**Parte II - Capitolo 5 - Scheduling Della CPU**

Lo ***scheduling della CPU*** è alla base dei sistemi operativi multiprogrammati. In un sistema monoprocessore si può eseguire un processo alla volta, mentre gli altri devono attendere che la CPU sia libera e possa nuovamente essere sottoposta a scheduling. Un processo è in esecuzione finchè non deve attendere un evento, generalmente il completamente di qualche richiesta di I/O. Con la multiprogrammazione si vuole tentare di impiegare questi tempi di attesa in modo produttivo: si tengono in memoria più processi e, quando un processo deve attendere un evento, il sistema operativo gli sottrae il controllo della CPU per cederlo a un altro processo.

L'esecuzione di un processo comincia con una sequenza di operazione d'elaborazione svolte dalla CPU (*CPU burst*), seguita da una sequenza di operazioni di I/O (*I/O burst*), quindi da un'altra sequenza di operazioni della CPU, e così via. L'ultima sequenza di operazioni della CPU si conclude con una richiesta al sistema di terminare l'esecuzione. Si ha quindi una sorta di ciclicità delle fasi di elaborazione, che richiedono tempi lunghi, e di I/O, che richiedono tempi più brevi. Ogni volta che 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, lo ***scheduler a breve termine*** o ***scheduler della CPU*** sceglie il processo in memoria pronto per l'esecuzione a cui assegnare la CPU. La coda dei processi pronti non è una coda in ordine d'arrivo (FIFO), ma, generalmente, è formata da elementi detti *process control block* (PCB). Se lo scheduling interviene quando un processo passa dallo stato di esecuzione allo stato di attesa o quando un processo termina, si dice che lo schema di scheduling è ***senza diritto di prelazione*** (*nonpreemptive*) o ***cooperativo*** (*cooperative*). Se lo scheduling interviene quando un processo passa dallo stato di esecuzione allo stato di pronto o passa dallo stato di attesa allo stato di pronto, si dice che lo schema dello scheduling è ***con diritto di prelazione*** (*preemptive*). Un altro elemento coinvolto nella funzione di scheduling della CPU è il ***dispatcher***. Si tratta di un modulo che passa effettivamente il controllo della CPU ai processi scelti dallo scheduler a breve termine. Poichè si attiva a ogni cambio di contesto, il dispatcher deve essere il più veloce possibile. Il tempo richiesto dal dispatcher per fermare un processo e avviare l'esecuzione di un altro è detto *latenza di dispatch*.

Per il confronto tra gli algoritmi di scheduling della CPU vengono usati confronti fra le caratteristiche che possono incidere in modo rilevante: utilizzo della CPU (la CPU deve essere più attiva possibile), produttività (la CPU è attiva quando svolge lavoro), tempo di completamento (tempo necessario per eseguire un processo), tempo d'attesa (somma degli intervalli d'attesa passati nella coda dei processi pronti) e il tempo di risposta (tempo che intercorre tra la sottomissione di una richiesta e la prima risposta prodotta). 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.

Analizziamo i più diffusi algoritmi di scheduling della CPU.

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 si assegna la CPU al processo che la richiede per primo. La realizzazione del criterio FCFS si fonda su una coda FIFO. Il tempo medio d'attesa è spesso abbastanza lungo e può variare sostanzialmente al variare della durata delle sequenze di operazioni della CPU dei processi. Per quanto riguarda le prestazioni dello scheduling FCFS in una situazione dinamica, esso può comportare un *effetto convoglio* se tutti i processi attendono che un processo lungo liberi la CPU, e 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 FCFS è senza prelazione e risulta particolarmente problematico per i sistemi a tempo ripartito, dov'è importante che un utente disponga della CPU a intervalli regolari.

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 seguenza di operazioni della CPU. Quando è disponibile, si assegna la CPU al processo che ha la lunghezza più breve della successiva sequenza di operazioni. Se due processi hanno la stessa lunghezza si applica lo scheduling FCFS. Si nota che sarebbe più appropiato il termine *shortest next CPU burst* poichè lo scheduling si esegue esaminando la lunghezza della successiva sequenza di operazioni della CPU e non la sua lunghezza totale. L'algoritmo di scheduling SJF è ottimale, cioè rende minimo il tempo di attesa medio per un dato insieme di processi. La difficoltà reale nell'algoritmo SJF consiste nel conoscere la durata della successiva richiesta della CPU. Un possibile metodo di risoluzione del problema consiste nel tentare di approssimare lo scheduling SJF in modo da predire il valore della prossima sequenza di operazioni della CPU. 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. L'algoritmo SJF può essere sia con prelazione che senza prelazione. La scelta si presenta quando alla coda dei processi pronti arriva un nuovo processo con una sequenza di operazioni più breve mentre un altro processo è ancora 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 terminare la propria sequenza di operazioni della CPU.

L'algoritmo SJF è un caso particolare del più generale algoritmo di ***scheduling per priorità***: si associa una priorità ad ogni processo e si assegna la CPU al processo con priorità più alta. Questo tipo di scheduling può essere con o 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 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 per priorità senza diritto di prelazione si limita a porre l'ultimo processo arrivato in testa alla coda dei processi pronti. Un problema importante relativo agli algoritmi di scheduling per priorità è l'*attesa indefinita* (*starvation*). Un processo pronto per l'esecuzione, ma che non disponde della CPU, si può considerare bloccato in attesa della CPU. Un algoritmo di scheduling per priorità può lasciare processi con bassa priorità nell'attesa indefinita della CPU. Una soluzione al problema è costituita dall'*invecchiamento* (*aging*) che consiste in una tecnica di aumento graduale della priorità dei processi che attendono nel sistema da parecchio tempo.

L'algoritmo di ***scheduling circolare*** (*round-robin*, *RR*) è stato progettato appositamente per i sistemi a tempo ripartito. Ciascun processo riceve una piccola quantità fissata di tempo della CPU, chiamato *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. I nuovi processi si aggiungono alla fine della coda dei processi pronti. Lo scheduler della CPU individua il primo processo della coda dei processi pronti, imposta un timer in modo che invii un segnale di 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 che: il processo ha una sequenza di operazioni della CPU della durata minore di un quanto di tempo, quindi il processo stesso rilascia volontariamente la CPU e lo scheduler passa al processo successivo della coda; la durata della sequenza delle operazioni della CPU del processo attualmente in esecuzione è più lunga del quanto di tempo, quindi, quando viene raggiunta la scadenza del quanto di tempo, il timer invia un segnale d'interruzione al sistema operativo, che esegue il 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. L'algoritmo RR è con prelazione. Le prestazioni dell'algoritmo dipendono molto dalle dimensioni del quanto di tempo. Se il quanto è molto breve, 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. Occorre tuttavia considerare l'effetto dei cambi di contesto che deve essere piccolo rispetto al quanto di tempo usato per l'algoritmo RR.

L'algoritmo di ***scheduling a code multiple*** (*multilevel queue scheduling algorithm*) suddivide la coda dei processi pronti in code distinte. I processi si assegnano in modo permanente a una coda, generalmente secondo qualche caratteristica del processo, come la quantità di memoria richiesta, la priorità, il tipo. In questa situazione è necessario avere uno scheduling tra le code per priorità fissa e con prelazione. I processi possono essere classificati per priorità nel seguente modo: processi di sistema, processi interattivi, processi interattivi di editing, processi eseguiti in sottofondo e processi degli studenti. Esiste anche la possibilità di impostare quanti di tempo per le code.

Lo ***scheduling a code multiple con retroazione*** (*multilevel feedback queue scheduling)* permette ai processi di spostarsi tra le code. Se un processo usa troppo tempo di elaborazione della CPU, viene spostato in una coda con priorità più bassa. Analogamente si può spostare in una coda con priorità più elevata un processo che attende troppo a lungo.

Si trattano ora le questioni dello scheduling che riguardano i thread a livello utente e a livello kernel. Nei sistemi che impiegano il modello da molti a uno e il modello da molti a molti, la libreria di thread pianifica l'esecuzione dei thread a livello utente su un LWP libero; si parla in questo caso di ***ambito della competizione ristretto al processo*** (*process-contention scope*, *PCS*), perchè la contesa per aggiudicarsi la CPU ha luogo fra thread dello stesso processo. Per determinare quale thread a livello kernel debba essere eseguito dalla CPU, il kernel esamina i thread di tutto il sistema; si parla allora di ***ambito della competizione allargato al sistema*** (*system-contention scope*, *SCS*), cioè tutti i thread del sistema competono per l'uso della CPU.

Fin qui l'attenzione è stata posta solo su sistemi a singolo processore. Se sono presenti più unità di elaborazione, anche il problema dello scheduling è proporzionalmente più complesso. Si considerano i sistemi in cui le unità di elaborazione sono identiche (*sistemi omogenei*): si può usare qualunque unità di elaborazione disponibile per eseguire qualsiasi processo presente nella coda. 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 i 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*** (SMP): in questo caso i processi pronti per l'esecuzione sono situati in una coda comune, oppure vi è un'apposita coda per ogni processore. Per ***predilezione per il processore*** (*processor affinity*) si intende che un processo ha una predilezione per il processore su cui è in esecuzione. Questo fenomeno è dovuto alla gestione della memoria cache, in quanto sarebbe molto costoso, nel caso in cui un processo si sposta da un processore all'altro dopo l'esecuzione, svuotare la memoria cache di un processore e riempire l'altra. La predilezione per il processore può avere varie forme. Quando il 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 un processo può migrare da un processore all'altro. Quando il sistema dispone di chiamate di sistema con cui specificare che il processo non deve cambiare processore si realizza la ***predilezione forte*** (*hard affinity*). È importante che il carico di lavoro sia distribuito equamente tra tutti i processori per sfruttare appieno i vantaggi della multiprogrammazione. Se ciò non avviene alcuni processori potrebbero restare inattivi. Il ***bilanciamento del carico*** tenta di ripartire il carico di lavoro uniformemente tra tutti i processori. Risulta necessario solo per i sistemi che presentano una coda privata per ogni processore. Questa tecnica si può eseguire attraverso due approcci: la ***migrazione guidata*** (*push migration*) prevede che un processo apposito controlli periodicamente il carico di un processore, riportando il carico in equilibrio se si verifica una sproporzione, spostando i processi dal processore saturo a quelli più liberi o inattivi; la ***migrazione spontanea*** (*pull migration*) si ha quando un processore inattivo sottraead un processore sovraccarico un processo in attesa. La tendenza recente nel progetto di processori di inserire più unità di calcolo in un unico chip fisico dando origine a processori multicore, complica i problemi relativi allo scheduling. 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. Per rimediare a questa situazione, molti progettisti hardware implementano delle unità di calcolo multithread in cui due o più thread sono assegnati a una singola unità di calcolo, in modo che se un thread è in situazione di stallo della memoria, l'unità di calcolo può passare il controllo a un altro thread. In generale ci sono due modi per rendere un processore multithread. Nel ***multithreading grezzo*** (*coarse-grained*) un thread resta in esecuzione su un processore fino al verificarsi di un evento a lunga latenza, come ad esempio uno stallo di memoria. Il ***multithreading fine*** (*fine-grained*) passa da un thread all'altro con un livello molto più fine di granularità (al termine di un ciclo d'istruzione).

- Esempi di sistemi operativi (pag. 198)

Scheduling di Solaris

Scheduling di Windows XP

Scheduling di Linux

La scelta di un algoritmo di scheduling della CPU per un sistema particolare può essere fatta valutando diverse misure per rendere massimo l'utilizzo della CPU con il vincolo che il massimo tempo di risposta sia 1 secondo e per rendere massima la produttività in modo che il tempo di completamento sia linearmente proporzionale al tempo di esecuzione totale. Una volta definiti i criteri si possono valutare gli algoritmi attraverso vari metodi di valutazione. 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. La definizione e lo studio di un modello deterministico è semplice e rapida; i risultati sono numeri esatti che consentono il confronto fra gli algoritmi. In molti sistemi i processi eseguiti variano di giorno in giorno, quindi non esiste un insieme statico di processi e tempi da usare nei modelli deterministici. Descriviamo il sistema di calcolo come una rete di server. La CPU è un server con la propria coda di processi pronti, il sistema di I/O ha le sue code di 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*). 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. Se il sistema è stabile, il numero dei processi che lascia la coda deve essere uguale al numero di processi che arrivano, quindi:

Quest'equazione nota come ***formula di Little*** è utile soprattutto perchè è valida per qualsiasi algoritmo di scheduling e distribuzione d'arrivi. 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. Durante l'esecuzione della simulazione si raccolgono e si stampano statistiche che indicano le prestazioni degli algoritmi. Anche questo metodo ha dei limiti dovuti alla mancanza di precisione. L'unico modo assolutamente sicuro per valutare un algoritmo di scheduling consiste nel codificarlo, inserirlo nel sistema operativo e osservarne il comportamento. Il problema principale è il costo.