un segnale d'interruzione alla scadenza di un intervallo pari a un quanto di tempo, e attiva il dispatcher per l'effettiva esecuzione del processo.

A questo punto si può verificare una delle seguenti situazioni: il processo ha una sequenza di operazioni della CPU di durata minore di un quanto di tempo, quindi il processo stesso rilascia volontariamente la CPU e lo scheduler passa al processo successivo della coda dei processi pronti; oppure la durata della sequenza di operazioni della CPU del processo attualmente in esecuzione è più lunga di un quanto di tempo; in questo caso si raggiunge la scadenza del quanto di tempo e il timer invia un segnale d'interruzione al sistema operativo, che esegue un cambio di contesto, aggiunge il processo alla fine della coda dei processi pronti e, tramite lo scheduler della CPU, seleziona il processo successivo nella coda dei processi pronti. Il tempo d'attesa medio per il criterio di scheduling RR è spesso abbastanza lungo.

Si consideri il seguente insieme di processi che si presenta al tempo 0, con la durata delle sequenze di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza
P_1	24
P_2	3
P_3	3

Se si usa un quanto di tempo di 4 millisecondi, il processo P_1 ottiene i primi 4 millisecondi ma, poiché richiede altri 20 millisecondi, è soggetto a prelazione dopo il primo quanto di tempo e la CPU passa al processo successivo della coda, il processo P_2. Poiché il processo P_2 non necessita di 4 millisecondi, termina prima che il suo quanto di tempo si esaurisca, così si assegna immediatamente la CPU al processo successivo, il processo P_3. Una volta che tutti i processi hanno ricevuto un quanto di tempo, si assegna nuovamente la CPU al processo P_1 per un ulteriore quanto di tempo. Dallo scheduling RR risulta quanto segue.



Calcoliamo ora il tempo di attesa medio per questo scheduling. P_1 resta in attesa per 6 millisecondi (10-4), P_2 per 4 millisecondi e P_3 per 7 millisecondi.Il tempo d'attesa medio è di 17/3 = 5,66 millisecondi.

Nell'algoritmo di scheduling RR la CPU si assegna a un processo per non più di un quanto di tempo per volta. Se la durata della sequenza di operazioni della CPU di un processo eccede il quanto di tempo, il processo viene sottoposto a *prelazione* e riportato nella coda dei processi pronti. L'algoritmo di scheduling RR è pertanto con prelazione.

Se nella coda dei processi pronti esistono n processi e il quanto di tempo è pari a q, ciascun processo ottiene $1/n$-esimo del tempo di elaborazione della CPU in frazioni di, al massimo, q unità di tempo. Ogni processo non deve attendere per più di $(n - 1) \times q$ unità di tempo. Per esempio, dati cinque processi e un quanto di tempo di 20 millisecondi, ogni processo usa 20 millisecondi ogni 100 millisecondi.

Le prestazioni dell'algoritmo RR dipendono molto dalla dimensione del quanto di tempo. Nel caso limite in cui il quanto di tempo sia molto lungo (indefinito), il criterio di scheduling RR si riduce al criterio di scheduling FCFS. Se il quanto di tempo è molto breve (per esempio, un microsecondo), il criterio RR si chiama **condivisione della CPU** (*processor sharing*) e teoricamente gli utenti hanno l'impressione che ciascuno degli n processi disponga di una propria CPU in esecuzione a $1/n$ della velocità della CPU reale. Questo metodo fu usato nell'architettura del sistema della Control Data Corporation (CDC) per simulare 10 unità d'elaborazione con 10 gruppi di registri e una sola CPU, che eseguiva un'istruzione per un gruppo di registri, quindi procedeva col successivo. Questo ciclo continuava, con il risultato che si avevano 10 unità d'elaborazione lente al posto di una CPU veloce. (In effetti, poiché la CPU era molto più rapida della memoria e ogni istruzione faceva riferimento alla memoria, le unità d'elaborazione simulate non erano molto più lente di quanto sarebbero state dieci vere unità d'elaborazione.)

Riguardo alle prestazioni dello scheduling RR, occorre tuttavia considerare l'effetto dei cambi di contesto. Dato un solo processo della durata di 10 unità di tempo, se il quanto di tempo è di 12 unità, il processo impiega meno di un quanto di tempo; se però il quanto di tempo è di 6 unità, il processo richiede 2 quanti di tempo e un cambio di contesto; e se il quanto di tempo è di un'unità di tempo, occorrono nove cambi di contesto, con proporzionale rallentamento dell'esecuzione del processo. Tale situazione è visibile nello schema della Figura 5.4.

Da ciò che si è affermato segue che il quanto di tempo deve essere ampio rispetto alla durata del cambio di contesto; se, per esempio, questa è pari al 10 per cento del quanto di tempo, allora s'impiega nel cambio di contesto circa il 10 per cento del tempo d'elaborazione della CPU. In pratica, nella maggior parte dei sistemi moderni un quanto di tempo va dai 10 ai 100 millisecondi. Il tempo richiesto per un cambio di contesto non eccede solitamente i 10 microsecondi, risultando quindi una modesta frazione del quanto di tempo.

Anche il tempo di completamento (*turnaround time*) dipende dalla dimensione del quanto di tempo: com'è evidenziato nella Figura 5.5, il tempo di completamento medio di un insieme di processi non migliora necessariamente con l'aumento della dimensione del quanto di tempo. In generale, il tempo di completamento medio può migliorare se la maggior parte dei processi termina la successiva sequenza di operazioni della CPU in un solo

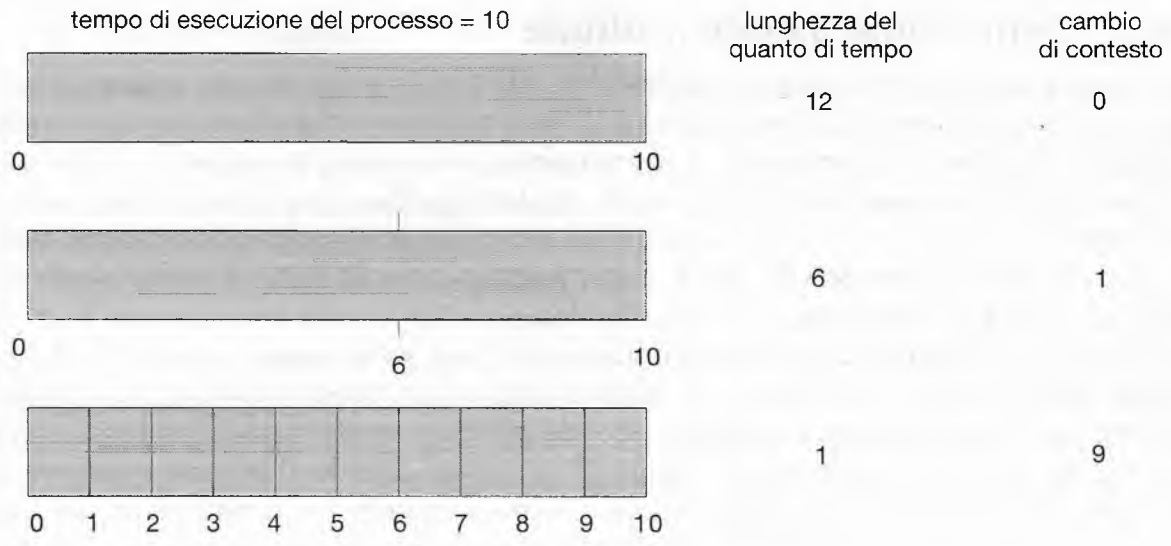
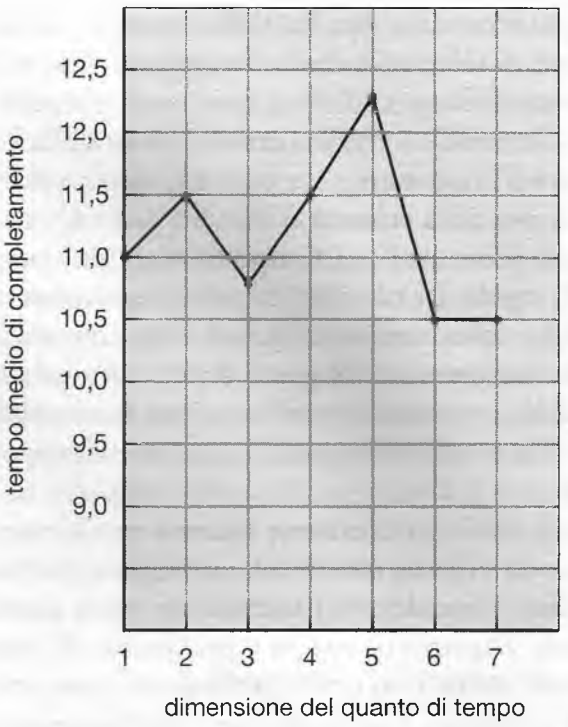


Figura 5.4 Aumento del numero dei cambi di contesto al diminuire del quanto di tempo.



processo	tempo
P_1	6
P_2	3
P_3	1
P_4	7

Figura 5.5 Variazione del tempo di completamento in funzione del quanto di tempo.

quanto di tempo. Per esempio, dati tre processi della durata di 10 unità di tempo ciascuno e un quanto di una unità di tempo, il tempo di completamento medio è di 29 unità. Se però il quanto di tempo è di 10 unità, il tempo di completamento medio scende a 20 unità. Aggiungendo il tempo del cambio di contesto, con un piccolo quanto di tempo, il tempo di completamento medio aumenta poiché sono richiesti più cambi di contesto.

È bene comunque che il quanto di tempo sia maggiore del tempo necessario al cambio di contesto: d'altro canto, è opportuno che lo scarto non sia eccessivo: anche se il quanto di tempo è molto ampio, il criterio di scheduling RR tende al criterio FCFS. Empiricamente si può stabilire che l'80 per cento delle sequenze di operazioni della CPU debba essere più breve del quanto di tempo.

5.3.5 Scheduling a code multiple

È stata creata una classe di algoritmi di scheduling adatta a situazioni in cui i processi si possono classificare facilmente in gruppi diversi. Una distinzione diffusa è per esempio quella che si fa tra i processi che si eseguono in **primo piano** (*foreground*), o **interattivi**, e i processi che si eseguono in **sottofondo** (*background*), o a **lotti** (*batch*). Questi due tipi di processi hanno tempi di risposta diversi e possono quindi avere diverse necessità di scheduling. Inoltre, i processi che si eseguono in primo piano possono avere la priorità, definita esternamente, sui processi che si eseguono in sottofondo.

L'**algoritmo di scheduling a code multiple** (*multilevel queue scheduling algorithm*) suddivide la coda dei processi pronti in code distinte (Figura 5.6). I processi si assegnano in modo permanente a una coda, generalmente secondo qualche caratteristica del processo, come la quantità di memoria richiesta, la priorità o il tipo. Ogni coda ha il proprio algoritmo di scheduling. Per esempio, per i processi in primo piano e i processi in sottofondo si possono usare code distinte. La coda dei processi in primo piano si può gestire con un algoritmo RR, mentre quella dei processi in sottofondo si può gestire con un algoritmo FCFS.

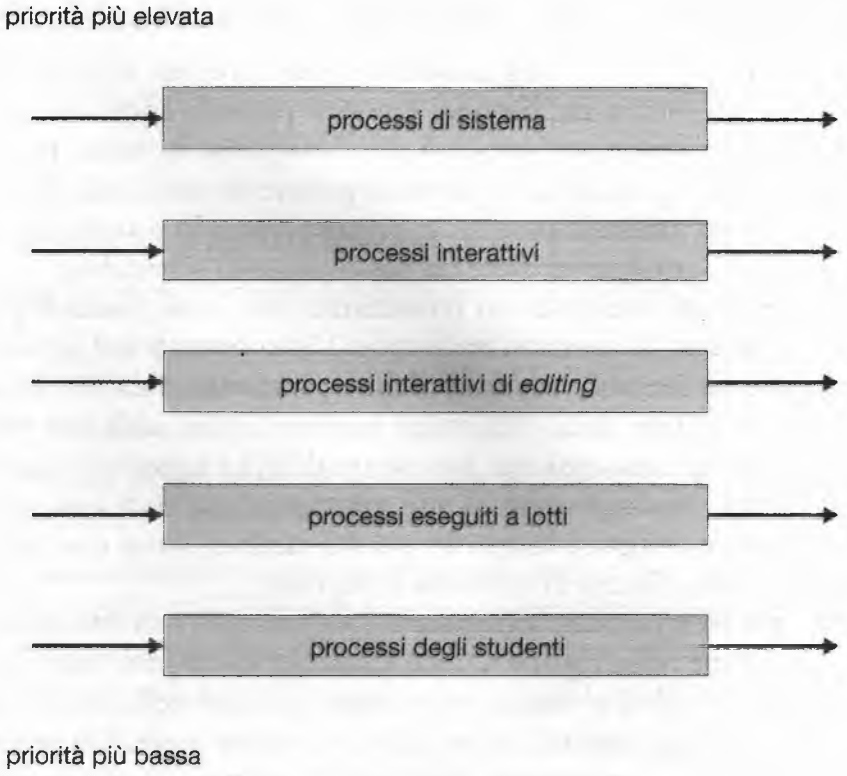


Figura 5.6 Scheduling a code multiple.

In questa situazione è inoltre necessario avere uno scheduling tra le code; si tratta comunemente di uno scheduling per priorità fissa e con prelazione. Per esempio, la coda dei processi in primo piano può avere la priorità assoluta sulla coda dei processi in sottofondo.

Si consideri il seguente algoritmo di scheduling a code multiple, in ordine di priorità:

1. processi di sistema;
2. processi interattivi;
3. processi interattivi di *editing*;
4. processi eseguiti in sottofondo;
5. processi degli studenti.

Ogni coda ha la priorità assoluta sulle code di priorità più bassa; nessun processo della coda dei processi in sottofondo può iniziare l'esecuzione finché le code per i processi di sistema, interattivi e interattivi di *editing* non siano tutte vuote. Se un processo interattivo di *editing* entrasse nella coda dei processi pronti durante l'esecuzione di un processo in sottofondo, si avrebbe la prelazione su quest'ultimo.

Esiste anche la possibilità di impostare i quanti di tempo per le code. Per ogni coda si stabilisce una parte del tempo d'elaborazione della CPU, suddivisibile a sua volta tra i processi che la costituiscono. Nell'esempio precedente, si può assegnare l'80 per cento del tempo d'elaborazione della CPU alla coda dei processi in primo piano, per lo scheduling RR tra i suoi processi; mentre per la coda dei processi in sottofondo si riserva il 20 per cento del tempo d'elaborazione della CPU, da assegnare ai suoi processi sulla base del criterio FCFS.

5.3.6 Scheduling a code multiple con retroazione

Di solito in un algoritmo di scheduling a code multiple i processi si assegnano in modo permanente a una coda all'entrata nel sistema, e non si possono spostare tra le code. Se, per esempio, esistono code distinte per i processi che si eseguono in primo piano e quelli che si eseguono in sottofondo, i processi non possono passare da una coda all'altra, poiché non possono cambiare la loro natura di processi in primo piano o in sottofondo. Quest'impostazione è rigida, ma ha il vantaggio di avere un basso carico di scheduling.

Lo **scheduling a code multiple con retroazione** (*multilevel feedback queue scheduling*), invece, permette ai processi di spostarsi fra le code. L'idea consiste nel separare i processi che hanno caratteristiche diverse nelle sequenze delle operazioni della CPU. Se un processo usa troppo tempo di elaborazione della CPU, viene spostato in una coda con priorità più bassa. Questo schema mantiene i processi con prevalenza di I/O e i processi interattivi nelle code con priorità più elevata. Analogamente, si può spostare in una coda con priorità più elevata un processo che attende troppo a lungo. In questo modo si attua una forma d'invecchiamento che impedisce il verificarsi di un'attesa indefinita.

Si consideri, per esempio, uno scheduler a code multiple con retroazione con tre code, numerate da 0 a 2, come nella Figura 5.7. Lo scheduler fa eseguire tutti i processi presenti nella coda 0; quando la coda 0 è vuota, si eseguono i processi nella coda 1; analogamente, i processi nella coda 2 si eseguono solo se le code 0 e 1 sono vuote. Un processo in ingresso nella coda 1 ha la prelazione sui processi della coda 2; un processo in ingresso nella coda 0, a sua volta, ha la prelazione sui processi della coda 1.

All'ingresso nella coda dei processi pronti, i processi vengono assegnati alla coda 0 e ottengono un quanto di tempo di 8 millisecondi; i processi che non terminano entro tale quanto di tempo, vengono spostati alla fine della coda 1. Se la coda 0 è vuota, si assegna un quanto di tempo di 16 millisecondi al processo alla testa della coda 1, ma se questo non riesce a completare la propria esecuzione, viene sottoposto a prelazione e messo nella coda 2. Se le code 0 e 1 sono vuote, si eseguono i processi della coda 2 secondo il criterio FCFS.

Questo algoritmo di scheduling dà la massima priorità ai processi con una sequenza di operazioni della CPU della durata di non più di 8 millisecondi. I processi di questo tipo ottengono rapidamente la CPU, terminano la propria sequenza di operazioni della CPU e passano alla successiva sequenza di operazioni di I/O; anche i processi che necessitano di più di 8 ma di non più di 24 millisecondi (coda 1) vengono serviti rapidamente. I processi più lun-

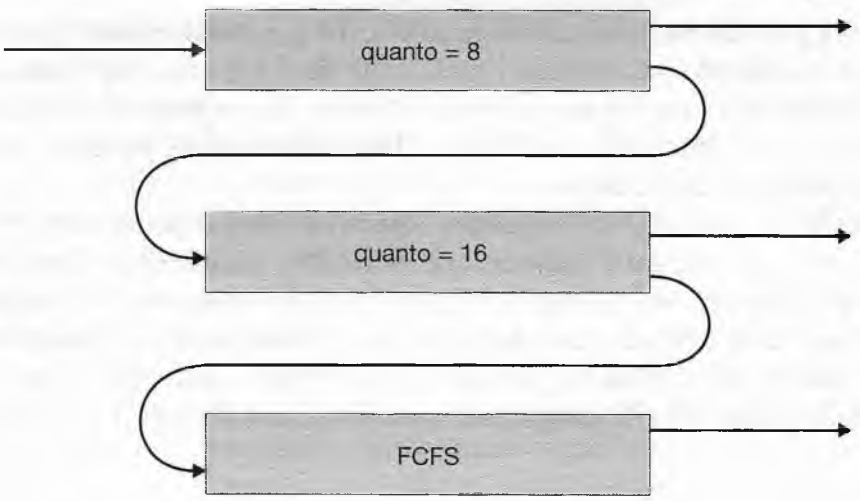


Figura 5.7 Code multiple con retroazione.

ghi finiscono nella coda 2 e sono serviti secondo il criterio FCFS all'interno dei cicli di CPU lasciati liberi dai processi delle code 0 e 1.

Generalmente uno scheduler a code multiple con retroazione è caratterizzato dai seguenti parametri:

- numero di code;
- algoritmo di scheduling per ciascuna coda;
- metodo usato per determinare quando spostare un processo in una coda con priorità maggiore;
- metodo usato per determinare quando spostare un processo in una coda con priorità minore;
- metodo usato per determinare in quale coda si deve mettere un processo quando richiede un servizio.

La definizione di uno scheduler a code multiple con retroazione costituisce il più generale criterio di scheduling della CPU, che nella fase di progettazione si può adeguare a un sistema specifico. Sfortunatamente corrisponde anche all'algoritmo più complesso; la definizione dello scheduler migliore richiede infatti particolari metodi per la selezione dei valori dei diversi parametri.

5.4 Scheduling dei thread

Nel Capitolo 4 abbiamo arricchito il modello dei processi con i thread, distinguendo quelli *a livello utente* da quelli *a livello kernel*. Sui sistemi operativi che prevedono la loro presenza, il sistema pianifica l'esecuzione dei thread a livello kernel, non dei processi. I thread a livello utente sono gestiti da una libreria: il kernel non è consapevole della loro esistenza. Di conseguenza, per eseguire i thread a livello utente occorre associare loro dei thread a livello kernel. Tale associazione può essere indiretta, ossia realizzata con un processo leggero (LWP). Trattiamo adesso le questioni dello scheduling che riguardano i thread a livello utente e a livello kernel, offrendo esempi specifici dello scheduling per Pthreads.

5.4.1 Ambito della competizione

La prima distinzione fra thread a livello utente e a livello kernel riguarda il modo in cui è pianificata la loro esecuzione. Nei sistemi che impiegano il modello da molti a uno (Paragrafo 4.2.1) e il modello da molti a molti (Paragrafo 4.2.3), la libreria dei thread pianifica l'esecuzione dei thread a livello utente su un LWP libero; si parla allora di **ambito della competizione ristretto al processo** (*process-contention scope*, PCS), perché la contesa per aggiudicarsi la CPU ha luogo fra thread dello stesso processo. In realtà, affermando che la libreria dei thread *pianifica* l'esecuzione dei thread a livello utente associandoli agli LWP liberi, non si intende che il thread sia in esecuzione su una CPU; ciò avviene solo quando il sistema operativo pianifica l'esecuzione di thread del kernel su un processore fisico. Per determinare quale thread a livello kernel debba essere eseguito da una CPU, il kernel esamina i thread di tutto il sistema; si parla allora di **ambito della competizione allargato al sistema** (*system-contention scope*, SCS). Quindi, nel caso di SCS, tutti i thread del sistema competono per l'uso della CPU. I sistemi caratterizzati dal modello da uno a uno (quali Windows XP, Solaris 9 e Linux) pianificano i thread unicamente sulla base di SCS.

Se l'ambito della competizione è ristretto al processo, lo scheduling è solitamente basato sulle priorità: lo scheduler sceglie per l'esecuzione il thread con priorità più alta. Le priorità dei thread a livello utente sono stabilite dal programmatore, e la libreria dei thread non le modifica; alcune librerie danno facoltà al programmatore di cambiare la priorità di un thread. Si noti che quando l'ambito della competizione è ristretto al processo si è soliti applicare la prelazione al thread in esecuzione, a vantaggio di thread con priorità più alta; tuttavia, se i thread sono dotati della medesima priorità, non vi è garanzia sulla ripartizione del tempo (Paragrafo 5.3.4).

5.4.2 Scheduling di Pthread

La generazione dei thread con POSIX Pthreads è stata introdotta nel Paragrafo 4.3.1, insieme a un programma esemplificativo. Ci accingiamo ora a esaminare la API Pthread del POSIX, che consente di specificare sia PCS che SCS per la generazione dei thread. Per specificare l'ambito della contesa Pthreads usa i valori seguenti:

- PTHREAD_SCOPE_PROCESS pianifica i thread con lo scheduling PCS
- PTHREAD_SCOPE_SYSTEM pianifica i thread tramite lo scheduling SCS

Nei sistemi che si avvalgono del modello da molti a molti (Paragrafo 4.2.3), la politica PTHREAD_SCOPE_PROCESS pianifica i thread a livello utente sugli LWP disponibili. Il numero di LWP viene stabilito dalla libreria dei thread, che in qualche caso si serve delle attivazioni dello scheduler (4.4.6). La seconda politica, PTHREAD_SCOPE_SYSTEM, crea, in corrispondenza di ciascun thread a livello utente, un LWP a esso vincolato, realizzando così una corrispondenza secondo il modello da molti a uno (Paragrafo 4.2.2).

Lo IPC di Pthread offre due funzioni per appurare e impostare l'ambito della contesa:

- `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
- `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`

Il primo parametro per entrambe le funzioni è un puntatore agli attributi del thread. Il secondo parametro della funzione `pthread_attr_setscope()` riceve uno dei valori PTHREAD_SCOPE_SYSTEM o PTHREAD_SCOPE_PROCESS, che stabiliscono l'ambito della contesa. Qualora si verifichi un errore, ambedue le funzioni restituiscono valori non nulli.

Nella Figura 5.8 presentiamo un programma Pthread che determina l'ambito della contesa in vigore e lo imposta a PTHREAD_SCOPE_PROCESS; quindi crea cinque thread distinti, che andranno in esecuzione secondo il modello di scheduling SCS. Si noti che, nel caso di alcuni sistemi, sono possibili solo determinati valori per l'area della disputa. I sistemi Linux e Mac OS X, per esempio, consentono soltanto PTHREAD_SCOPE_SYSTEM.

5.5 Scheduling per sistemi multiprocessore

Fin qui la trattazione ha riguardato i problemi inerenti lo scheduling della CPU in un sistema a processore singolo; se sono disponibili più unità d'elaborazione, anche il problema dello scheduling è proporzionalmente più complesso. Si sono sperimentate diverse possibilità e, come s'è visto nella trattazione dello scheduling di una sola CPU, "la soluzione migliore" non esiste. Nel seguito si analizzano brevemente alcuni argomenti attinenti lo scheduling per sistemi multiprocessore. Si considerano i sistemi in cui le unità d'elaborazione sono, in relazione alle loro funzioni, identiche (**sistemi omogenei**): si può usare qualunque unità d'elaborazione disponibile per eseguire qualsiasi processo presente nella coda. (Ma anche i

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
  int i, scope;
  pthread_t tid[NUM_THREADS];
  pthread_attr_t attr;

  /* ottiene gli attributi di default */
  pthread_attr_init(&attr);

  /* per prima cosa appura l'ambito della contesa */
  if (pthread_attr_getscope(&attr, &scope) != 0)
    fprintf(stderr, "Impossibile appurare l'ambito della contesa\n");
  else {
    if (scope == PTHREAD_SCOPE_PROCESS)
      printf("PTHREAD_SCOPE_PROCESS");
    else if (scope == PTHREAD_SCOPE_SYSTEM)
      printf("PTHREAD_SCOPE_SYSTEM");
    else
      fprintf(stderr, "Valore d'ambito della contesa non
              ammesso.\n");
   }

  /* imposta l'algoritmo di scheduling a PCS o SCS */
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  /* genera i thread */
  for (i = 0; i < NUM_THREADS; i++)
     pthread_create(&tid[i],&attr,runner,NULL);

  /* adesso aspetta la terminazione di tutti i thread */
  for (i = 0; i < NUM_THREADS; i++)
     pthread_join(tid[i], NULL);
}

/* ogni thread inizia l'esecuzione da questa funzione */
void *runner(void *param)
{
  /* fai qualcosa ... */

  pthread_exit(0);
}
```

Figura 5.8 API Pthread per lo scheduling.

multiprocessori omogenei, talvolta, possono incorrere in limitazioni. Si consideri un sistema con un dispositivo di I/O collegato a un processore mediante un bus privato; ogni processo che intenda avvalersi di tale dispositivo dovrà essere pianificato per l'esecuzione su quel processore.)

5.5.1 Soluzioni di scheduling per multiprocessori

Una prima strategia di scheduling della CPU per i sistemi multiprocessore affida tutte le decisioni, l'elaborazione dell' I/O e le altre attività del sistema a un solo processore, il cosiddetto *master server*. Gli altri processori eseguono soltanto il codice dell'utente. Si tratta della **multielaborazione asimmetrica**, che riduce la necessità di condividere dati grazie all'accesso di un solo processore alle strutture dati del sistema.

Quando invece ciascun processore provvede al proprio scheduling, si parla di **multielaborazione simmetrica** (*symmetric multiprocessing*, SMP). In questo caso i processi pronti per l'esecuzione sono situati tutti in una coda comune, oppure vi è un'apposita coda per ogni processore. In entrambi i casi, lo scheduler di ciascun processore esamina la coda appropriata, da cui seleziona un processo da eseguire. Come vedremo nel Capitolo 6, l'accesso concorrente di più processori a una struttura dati comune rende delicata la programmazione dello scheduler, al fine di evitare che due processori scelgano il medesimo processo o che un processo in coda non vada perso. La SMP è messa a disposizione da quasi tutti i sistemi operativi moderni, quali Windows XP, Windows 2000, Solaris, Linux e Mac OS X.

Nei paragrafi successivi discuteremo le questioni concernenti la SMP.

5.5.2 Predilezione per il processore

Si consideri che cosa accade alla memoria cache dopo che un processo sia stato eseguito da uno specifico processore: i dati che il processore ha trattato da ultimo permangono nella cache e, di conseguenza, i successivi accessi alla memoria da parte del processo tendono a utilizzare spesso la memoria cache. Ecco allora che cosa succede se un processo si sposta su un altro processore: i contenuti della memoria cache devono essere invalidati sul processore di partenza, mentre la cache del processore di arrivo deve essere nuovamente riempita. A causa degli alti costi di svuotamento e riempimento della cache, molti sistemi SMP tentano di impedire il passaggio di processi da un processore all'altro, mirando a mantenere un processo sullo stesso processore che lo sta eseguendo. Si parla di **predilezione per il processore** (*processor affinity*), intendendo con ciò che un processo ha una predilezione per il processore su cui è in esecuzione.

La predilezione per il processore può assumere varie forme. Quando un sistema operativo si propone di mantenere un processo su un singolo processore, ma non garantisce che sarà così, si parla di **predilezione debole** (*soft affinity*). In questo caso è possibile che un processo migri da un processore all'altro. Alcuni sistemi, per esempio Linux, dispongono di chiamate di sistema con cui specificare che un processo non debba cambiare processore; in tal modo, si realizza la **predilezione forte** (*hard affinity*). Solaris permette che i processi siano assegnati a un determinato insieme di processori, definendo quali processi possono essere eseguiti da una data CPU. Solaris permette anche la predilezione debole.

L'architettura della memoria principale di un sistema può influenzare le questioni relative alla predilezione. La Figura 5.9 mostra un'architettura con accesso non uniforme alla memoria (NUMA) in cui la CPU ha un accesso più rapido ad alcune zone di memoria rispetto ad altre. Di solito una situazione di questo tipo si ha in sistemi dove sono presenti diverse schede, ognuna con CPU e memoria proprie. Le CPU su una scheda possono accedere alla memoria sulla stessa scheda con meno ritardo rispetto a quella su schede diverse. Se lo sche-

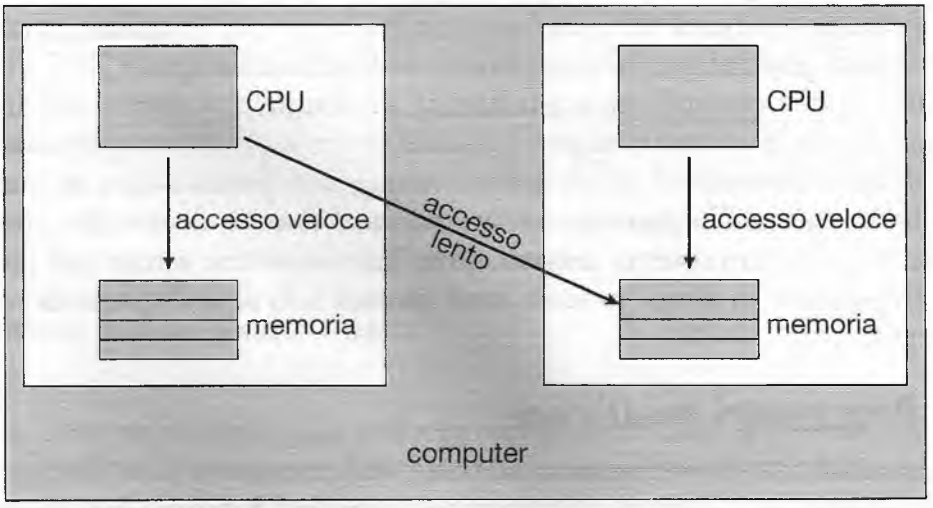


Figura 5.9 NUMA e lo scheduling della CPU.

duler della CPU di un sistema operativo e gli algoritmi di allocazione della memoria lavorano insieme, allora un processo con predilezione per una determinata CPU può essere allocato nella memoria residente sulla stessa scheda in cui è montata la CPU. Questo esempio mostra anche come i sistemi operativi non siano definiti e implementati in maniera tanto nitida quanto è descritto nei libri di testo. Le linee di demarcazione tra le componenti di un sistema operativo, lungi dall'essere nette, sono sfumate; opportuni algoritmi instaurano poi connessioni fra le varie parti allo scopo di ottimizzare le prestazioni e l'affidabilità.

5.5.3 Bilanciamento del carico

Sui sistemi SMP è importante che il carico di lavoro sia distribuito equamente tra tutti i processori per sfruttare appieno i vantaggi della multielaborazione. Se ciò non avviene, alcuni processori potrebbero restare inattivi mentre altri verrebbero intensamente sfruttati con una coda di processi in attesa. Il **bilanciamento del carico** tenta di ripartire il carico di lavoro uniformemente tra tutti i processori di un sistema SMP. Bisogna notare che è necessario, di norma, solo nei sistemi in cui ciascun processore detiene una coda privata di processi passibili di esecuzione. Nei sistemi che mantengono una coda comune, il bilanciamento del carico è sovente superfluo: un processore inattivo passerà immediatamente all'esecuzione di un processo dalla coda comune eseguibile dei processi. Va ricordato, tuttavia, che in quasi tutti i sistemi operativi attuali predisposti per la SMP, ogni processore è effettivamente dotato di una propria coda di processi eseguibili.

Il bilanciamento del carico può seguire due approcci: la **migrazione guidata** (*push migration*) e la **migrazione spontanea** (*pull migration*). La prima prevede che un processo apposito controlli periodicamente il carico di ogni processore: nell'ipotesi di una sproporzione, riporterà il carico in equilibrio, spostando i processi dal processore saturo ad altri più liberi, o inattivi. La migrazione spontanea, invece, si ha quando un processore inattivo sottrae ad un processore sovraccarico un processo in attesa. I due tipi di migrazione non sono mutuamente esclusivi, e trovano spesso applicazione contemporanea nei sistemi con bilanciamento del carico. Lo scheduler Linux, per esempio, che vedremo nel Paragrafo 5.6.3, e lo scheduler ULE dei sistemi FreeBSD, si avvalgono di entrambe le tecniche. Linux esegue il proprio algoritmo di bilanciamento del carico a intervalli di 200 millisecondi (migrazione guidata), e ogniqualvolta si svuoti la coda di esecuzione di un processore (migrazione spontanea).

Una singolare proprietà del bilanciamento del carico è di annullare, spesso, il vantaggio derivante dalla predilezione del processore, analizzata nel Paragrafo 5.5.2. Il vantaggio di poter eseguire un processo dall'inizio alla fine sul medesimo processore è che in tal modo la memoria cache contiene i dati necessari per quel processo. Con la migrazione dei processi necessaria al bilanciamento del carico, questo vantaggio si perde. Anche in questo frangente, come di consueto nell'ingegneria dei sistemi, non vi sono dogmi che prescrivano una strategia ottima: in alcuni sistemi, un processore inattivo sottrae sempre un processo da un processore impegnato; in altri, i processi sono spostati solo se la disparità del carico supera una data soglia.

5.5.4 Processori multicore

Tradizionalmente i sistemi SMP hanno reso possibile la concorrenza tra thread con l'utilizzo di diversi processori fisici. Tuttavia, la tendenza recente nel progetto di processori è di inserire più unità di calcolo in un unico chip fisico, dando origine a un **processore multicore**. Ogni unità di calcolo ha i registri che le servono per conservare infomazioni sul suo stato e appare dunque al sistema operativo come un processore fisico separato. I sistemi SMP che usano processori multicore sono più veloci e consumano meno energia dei sistemi in cui ciascun processore è costituito da un singolo chip.

I processori multicore possono complicare i problemi relativi allo scheduling. Proviamo a vedere che cosa può succedere. Le ricerche hanno permesso di scoprire che quando un processore accede alla memoria, una quantità significativa di tempo trascorre in attesa della disponibilità dei dati. Questa situazione, nota come **stallo della memoria**, può verificarsi per varie ragioni, come ad esempio la mancanza dei dati richiesti nella cache. La Figura 5.10 mostra uno stallo della memoria. In questo scenario, il processore può trascorrere fino al 5 per cento del suo tempo attendendo che i dati siano disponibili in memoria. Per rimediare a questa situazione, molti dei progetti hardware recenti implementano delle unità di calcolo multithread in cui due o più thread hardware sono assegnati a una singola unità di calcolo. In questo modo, se un thread è in situazione di stallo in attesa della memoria, l'unità di calcolo può passare il controllo a un altro thread. La Figura 5.11 mostra un processore a due thread nel quale l'esecuzione dei thread 0 e 1 sono interfogliate (*interleaved*). Dal punto di vista del sistema operativo, ogni thread hardware appare come un processore logico in grado di eseguire un thread software. In un sistema a due thread con due unità di calcolo il sistema operativo vede dunque quattro processori logici. La CPU UltraSPARC T1 monta otto unità di calcolo per singolo chip, e quattro thread hardware per ogni unità di calcolo; dalla prospettiva del sistema operativo si vedono 32 processori logici.

In generale, ci sono due modi per rendere un processore multithread: attraverso il **multithreading grezzo** (*coarse-grained*) o il **multithreading fine** (*fine-grained*). Nel multithreading grezzo un thread resta in esecuzione su un processore fino al verificarsi di un evento a

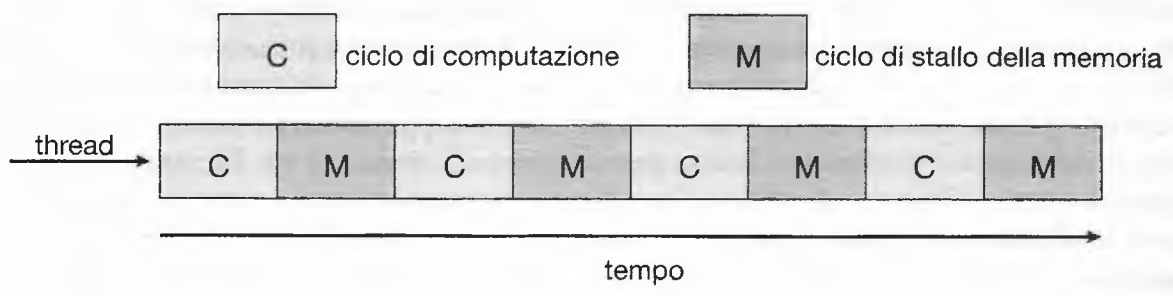


Figura 5.10 Stallo della memoria.

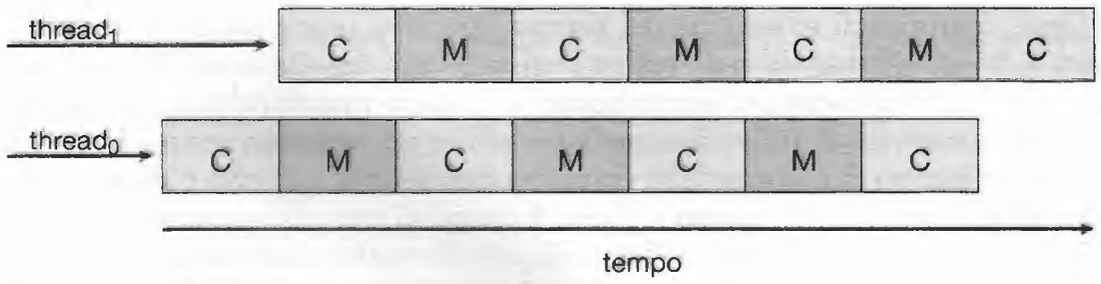


Figura 5.11 Sistema multicore e multithread.

lunga latenza, come ad esempio uno stallo di memoria. A causa dell'attesa introdotta dall'evento a lunga latenza, il processore deve passare a un altro thread e iniziare a eseguirlo. Tuttavia, il costo per cambiare il thread in esecuzione è alto, perché occorre ripulire la pipeline delle istruzioni prima che il nuovo thread possa iniziare a essere eseguito sull'unità di calcolo. Quando il nuovo thread è in esecuzione inizia a riempire la pipeline con le sue istruzioni. Il multithreading fine (anche detto *multithreading interfogliato*) passa da un thread a un altro con un livello molto più fine di granularità (tipicamente al termine di un ciclo di istruzione). Tuttavia, il progetto di sistemi a multithreading fine include una logica dedicata al cambio di thread. Ne risulta così che il costo del passaggio da un thread a un altro è piuttosto basso.

Va osservato che un processore multithread e multicore richiede due diversi livelli di scheduling. A un primo livello vi sono le decisioni di scheduling che devono essere intraprese dal sistema operativo per stabilire quale thread software mandare in esecuzione su ciascun thread hardware (processore logico). Per realizzare questo tipo di scheduling il processore può scegliere qualunque algoritmo, ad esempio quelli descritti nel Paragrafo 5.3. Un secondo livello di scheduling specifica come ogni unità di calcolo decida quale thread hardware eseguire. Ci sono diverse strategie che si possono adottare in questa situazione. Il processore UltraSPARC T1, menzionato poche righe fa, usa un semplice algoritmo circolare (round-robin) per lo scheduling dei quattro thread hardware su ogni unità di calcolo. Un altro esempio ci è dato da Intel Itanium, una CPU dual-core con due thread hardware per ogni unità di calcolo. Un valore dinamico di *urgency*, compreso tra 0 e 7, dove 0 rappresenta l'urgenza minore e 7 la più alta, viene assegnato a ogni thread hardware del processore. L'Itanium identifica cinque diversi eventi che possono far scattare un cambio di thread. Quando uno di questi eventi si verifica, la logica preposta al cambio di thread confronta l'urgenza dei thread che dovrebbero scambiarsi e sceglie di mandare in esecuzione sull'unità di calcolo il thread con il valore più alto.

5.5.5 Virtualizzazione e scheduling

Un sistema dotato di virtualizzazione, anche se a singola CPU, agisce spesso come un sistema multiprocessore. La virtualizzazione software offre una o più CPU virtuali a ogni macchina virtuale in esecuzione sul sistema, e quindi pianifica l'utilizzo della CPU fisica condivisa dalle macchine virtuali. Le sostanziali differenze tra le varie tecnologie di virtualizzazione rendono difficile sintetizzare gli effetti della virtualizzazione sullo scheduling (si veda il Paragrafo 2.8). In generale, comunque, la maggior parte degli ambienti di virtualizzazione ha un sistema operativo ospitante e diversi sistemi ospiti. Il sistema operativo ospitante crea e gestisce le macchine virtuali, e ogni macchina virtuale ha un sistema operativo ospite installato con applicazioni in esecuzione su di esso. Ogni sistema operativo ospite può essere messo a punto per usi specifici, per particolari applicazioni e utenti e anche per il tempo ripartito (*time sharing*) o le operazioni real-time.

Ogni algoritmo di scheduling del sistema operativo ospite che fa assunzioni sulla quantità di lavoro effettuabile in un tempo prefissato verrà negativemente influenzato dalla virtualizzazione. Si consideri un sistema operativo a tempo ripartito che prova a suddividere il tempo in intervalli di 100 millisecondi, per offrire agli utenti un tempo di risposta ragionevole. All'interno di una macchina virtuale questo sistema operativo è alla mercé del sistema di virtualizzazione per quanto riguarda il tempo di CPU che gli viene assegnato. Un intervallo di tempo fissato in 100 millisecondi può diventare nella realtà molto più lungo dei 100 millisecondi di tempo di CPU virtuale. A seconda di quanto è occupato il sistema, i 100 millisecondi possono anche diventare un secondo di tempo reale o più, con il risultato che il tempo di risposta per gli utenti che lavorano sulla macchina virtuale sarà insoddisfacente. Gli effetti su un sistema real-time, poi, sarebbero catastrofici.

Il risultato finale di questi livelli di scheduling è che i singoli sistemi operativi virtualizzati sfruttano a loro insaputa solo una parte dei cicli di CPU disponibili, mentre effettuano lo scheduling come se sfruttassero tutti i cicli fisicamente disponibili. Tipicamente, l'orologio all'interno di una macchina virtuale fornisce valori sbagliati, perché i contatori impiegano più tempo a scattare di quanto farebbero su una CPU dedicata. La virtualizzazione può quindi vanificare i benefici di un buon algoritmo di scheduling del sistema operativo ospitato sulla macchina virtuale.

5.6 Esempi di sistemi operativi

Procediamo ora nella descrizione dei criteri di scheduling per i sistemi operativi Solaris, Windows XP e Linux. Sarà bene tenere presente che, per Solaris e Linux, lo scheduling in questione è relativo ai thread a livello kernel. Si rammenti inoltre che Linux non distingue tra processi e thread; di conseguenza, relativamente allo scheduler di Linux, useremo il termine **task** (*compito*).

5.6.1 Esempio: scheduling di Solaris

Solaris utilizza uno scheduling dei thread basato sulle priorità in cui ogni thread appartiene a una delle sei seguenti classi.

1. Tempo ripartito (TS).
2. Interattivo (IA).
3. Real-time (RT).
4. Sistema (SYS).
5. Ripartizione equa (FSS, per *fair share*).
6. Priorità fissa (FP).

All'interno di ciascuna classe vi sono priorità e algoritmi di scheduling differenti.

La classe di scheduling predefinita per i processi è quella a tempo ripartito. È basata su un criterio di scheduling che modifica dinamicamente le priorità, assegnando porzioni di tempo variabili, grazie a una coda multipla con retroazione. Per default, tra le priorità e le frazioni di tempo sussiste una relazione inversa: più alta è la priorità, minore la frazione di tempo associata; più bassa è la priorità, maggiore sarà la frazione di tempo. Di solito, i processi interattivi hanno priorità alta, mentre i processi con prevalenza d'elaborazione hanno priorità bassa. Questo criterio di scheduling offre un buon tempo di risposta per i processi interattivi e una buona produttività per i processi con prevalenza d'elaborazione. La classe

priorità	quanto di tempo	quanto di tempo esaurito	ripresa dell'attività
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Figura 5.12 Tabella di dispatch di Solaris per i thread interattivi e a tempo ripartito.

interattiva e quella a tempo ripartito utilizzano gli stessi criteri di scheduling, ma la prima privilegia le applicazioni dotate di interfacce a finestre, a cui attribuisce priorità più elevate per ottenere prestazioni migliori.

La Figura 5.12 mostra la tabella di dispatch per lo scheduling dei thread interattivi e a tempo ripartito. Queste due classi contemplano 60 livelli di priorità; ne elenchiamo solo alcuni. La tabella di dispatch della Figura 5.12 contiene i seguenti campi.

- **Priorità.** È la priorità dipendente dalle classi, nel caso di quelle interattiva e a tempo ripartito. Il valore cresce al crescere della priorità.
- **Quanto di tempo.** È il quanto di tempo della relativa priorità. Come si può notare, priorità e frazioni di tempo sono inversamente correlate: alla priorità 0, infatti, spetta il quanto di tempo più lungo (200 millisecondi), mentre alla priorità 59, cioè la più alta, corrisponde il quanto minimo (20 millisecondi).
- **Quanto di tempo esaurito.** È la nuova priorità dei thread che abbiano consumato l'intero quanto di tempo loro assegnato senza sospendersi. Tali thread sono considerati a prevalenza d'elaborazione; le loro priorità, come evidenziato dalla tabella, subiscono una diminuzione.
- **Ripresa dell'attività.** È la priorità di un thread che ritorni in attività dopo un periodo di attesa (dell'I/O, per esempio). Come si può vedere nella tabella, quando è disponibile l' I/O per un thread in attesa, la priorità del thread assume un valore compreso tra 50 e 59. Si implementa così il criterio di scheduling che consiste nel fornire risposte sollecite ai processi interattivi.

La classe dei thread real-time ha la priorità maggiore. Ciò fa sì che i processi real-time abbiano la garanzia di ottenere una risposta dal sistema entro limiti di tempo prefissati. Un

processo real-time sarà eseguito prima di un processo appartenente a qualsiasi altra classe. In generale, tuttavia, pochi processi appartengono alla classe real-time.

Solaris sfrutta la classe sistema per eseguire i processi del kernel, quali lo scheduler e il demone per la paginazione. La priorità di un processo del sistema, una volta fissata, non cambia. La classe sistema è riservata all'uso da parte del kernel (i processi utenti eseguiti in modalità di sistema non appartengono alla classe sistema).

Le classi a priorità fissa e a ripartizione equa sono state introdotte in Solaris 9. I thread appartenenti alla classe a priorità fissa hanno lo stesso livello di priorità di quelli della classe a tempo ripartito, ma le loro priorità non vengono modificate dinamicamente. Per la classe a ripartizione equa le decisioni di scheduling vengono prese sulla base delle **quote** (*shares*) di CPU, e non sulla base delle proprietà. Le quote di CPU sono assegnate a un insieme di processi, chiamato **progetto**, e indicano in che misura il progetto ha diritto all'uso delle risorse disponibili.

Ogni classe di scheduling include una scala di priorità. Tuttavia, lo scheduler converte le priorità specifiche della classe in priorità globali e sceglie per l'esecuzione il thread con la priorità globale più elevata. La CPU esegue il thread prescelto finché (1) si blocca, (2) esaurisce la propria frazione di tempo, o (3) è soggetto a prelazione da un thread con priorità più alta. Se vi sono più thread con la stessa priorità, lo scheduler utilizza una coda circolare RR. La Figura 5.13 mostra le relazioni fra le sei classi di scheduling, e quali sono le priorità globali a loro assegnate. Va notato che il kernel utilizza 10 thread per servire le interruzioni. Questi thread non appartengono ad alcuna classe e sono eseguiti con massima priorità (160-169). Come già ricordato, tradizionalmente Solaris utilizzava il modello da molti a molti (Paragrafo 4.2.3), ma a partire da Solaris 9 è passato al modello da uno a uno (Paragrafo 4.2.2).

5.6.2 Esempio: scheduling di Windows XP

Il sistema operativo Windows XP compie lo scheduling dei thread servendosi di un algoritmo basato su priorità e prelazione. Lo scheduler assicura che si eseguano sempre i thread a più alta priorità. La porzione del kernel che si occupa dello scheduling si chiama *dispatcher*. Una volta selezionato dal *dispatcher*, un thread viene eseguito finché non sia sottoposto a prelazione da un altro thread a priorità più alta oppure termini, esaurisca il suo quanto di tempo o esegua una chiamata di sistema bloccante, per esempio un'operazione di I/O. Se un thread d'elaborazione in tempo reale, ad alta priorità, entra nella coda dei processi pronti per l'esecuzione mentre è in esecuzione un thread a bassa priorità, quest'ultimo viene sottoposto a prelazione. Ciò realizza un accesso preferenziale alla CPU per i thread d'elaborazione in tempo reale che ne hanno necessità.

Per determinare l'ordine d'esecuzione dei thread il *dispatcher* impiega uno schema di priorità a 32 livelli. Le priorità sono suddivise in due classi: la **classe variable** raccoglie i thread con priorità da 1 a 15, mentre la **classe real-time** raccoglie i thread con priorità tra 16 e 31 (esiste anche un thread, per la gestione della memoria, che si esegue con priorità 0). Il *dispatcher* adopera una coda per ciascuna priorità di scheduling e percorre l'insieme delle code da quella a priorità più alta a quella a priorità più bassa, finché trova un thread pronto per l'esecuzione. In assenza di tali thread, il *dispatcher* fa eseguire un thread speciale detto **idle thread**.

C'è una relazione tra le priorità numeriche del kernel del sistema operativo Windows XP e quelle dell'API Win32. Secondo l'API Win32 un processo può appartenere a diverse classi di priorità, tra cui le seguenti:

- REALTIME_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- ABOVE_NORMAL_PRIORITY_CLASS

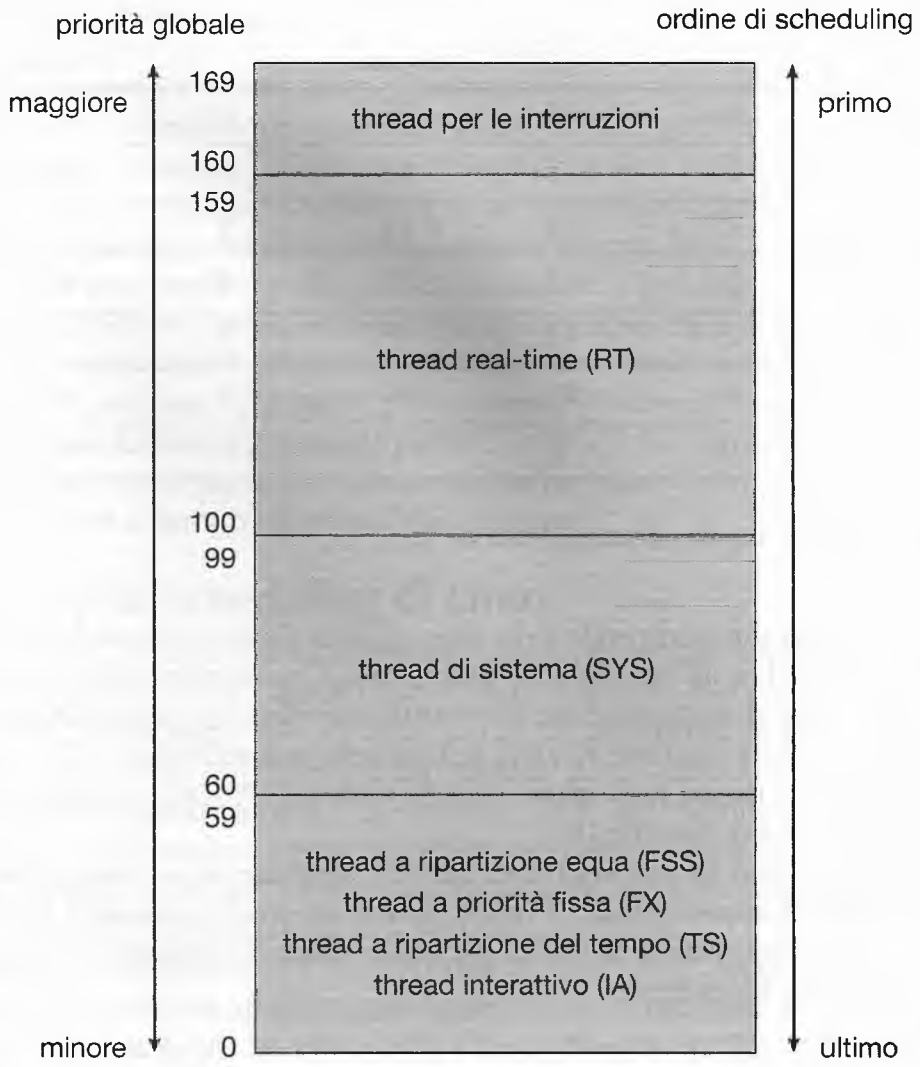


Figura 5.13 Scheduling di Solaris.

- NORMAL_PRIORITY_CLASS
- BELOW_NORMAL_PRIORITY_CLASS
- IDLE_PRIORITY_CLASS.

Le priorità di ciascuna classe eccetto REALTIME_PRIORITY_CLASS sono priorità di classe variabile, quindi la priorità di un thread appartenente a queste classi può cambiare.

Ciascuna di queste classi prevede delle priorità relative, i cui valori comprendono i seguenti:

- TIME_CRITICAL
- HIGHEST
- ABOVE_NORMAL
- NORMAL
- BELOW_NORMAL
- LOWEST
- IDLE.

	realtime	high	above_ normal	normal	below_ normal	idle_ priority
time_critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above_normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below_normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figura 5.14 Priorità nel sistema operativo Windows XP.

La priorità di ciascun thread dipende dalla priorità della classe cui appartiene e dalla priorità relativa che il thread ha all'interno della stessa classe. Questa relazione è rappresentata nella Figura 5.14. I valori di ciascuna classe di priorità sono riportati nella prima riga in alto. La prima colonna a sinistra contiene i valori delle diverse priorità relative. Per esempio, se la priorità relativa di un thread nella classe ABOVE_NORMAL_PRIORITY_CLASS è NORMAL, la priorità numerica di quel thread è 10.

Inoltre, ogni thread ha una priorità di base che rappresenta un valore nell'intervallo di priorità della classe di appartenenza. Il valore predefinito per la priorità di base in una classe è quello della priorità relativa NORMAL per quella classe. Le priorità di base per ciascuna classe di priorità sono le seguenti:

- REALTIME_PRIORITY_CLASS–24
- HIGH_PRIORITY_CLASS–13
- ABOVE_NORMAL_PRIORITY_CLASS–10
- NORMAL_PRIORITY_CLASS–8
- BELOW_NORMAL_PRIORITY_CLASS–6
- IDLE_PRIORITY_CLASS–4.

Di solito, i processi appartengono alla classe NORMAL_PRIORITY_CLASS, sempre che il processo genitore non appartenga alla classe IDLE_PRIORITY_CLASS, o sia stata specificata un'altra classe alla creazione del processo. Di solito la priorità iniziale di un thread è la priorità di base del processo a cui il thread appartiene.

Quando il quanto di tempo di un thread si esaurisce, il thread viene interrotto e se il thread fa parte della classe a priorità variabile, la sua priorità viene ridotta. Tuttavia, la priorità non si abbassa mai oltre la priorità di base. L'abbassamento della priorità tende a limitare l'uso della CPU da parte dei thread con prevalenza d'elaborazione. Se un thread a priorità variabile è rilasciato da un'operazione d'attesa, il *dispatcher* aumenta la sua priorità. L'entità di questo aumento dipende dal tipo d'evento che il thread attendeva: un thread che attendeva dati dalla tastiera riceve un forte aumento di priorità, uno che attendeva operazioni relative a un disco riceve un aumento più moderato. Questa strategia mira a fornire buoni tempi di risposta per i thread interattivi, con interfacce basate su mouse e finestre; permette

inoltre ai thread con prevalenza di I/O di tenere occupati i dispositivi di I/O, e rende nel contempo possibile l'utilizzo con esecuzione in sottofondo dei cicli di CPU inutilizzati da parte dei thread con prevalenza d'elaborazione. Questa strategia si segue in molti sistemi operativi a tempo ripartito d'elaborazione, compreso UNIX. Inoltre, per migliorare il tempo di risposta, la finestra attraverso cui l'utente sta interagendo ottiene un incremento di priorità.

Quando un utente richiede l'esecuzione di un programma interattivo, il sistema deve fornire al relativo processo prestazioni particolarmente elevate. Per questa ragione, il sistema Windows XP segue una regola specifica di scheduling per i processi della classe NORMAL_PRIORITY_CLASS. Il sistema operativo Windows XP distingue tra il *processo in primo piano*, correntemente selezionato sullo schermo e i *processi in sottofondo*, che non sono attualmente selezionati. Quando un processo passa in primo piano, Windows XP aumenta il suo quanto di tempo di un certo fattore, tipicamente pari a 3; ciò fa sì che il processo in primo piano possa continuare la propria esecuzione per un tempo tre volte più lungo, prima che si abbia una prelazione dovuta al tempo ripartito d'elaborazione.

5.6.3 Esempio: scheduling di Linux

Prima della versione 2.5, il kernel di Linux impiegava, per lo scheduling, una variante dell'algoritmo tradizionale di UNIX. Quest'ultimo, però, comporta due problemi: non fornisce strumenti adeguati per i sistemi SMP, e risponde male al crescere dell'ordine di grandezza del numero dei task nel sistema. A partire dalla versione 2.5, il kernel offre un algoritmo di scheduling che gira in tempo costante – in simboli, $O(1)$ – a prescindere dal numero di task nel sistema. Il nuovo scheduler è anche migliorato nei confronti della SMP: permette infatti la gestione della predilezione per il processore, rende possibile il bilanciamento del carico, e offre inoltre strumenti volti a garantire equità ai task interattivi.

Lo scheduler di Linux ricorre a un algoritmo di scheduling con prelazione, basato sulle priorità, con due gamme di priorità separate: un intervallo **real-time** che va da 0 a 99 e un intervallo detto **nice** compreso tra 100 e 140. Queste due gamme sono poi tradotte all'interno di una scala globale di priorità, in cui i valori numericamente più bassi rappresentano le priorità più alte.

A differenza degli scheduler di molti altri sistemi, come Solaris (Paragrafo 5.6.1) e Windows XP (Paragrafo 5.6.2), Linux assegna ai task con priorità più alta porzioni di tempo più cospicue e a quelle dalla priorità più bassa quanti di tempo più brevi. La relazione tra priorità e lunghezza del quanto di tempo è mostrata dalla Figura 5.15.

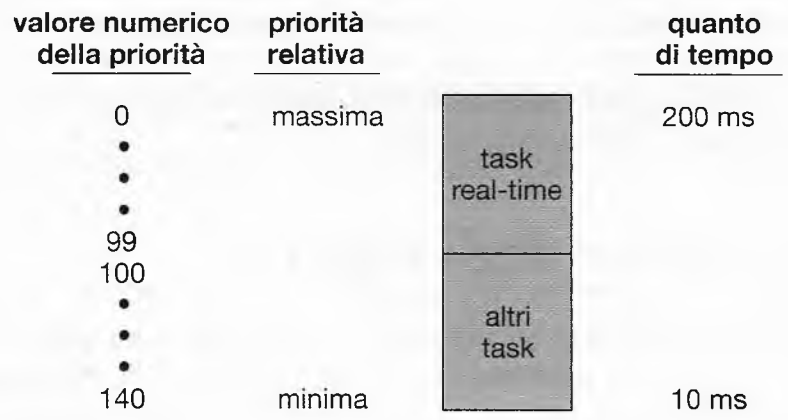


Figura 5.15 Relazione fra le priorità e la lunghezza del quanto di tempo.

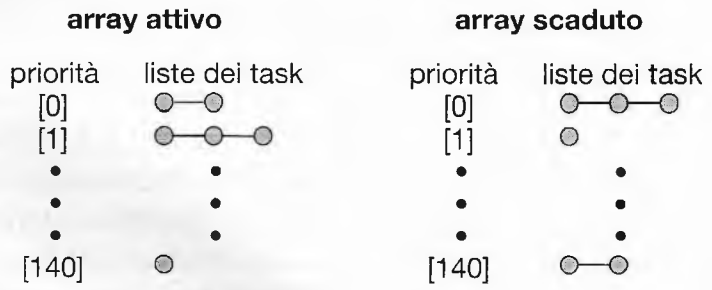


Figura 5.16 Liste dei task indicizzate sulla base della priorità.

Un task pronto per l'esecuzione è considerato eseguibile dalla CPU se non ha ancora consumato per intero il proprio quanto di tempo. Allorché esaurisce tale quanto, il task è considerato scaduto, e non può più essere posto in esecuzione finché tutti gli altri task non abbiano consumato la propria porzione di tempo. Il kernel elenca i task pronti per l'esecuzione in una struttura dati detta **runqueue** (*coda di esecuzione*). Per garantire la possibilità di SMP, ciascun processore mantiene la propria coda di esecuzione e opera una pianificazione autonoma. Ogni coda di esecuzione contiene due array di priorità: **attivo** e **scaduto**. Il primo contiene tutti i task che hanno ancora tempo da sfruttare, mentre il secondo elenca i task scaduti. Entrambi gli array contengono una lista di task, ordinati progressivamente secondo la priorità (Figura 5.16). Lo scheduler sceglie il task con la priorità più alta dall'array attivo affinché sia eseguito dalla CPU. Sulle macchine dotate di più processori, ogni processore mantiene una propria coda di esecuzione, e seleziona il task successivo da eseguire. Una volta che tutti i task nell'array attivo abbiano consumato il loro quanto di tempo (cioè, quando l'array attivo è vuoto), gli array delle priorità si scambiano i ruoli: quello scaduto diventa quello attivo e viceversa.

Linux implementa lo scheduling real-time come definito dallo standard POSIX.1b; si veda il Paragrafo 5.4.2 per un'analisi approfondita. I task real-time ricevono priorità statiche, mentre gli altri task hanno priorità dinamiche, basate sul loro valore *nice*, cui si aggiunge o si sottrae il valore 5. Il grado di interattività di un task determina se il valore 5 sarà sommato al valore nice o sottratto da esso. Tale grado di interattività è determinato dal tempo trascorso dal task in attesa prima che il suo I/O sia disponibile. I task maggiormente interattivi hanno, in genere, tempi di attesa più lunghi; poiché lo scheduler dà loro preferenza, la probabilità che essi ottengano un bonus vicino a -5 è alta. Questi adattamenti hanno l'effetto di attribuire priorità più alte per i task interattivi. Al contrario, i task con tempi di attesa più brevi sono spesso a prevalenza di elaborazione, e dunque vedranno diminuire la loro priorità.

La priorità dinamica di un task è ricalcolata nel momento in cui abbia esaurito la propria porzione di tempo e debba passare dall'array attivo a quello scaduto. Così, quando i due array si scambiano i ruoli, tutti i task del nuovo array attivo avranno ricevuto nuove priorità, con i relativi quanti di tempo.

5.7 Valutazione degli algoritmi

Ci si può chiedere come scegliere un algoritmo di scheduling della CPU per un sistema particolare. Come abbiamo visto nel Paragrafo 5.3, esistono molti algoritmi di scheduling, ciascuno dotato dei propri parametri; quindi, la scelta di un algoritmo può essere abbastanza difficile.

Il primo problema da affrontare riguarda la definizione dei criteri da usare per la scelta dell'algoritmo. Nel Paragrafo 5.2 si spiega che i criteri si definiscono spesso nei termini dell'utilizzo della CPU, del tempo di risposta o della produttività. Per scegliere un algoritmo occorre innanzitutto stabilire l'importanza relativa di queste misure. Tra i criteri suggeriti si possono inserire diverse misure, per esempio le seguenti:

- rendere massimo l'utilizzo della CPU con il vincolo che il massimo tempo di risposta sia 1 secondo;
- rendere massima la produttività in modo che il tempo di completamento sia (in media) linearmente proporzionale al tempo d'esecuzione totale.

Una volta definiti i criteri di selezione, è necessario valutare gli algoritmi considerati. Di seguito si descrivono i vari metodi di valutazione.

5.7.1 Modelli deterministici

Fra i metodi di valutazione sono di grande importanza quelli che rientrano nella classe della valutazione analitica. La **valutazione analitica**, secondo l'algoritmo dato e il carico di lavoro del sistema, fornisce una formula o un numero che valuta le prestazioni dell'algoritmo per quel carico di lavoro.

La definizione e lo studio di un **modello deterministico** è un tipo di valutazione analitica, che considera un carico di lavoro predeterminato e definisce le prestazioni di ciascun algoritmo per quel carico di lavoro.

Si supponga, per esempio, di avere il carico di lavoro illustrato di seguito; i cinque processi si presentano al tempo 0, nell'ordine dato, e la durata delle sequenze di operazioni della CPU è espressa in millisecondi:

Processo	Durata della sequenza
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

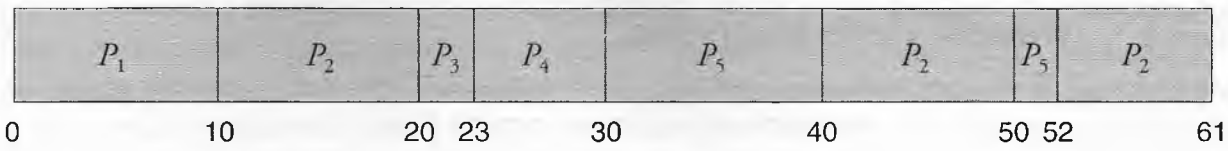
Si può stabilire con quale fra gli algoritmi di scheduling FCFS, SJF e RR (quanto di tempo = 10 millisecondi) per questo insieme di processi si ottenga il minimo tempo medio d'attesa. Con l'algoritmo FCFS i processi si eseguono secondo lo schema seguente.

Il tempo d'attesa è di 0 millisecondi per il processo P_1, di 10 millisecondi per il processo P_2, di 39 millisecondi per il processo P_3, di 42 millisecondi per il processo P_4 e di 49 millisecondi per il processo P_5. Quindi, il tempo d'attesa medio è di (0 + 10 + 39 + 42 + 49)/5 = 28 millisecondi.

Con l'algoritmo SJF senza prelazione i processi si eseguono come segue.

Il tempo d'attesa è di 10 millisecondi per il processo P_1, di 32 millisecondi per il processo P_2, di 0 millisecondi per il processo P_3, di 3 millisecondi per il processo P_4 e di 20 millisecondi per il processo P_5. Quindi, il tempo d'attesa medio è di (10 + 32 + 0 + 3 + 20)/5 = 13 millisecondi.

Con l'algoritmo RR i processi si eseguono come segue.



Il tempo d'attesa è di 0 millisecondi per il processo P_1, di 32 millisecondi per il processo P_2, di 20 millisecondi per il processo P_3, di 23 millisecondi per il processo P_4 e di 40 millisecondi per il processo P_5. Quindi, il tempo d'attesa medio è di (0 + 32 + 20 + 23 + 40)/5 = 23 millisecondi.

È importante notare come, *in questo caso*, il criterio SJF fornisca come risultato un tempo medio d'attesa minore della metà del tempo corrispondente ottenuto con lo scheduling FCFS; l'algoritmo RR fornisce un risultato intermedio tra i precedenti.

La definizione e lo studio di un modello deterministico è semplice e rapida; i risultati sono numeri esatti che consentono il confronto tra gli algoritmi. Nondimeno, anche i parametri devono essere numeri esatti e i risultati sono applicabili solo a quei casi. Il suo impiego principale consiste nella descrizione degli algoritmi di scheduling e nella presentazione d'esempi. Nei casi in cui si possono eseguire ripetutamente gli stessi programmi e si possono misurare con precisione i requisiti d'elaborazione dei programmi, i modelli deterministici sono utilizzabili per scegliere un algoritmo di scheduling. Lo studio dei modelli deterministici rispetto a un insieme d'esempi può indicare tendenze che si possono poi analizzare e verificare separatamente. Si può per esempio mostrare che per l'ambiente descritto, vale a dire tutti i processi e i relativi tempi disponibili al tempo 0, con il criterio SJF si ottiene sempre il tempo d'attesa minimo.

5.7.2 Reti di code

In molti sistemi i processi eseguiti variano di giorno in giorno, quindi non esiste un insieme statico di processi (e di tempi) da usare nei modelli deterministici. Si possono però determinare le distribuzioni delle sequenze di operazioni della CPU e delle sequenze di operazioni di I/O, poiché si possono misurare e quindi approssimare, o più semplicemente stimare. Si ottiene una formula matematica che indica la probabilità di una determinata sequenza di operazioni della CPU. Comunemente questa distribuzione è di tipo esponenziale ed è descritta dalla sua media. Analogamente, è necessario fornire anche la distribuzione degli istanti d'arrivo dei processi nel sistema. Da queste due distribuzioni si può calcolare la produttività media, l'utilizzo o il tempo d'attesa medi, e così via, per la maggior parte degli algoritmi.

Il sistema di calcolo si descrive come una rete di server, ciascuno con una coda d'attesa. La CPU è un server con la propria coda dei processi pronti, e il sistema di I/O ha le sue code dei dispositivi. Se sono noti l'andamento degli arrivi e dei servizi, si possono calcolare l'utilizzo, la lunghezza media delle code, il tempo medio d'attesa e così via. Questo tipo di studio si chiama **analisi delle reti di code** (*queueing-network analysis*).

Si consideri il seguente esempio: sia n la lunghezza media di una coda, escluso il processo correntemente servito, detti W il tempo medio d'attesa nella coda e λ l'andamento medio d'arrivo dei nuovi processi nella coda (per esempio, 3 processi al secondo); si prevede che, nel tempo W durante il quale un processo attende nella coda, raggiungano la coda $\lambda \times W$ nuovi processi. Se il sistema è stabile, il numero dei processi che lasciano la coda deve essere uguale al numero dei processi che vi arrivano; quindi,

$$n = \lambda \times W$$

Quest'equazione è nota come **formula di Little** ed è utile soprattutto perché è valida per qualsiasi algoritmo di scheduling e distribuzione d'arrivi.

La formula di Little è utilizzabile per il calcolo di una delle tre variabili, quando sono note le altre due. Per esempio, sapendo che ogni secondo arrivano 7 processi (in media), e che normalmente nella coda ne sono presenti 14, si può calcolare che il tempo medio d'attesa per ogni processo è di 2 secondi.

L'analisi delle reti di code può essere utile per il confronto degli algoritmi di scheduling, ma presenta alcuni limiti. Attualmente le classi di algoritmi e distribuzioni trattabili sono piuttosto limitate. Inoltre, poiché può essere difficile lavorare con la matematica di distribuzioni e algoritmi complicati, spesso, affinché siano trattabili matematicamente, si definiscono in modo irrealistico le distribuzioni d'arrivo e servizio. Generalmente è necessario stabilire anche un numero di presupposti indipendenti che possono non essere precisi. Per poter calcolare una risposta, le reti di code spesso si limitano ad approssimare un sistema reale, rendendo discutibile la precisione dei risultati ottenuti.

5.7.3 Simulazioni

Per riuscire ad avere una valutazione più precisa degli algoritmi di scheduling ci si può servire di **simulazioni**. Le simulazioni implicano la programmazione di un modello del sistema di calcolo; le strutture dati rappresentano gli elementi principali del sistema. Il simulatore dispone di una variabile che rappresenta un clock; con l'aumentare del valore di questa variabile, il simulatore modifica lo stato del sistema in modo da descrivere le attività dei dispositivi, dei processi e dello scheduler. Durante l'esecuzione della simulazione si raccolgono e si stampano statistiche che indicano le prestazioni degli algoritmi.

I dati necessari per condurre la simulazione si possono ottenere in vari modi. Il metodo più diffuso impiega un generatore di numeri casuali, programmato per creazione di processi, durata delle sequenze di operazioni della CPU, arrivi, conclusioni e così via; in modo conforme alle rispettive distribuzioni di probabilità, definibili matematicamente (esponenziali, uniformi, di Poisson) oppure in modo empirico. Se la distribuzione deve essere definita in modo empirico, si fanno misure sul sistema reale in esame, e si usano i risultati per definire la distribuzione effettiva degli eventi nel sistema reale.

Tuttavia, una simulazione condotta secondo la distribuzione può non essere precisa, a causa delle relazioni esistenti tra eventi successivi nel sistema reale. La distribuzione relativa alle frequenze, infatti, si limita a indicare quanti eventi di una data categoria si verificano, senza fornire informazioni sul loro ordine. Per rimediare a questo problema si può sottoporre il sistema reale a un controllo continuo, con la registrazione della sequenza degli even-

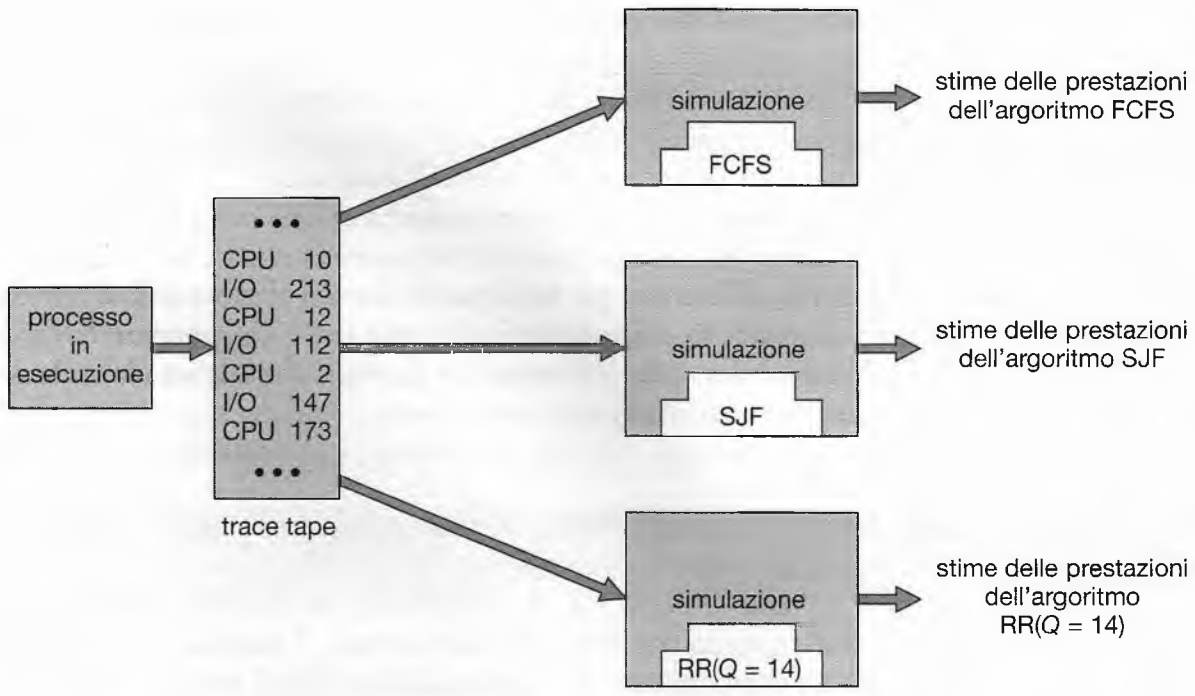


Figura 5.17 Valutazione di algoritmi di scheduling della CPU tramite una simulazione.

ti effettivi – in questo modo si ottiene un cosiddetto **trace tape** – (Figura 5.17), che poi si usa per condurre la simulazione. Si tratta di uno strumento eccellente che permette di confrontare gli algoritmi rispetto allo stesso insieme di dati reali, e con cui si possono ottenere risultati molto precisi.

Poiché spesso richiedono diverse ore del tempo d'elaborazione, le simulazioni possono tuttavia essere molto onerose. Una simulazione dettagliata dà risultati molto precisi, ma richiede anche una gran quantità di tempo, e molto spazio di memoria per la registrazione degli eventi. Inoltre, la progettazione, la codifica e la messa a punto di un simulatore possono essere un compito assai impegnativo.

5.7.4 Realizzazione

Persino una simulazione ha dei limiti per quel che riguarda la precisione. L'unico modo assolutamente sicuro per valutare un algoritmo di scheduling consiste nel codificarlo, inserirlo nel sistema operativo e osservarne il comportamento nelle reali condizioni di funzionamento del sistema.

Il problema principale di questo metodo è il suo costo: le spese non sono dovute solo alla codifica dell'algoritmo e alle modifiche da fare al sistema operativo affinché possa gestire l'algoritmo con le sue strutture dati, ma anche alle reazioni degli utenti a fronte di costanti modifiche del sistema operativo. Alla maggior parte degli utenti non interessa la realizzazione di un sistema operativo migliore, ma soltanto eseguire i processi e usare i risultati. Un sistema operativo che si trovi costantemente in fasi di modifica e messa a punto non aiuta gli utenti a svolgere il loro lavoro.

Un'altra difficoltà da affrontare per fare qualsiasi valutazione di un algoritmo è il cambiamento dell'ambiente in cui lo si usa. L'ambiente non cambia solo nel modo consueto, cioè per la scrittura di nuovi programmi e nuovi problemi che si possono riscontrare, ma si modifica anche in seguito alle prestazioni dello scheduler. Se si dà la priorità ai processi brevi, gli utenti possono suddividere i processi più lunghi in gruppi di processi brevi. Se si dà la

priorità ai processi interattivi rispetto ai processi non interattivi, gli utenti possono passare all'uso interattivo.

Per esempio, alcuni ricercatori progettarono un sistema che classificava automaticamente i processi nelle categorie interattiva e non interattiva sulla base della quantità di I/O eseguita dal o verso il terminale. I processi che non leggessero o scrivessero sul terminale per un intero secondo erano classificati come non interattivi, e conseguentemente spostati in una coda a bassa priorità. Un programmatore reagì a questa strategia modificando i suoi programmi di modo che scrivessero sul terminale un carattere a intervalli regolari di meno di un secondo. Il risultato fu che le applicazioni del programmatore ricevettero alta priorità dal sistema, sebbene il loro output fosse del tutto privo di senso.

Gli algoritmi di scheduling più flessibili sono quelli che possono essere tarati dagli amministratori del sistema o dai suoi utenti in modo da adattarsi a una specifica gamma di applicazioni. Per esempio, le macchine impegnate in applicazioni grafiche all'avanguardia avranno necessità di scheduling diverse da quelle di un server web o di un file server. Alcuni sistemi operativi, e in particolare diverse versioni di UNIX, danno all'amministratore la possibilità di calibrare con grande precisione i parametri di scheduling in vista di particolari configurazioni del sistema. Solaris, per esempio, offre il comando `dispadmin` per la modifica dei parametri che regolano le classi di scheduling discusse nel Paragrafo 5.6.1.

Un altro approccio è di usare delle API appropriate per modificare la priorità di processi e thread. Le API Java, POSIX e Windows offrono tali funzionalità. Questa tecnica ha però lo svantaggio che tarare un sistema o un'applicazione per migliorarne le prestazioni spesso non ha lo stesso riscontro in situazioni più generali.

5.8 Sommario

Lo scheduling della CPU consiste nella scelta di un processo dalla coda dei processi pronti cui assegnare la CPU. L'effettiva assegnazione della CPU al processo prescelto è eseguita dal dispatcher.

L'algoritmo di scheduling in ordine d'arrivo (FCFS) è il più semplice, ma può far sì che brevi processi attendano processi molto lunghi. Si dimostra che lo scheduling ottimale, che determina il minimo tempo medio d'attesa, è lo scheduling per brevità (SJF). Realizzare lo scheduling SJF è complicato, poiché è difficile prevedere la lunghezza della successiva sequenza di operazioni della CPU. L'algoritmo SJF è un caso particolare dell'algoritmo generale di scheduling per priorità, che si limita ad assegnare la CPU al processo con priorità più elevata. Sia lo scheduling per priorità sia lo scheduling SJF possono condurre a situazioni d'attesa indefinita. L'invecchiamento (*aging*) è una tecnica che si usa per impedire che avvengano tali situazioni.

Lo scheduling circolare (RR) è il più appropriato per un sistema a tempo ripartito: si assegna la CPU al primo processo della coda dei processi pronti per q unità di tempo (quanto di tempo); dopodiché si ha la prelazione della CPU e si mette il processo in fondo alla coda dei processi pronti. Il problema principale è la scelta della durata del quanto di tempo; se è troppo lungo, lo scheduling RR si riduce a uno scheduling FCFS; se è troppo breve, il carico di scheduling dovuto al tempo dei cambi di contesto diventa eccessivo.

L'algoritmo FCFS è senza prelazione; l'algoritmo RR è con prelazione; gli algoritmi SJF e con priorità possono essere sia con prelazione sia senza prelazione.

Lo scheduling a code multiple permette l'uso di diversi algoritmi per diverse classi di processi. Le più comuni sono la coda dei processi interattivi, eseguiti in primo piano, con

scheduling RR, e la coda per processi a lotti, eseguiti in sottofondo, con scheduling FCFS. Le code multiple con retroazione permettono ai processi di spostarsi da una coda all'altra.

Molti dei sistemi attuali supportano processori multipli permettendo a ciascun processore una pianificazione indipendente. In genere ogni processore mantiene una propria coda di processi pronti (o di thread), tutti disponibili all'esecuzione. Tra gli altri aspetti relativi allo scheduling in sistemi multiprocessore vi sono la predilezione, il bilanciamento del carico e lo scheduling con processori multicore e in sistemi di virtualizzazione.

I sistemi operativi che gestiscono i thread a livello kernel devono occuparsi dello scheduling dei thread, e non di quello dei processi. Tra questi vi sono il Solaris e il Windows XP; entrambi gestiscono lo scheduling dei thread impiegando un algoritmo di scheduling con diritto di prelazione, basato su priorità e che comprende i thread in tempo reale. Anche lo scheduler dei processi di Linux impiega un algoritmo basato su priorità e che prevede la gestione dei processi in tempo reale. Gli algoritmi di scheduling di questi tre sistemi operativi favoriscono generalmente i processi interattivi rispetto ai processi a lotti o con prevalenza d'elaborazione.

La vasta gamma di algoritmi di scheduling esistenti rende imperativa la disponibilità di metodi per la loro selezione. I metodi analitici sfruttano strumenti matematici per valutare le prestazioni degli algoritmi. Le simulazioni determinano le prestazioni eseguendo gli algoritmi in presenza di un insieme "rappresentativo" di processi. Va detto, però, che la simulazione può tutt'al più dare un'approssimazione delle effettive prestazioni dei sistemi. L'unica tecnica affidabile per la valutazione delle prestazioni di un algoritmo di scheduling consiste nell'implementarlo su un vero sistema, e nel misurarne le prestazioni in un contesto reale.

Esercizi pratici

5.1 Un algoritmo di scheduling della CPU stabilisce un ordine per l'esecuzione dei processi pianificati. Dati n processi da mandare in esecuzione su di un singolo processore, quanti differenti scheduling sono possibili? Date una formula in funzione di n.

5.2 Spiegate la differenza tra lo scheduling con e senza prelazione.

5.3 Supponete che i seguenti processi siano pronti per l'esecuzione agli istanti di tempo indicati nella tabella che segue. Nella tabella è anche indicata la durata della sequenza di operazioni per ogni processo. Nel rispondere alle domande seguenti utilizzate uno scheduling senza prelazione e basate le vostre decisioni sulle informazioni che avete a disposizione nel momento in cui la decisione deve essere presa.

Processo	Istante di arrivo	Durata della sequenza
P_1	0,0	8
P_2	0,4	4
P_3	1,0	1

a. Qual è il tempo di completamento medio di questi processi se si utilizza un algoritmo di scheduling FCFS?
b. Qual è il tempo di completamento medio di questi processi se si utilizza un algoritmo di scheduling SJF?
c. L'algoritmo SJF dovrebbe, in teoria, migliorare le prestazioni, ma si noti che al tempo 0 la scelta ricadrà sul processo P_1, perché non si sa ancora che presto arriveranno due processi più piccoli. Calcolate il tempo di completamento medio

ipotizzando che la CPU venga lasciata inattiva per il primo istante di tempo e che successivamente venga utilizzato l'algoritmo SJF. Ricordate che i processi P_1 e P_2 restano in attesa durante il periodo di inattività, e quindi il loro tempo di attesa può aumentare. Questo algoritmo è conosciuto come scheduling a futuro conosciuto.

5.4 Quali vantaggi si hanno nell'avere quanti di tempo di dimensioni differenti a differenti livelli in un sistema di code multiple?

5.5 Molti algoritmi di scheduling della CPU sono parametrici. L'algoritmo RR, ad esempio, richiede un parametro che specifichi l'intervallo di tempo. Le code multiple con retroazione richiedono parametri per specificare il numero di code, l'algoritmo di scheduling da utilizzare per ogni coda, il criterio usato per muovere i processi tra le code, e così via.

Questi algoritmi sono dunque classi di algoritmi (per esempio, la classe di algoritmi RR per tutti gli intervalli di tempo, e così via). Una classe di algoritmi ne può includere un'altra (per esempio, l'algoritmo FCFS è un algoritmo RR con intervallo di tempo infinito). Quale relazione intercorre (se esiste una relazione) tra le seguenti coppie di classi di algoritmi?

a. Priorità e SJF.
b. Code mutiple con retroazione e FCFS.
c. Priorità ed FCFS.
d. RR e SJF.

5.6 Supponete che un algoritmo di scheduling (a livello dello scheduling della CPU a breve termine) favorisca quei processi che hanno usato meno CPU in un passato recente. Spiegate perché questo algoritmo favorirà programmi con prevalenza di I/O senza bloccare permanentemente i programmi con prevalenza di elaborazione.

5.7 Individuate le differenze tra gli scheduling PCS e SCS.

Esercizi

5.8 Perché è importante, per lo scheduler, distinguere i programmi con prevalenza di I/O da quelli con prevalenza di elaborazione?

5.9 Considerate come le seguenti coppie di criteri per lo scheduling entrino in conflitto in certe situazioni:

a. utilizzo della CPU e tempo di risposta;
b. tempo di completamento medio e tempo di attesa massimo;
c. utilizzo dei dispositivi di I/O e utilizzo della CPU.

5.10 Considerate la formula della media esponenziale atta a predire la durata della sequenza successiva della CPU. Quali implicazioni scaturiscono dall'assegnazione dei seguenti valori ai parametri usati dall'algoritmo?

a. $\alpha = 0$ e $\tau_0 = 100$ millisecondi
b. $\alpha = 0{,}99$ e $\tau_0 = 10$ millisecondi

5.11 Considerate il seguente insieme di processi, con la durata della sequenza di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza	Priorità
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Presumiamo che i processi siano arrivati nell'ordine P_1, P_2, P_3, P_4, P_5, e siano tutti presenti al tempo 0.

a. Disegnate quattro diagrammi di Gantt che illustrino l'esecuzione di questi processi con gli algoritmi di scheduling FCFS, SJF, con priorità senza prelazione (un numero di priorità più basso indica una priorità maggiore) e RR (quanto = 1).
b. Calcolate il tempo di completamento di ciascun processo per ciascun algoritmo di scheduling di cui al punto a).
c. Calcolate il tempo d'attesa di ciascun processo per ciascun algoritmo di scheduling di cui al punto a).
d. Dite quale, fra le esecuzioni di cui al punto a), ha il minimo tempo medio d'attesa (per tutti i processi).

5.12 Quale tra i seguenti algoritmi di scheduling potrebbe generare un'attesa indefinita?

a. In ordine di arrivo (FCFS).
b. Per brevità (SJF).
c. Circolare (RR).
d. Per priorità.

5.13 Data una variante dell'algoritmo di scheduling RR in cui gli elementi della coda dei processi pronti sono puntatori ai PCB:

a. descrivete l'effetto dell'inserimento di due puntatori allo stesso processo nella coda dei processi pronti;
b. descrivete principali vantaggi e svantaggi di questo schema;
c. ipotizzate una modifica all'algoritmo RR ordinario che consenta di ottenere lo stesso effetto senza duplicare i puntatori.

5.14 Considerate un sistema su cui vengano eseguiti dieci processi con prevalenza di I/O e un processo con prevalenza di elaborazione. Supponiamo che i primi processi richiedano un'operazione di I/O ogni millisecondo di elaborazione della CPU, e che ciascuna di tali operazioni sia completata in 10 millisecondi. Ipotizzate, inoltre, che il tempo necessario per il cambio di contesto sia di 0,1 millisecondi e che tutti i processi siano operazioni a lungo termine. Calcolate l'utilizzo della CPU in presenza di scheduler circolare RR se:

a. il quanto di tempo è pari a 1 millisecondo;
b. il quanto di tempo è pari a 10 millisecondi.

5.15 Considerate un sistema che applichi un algoritmo di scheduling a code multiple. A quale strategia può ricorrere l'utente che voglia ottenere per un suo processo la massima quantità di tempo dalla CPU?

5.16 Considerate il seguente algoritmo di scheduling con diritto di prelazione, e basato su priorità variabili dinamicamente. I numeri di priorità maggiori indicano una priorità più alta. Quando un processo attende la CPU (nella coda dei processi pronti), la sua priorità varia a un tasso α; quando è in esecuzione, la sua priorità varia a un tasso β. All'ingresso nella coda dei processi pronti, si attribuisce la priorità 0 a tutti i processi. I parametri α e β si possono impostare in modo da fornire algoritmi di scheduling diversi.

a. Descrivete l'algoritmo risultante da $\beta > \alpha > 0$.
b. Descrivete l'algoritmo risultante da $\alpha < \beta < 0$.

5.17 Spiegate a che cosa sia dovuta la maggiore preferenza che i seguenti algoritmi di scheduling accordano ai processi di breve durata:

a. in ordine di arrivo (FCFS);
b. circolare (RR);
c. a code multiple con retroazione.

5.18 Usando l'algoritmo di scheduling di Windows XP, quale priorità numerica è assegnata, nei casi che seguono, a:

a. Un thread appartenente alla REALTIME_PRIORITY_CLASS con una priorità relativa HIGHEST.
b. Un thread appartenente alla NORMAL_PRIORITY_CLASS con una priorità relativa NORMAL.
c. Un thread appartenente alla HIGH_PRIORITY_CLASS con una priorità relativa ABOVE_NORMAL.

5.19 Considerate l'algoritmo di scheduling del sistema operativo Solaris per i thread a tempo ripartito.

a. Qual è il quanto di tempo (in millisecondi) per un thread con priorità 10? E per uno con priorità 55?
b. Ipotizzate che un thread con priorità 35 abbia consumato l'intera porzione di tempo riservatagli senza bloccarsi. Quale sarà la nuova priorità assegnata dallo scheduler a questo thread?
c. Poniamo che un thread con priorità 35 si blocchi per l'I/O prima che la propria porzione di tempo sia finita. Qual è la nuova priorità che lo scheduler assegnerà a questo thread?

5.20 Lo scheduler tradizionale di UNIX stabilisce una relazione inversa fra numeri e priorità: più alto è il numero, più bassa la priorità. Lo scheduler calcola *ex novo* le priorità dei processi una volta ogni secondo, usando la funzione seguente:

Priorità = (uso recente della CPU/2) + base

dove base = 60 e *uso recente della CPU* è un valore che esprime la frequenza con cui un processo si è servito della CPU dall'ultima determinazione delle priorità.

Ipotizziamo che l'uso recente della CPU per il processo P_1 sia 40, per il processo P_2 sia 18, e per il processo P_3 sia 10. Quali saranno le nuove priorità di questi tre processi? Alla luce di quanto ottenuto, lo scheduler tradizionale di UNIX aumenta o diminuisce la priorità relativa di un processo a prevalenza di elaborazione?

5.9 Note bibliografiche

Le code con retroazione furono originariamente implementate sul sistema CTSS descritto in [Corbato et al. 1962]. Questa tecnica di scheduling è stata analizzata in [Schrage 1967]. L'algoritmo basato sulle priorità con prelazione dell'Esercizio 5.16 è stato suggerito da [Kleinrock 1975].

[Anderson et al. 1989], [Lewis e Berg 1998] e [Philbin et al. 1996] analizzano lo scheduling dei thread. Lo scheduling di processori multicore è trattato da McNairy e Bhatia [2005] e Kongetira et al. [2005].

Lo scheduling basato sulle quote (*fair-share*) è trattato da [Henry 1984], [Woodside 1986], e [Kay e Lauder 1988].

Le strategie di scheduling di UNIX V sono descritte da [Bach 1987]; quelle di UNIX FreeBSD 5.2 da [McKusick et al. 2005] e Neville-Neil [2005]; quelle di Mach da [Black 1990]. Love [2005] tratta lo scheduling in Linux. I dettagli dello scheduler ULE si trovano in Roberson [2003]. Lo scheduling di Solaris è trattato in [Mauro e McDougall 2007]. [Solomon 1998] e [Solomon e Russinovich 2000] e [Russinovich e Solomon 2005] trattano dello scheduling di Windows, rispettivamente. [Butenhof 1997] e [Lewis e Berg 1998] si occupano dello scheduling in Pthread. Siddha et al. [2007] discutono gli obiettivi futuri dello scheduling in sistemi multicore.