Sistemi Operativi

Alessandro Trigolo

29 febbraio 2024

Indice

Elenco delle figure

Elenco dei codici

Todo list

Parte I
Gestione dei Processi

1 Processi

Iniziamo dalle basi. Un **processo** è un programma in esecuzione, è un'istanza del programma che viene eseguita sulla CPU. Possiamo infatti avere diverse istanze dello stesso programma, ognuna che viene eseguita indipendentemente dall'altra. Possiamo quindi dire che il programma, ovvero il file eseguibile (`.exe`) è qualcosa di passivo mentre il processo è qualcosa di **attivo**.

1.1 Allocazione in memoria

Andando un po' più in dettaglio, quando il programma è in esecuzione, questo viene eseguito in maniera sequenziale. Al processo, una volta che è eseguito, viene dedicato dello spazio in memoria dal sistema operativo. Come è possibile osservare nella figura 1, la memoria messa a disposizione dal sistema operativo è suddivisa in diverse zone, ciascuna con un particolare compito. Prima di tutto, il codice sorgente

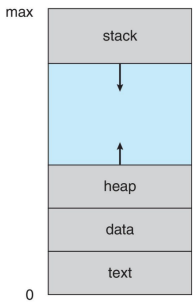


Figura 1: Spazio in memoria allocato per il processo dal sistema operativo.

del programma viene caricato nella zona **text**. Dopo di che, nella parte dedicata ai dati (**data**) vengono salvate generalmente le variabili globali, che permangono per tutta la vita del processo. Sono infine presenti due parti: lo **stack** e l'**heap** che crescono in direzione opposta. Lo stack contiene dati temporanei come variabili locali mentre l'heap è utilizzato al fine di allocare la memoria dinamicamente durante la vita del programma[1].

[1]Come abbiamo visto con C++, heap e *freestore* sono quasi dei sinonimi.

1.1.1 Process Control Block (PCB)

Ad ogni processo che è mandato in esecuzione è assegnata una particolare struttura dati dal sistema operativo, ovvero il *Process Control Block* (figura 2). Il PCB contiene diverse informazioni riguardanti il processo, in particolare:

1. Lo **stato** del processo;
2. Informazioni sul **program counter**, in particolare è importante sapere se il processo è fermato temporaneamente e poi fatto ripartire più tardi;
3. Valori dei registri, utili nel caso in cui un processo venga messo in pausa;
4. Altre informazione riguardanti lo scheduling della CPU (vedi capitolo 3), come la priorità del processo;
5. Informazioni per la gestione della memoria e dell'I/O.

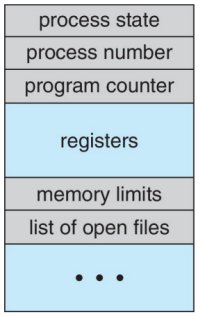


Figura 2: Rappresentazione del contenuto di un generico PCB.

In particolare, in `Linux`, nel PCB di un processo (che in Linux è chiamato `task`) sono presenti le seguenti informazioni: `pid` (numero assegnato al particolare processo), puntatori al processo genitore (che vedremo saranno utili nella fase di creazione di un processo), puntatori ai processi figli e altre informazioni come la lista dei file aperti. Quando un nuovo processo è creato in Linux, le sue informazioni sono detenute in una lista concatenata (*doubly-linked list*) dove ogni nodo della lista è il PCB di un processo specifico (figura 3). Al fine di andare a modificare delle informazioni del

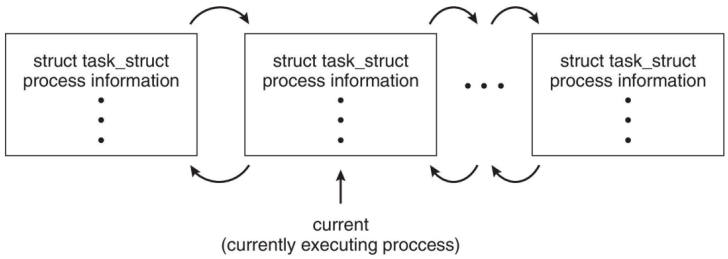


Figura 3: Lista concatenata che mantiene tutti i PCB dei processi (task) in Linux.

processo (come lo stato corrente) il sistema operativo scorre la lista e, dopo aver selezionato il PCB del processo desiderato, andrà a modificare il campo.

1.2 Stati di un processo

Durante l'intera vita del processo, questo passa in diversi stati (figura 4). I principali sono:

1. **New**: il processo è appena stato creato;
2. **Ready**: il processo è pronto per essere eseguito, quindi non è ancora stato assegnato al processore e sta aspettando l'assegnazione;
3. **Running**: dopo essere stato associato alla CPU il processo inizia ad essere eseguito. Come vedremo nel capitolo 3 e successivi, il processo può essere interrotto e quindi ritorna allo stato ready;
4. **Waiting**: se il processo deve aspettare qualche input da esterno si mette in attesa e una volta che riceve l'input ritorno nello stato ready;
5. **Terminated**: una volta che il processo finisce la sua esecuzione, questo ovviamente termina e viene rimosso dalla CPU.

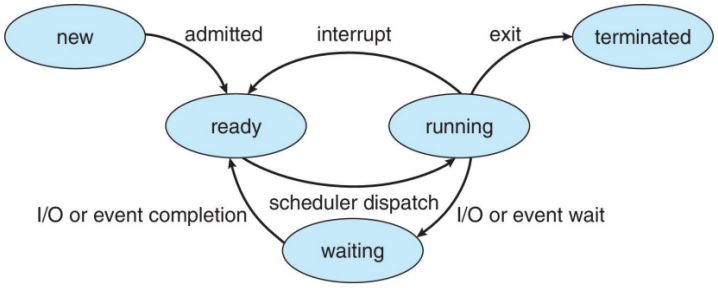


Figura 4: Gli stati della vita di un processo.

Lo stato *ready* e lo stato *wait* contengono delle code dove i processi attendono di essere eseguiti ovvero la **ready queue** e la **wait queue** che non sono altro che delle liste concatenate. Le liste contengono i PCB dei processi: il sistema operativo tiene traccia del primo e dell'ultimo processo nella lista al fine di riuscire a implementare le due code. Possiamo inoltre suddividere i processi in due macro categorie:

- ◇ **CPU bound** che sono i processi che hanno un uso massiccio della CPU;
- ◇ **I/O bound**, ovvero processi che spendono la maggior parte del loro tempo in una situazione di wait per leggere o scrivere sulle periferiche.

1.2.1 Context switch

Quando un processo A viene un rimosso dalla CPU, nel caso in cui sia stato interrotto per far spazio ad un altro processo B, è necessario salvare l'informazione del processo A in modo tale da poterlo sostituire con il processo B per poi, in un secondo momento, riuscire a ricaricare il processo A. Questa operazione è detta **context switch** (figura 5) e viene effettuata in pochi microsecondi. Ciò nonostante se è effettuata in maniera molto frequente durante l'esecuzione di diversi processi può causare molto spreco di tempo: il context switch può quindi generare un **overhead** che va sicuramente preso in considerazione negli algoritmi di scheduling (capitolo 3). È importante notare che il context switch tipicamente richiede anche un aggiuntivo utilizzo della

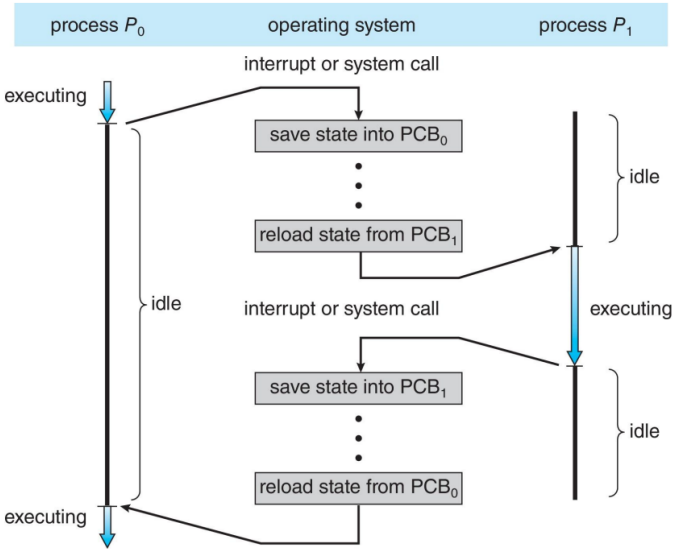


Figura 5: Il context switch.

memoria, che andremo a discutere nel capitolo **??** quando discuteremo di *paginazione* e *swapping*.

1.2.2 Creazione di un processo

Al fine di creare un processo ce ne deve sempre essere uno iniziale (*parent*) che genera il quello nuovo (*child*). Ogni nuovo processo ha un identificativo, il **pid**, che distingue univocamente il processo creato. Al momento della creazione è possibile specificare alcune opzioni al fine di creare il processo child in un determinato modo. Prima tra tutte è l'opzione di **condivisione di risorse**, dove si può specificare se il figlio condivide le stesse risorse del genitore, un sottoinsieme oppure si può specificare che il figlio non condivida alcuna risorsa con il *parent*. Inoltre si possono specificare le opzioni di **esecuzione**: si specifica se il figlio e il genitore possano essere eseguiti in maniera concorrente oppure se il *parent* deve aspettare il termine dell'esecuzione del *child*. Infine si può anche specificare lo **spazio degli indirizzi**, in particolare si sceglie se il figlio crea una copia identica della memoria utilizzata dal genitore oppure se carica un programma completamente nuovo.

Vediamo ora un esempio di creazione di un processo in `UNIX` (figura 6). Questo sistema operativo fornisce tre particolari *system calls*:

- ◇ `fork()`: questa system call non fa altro che creare un processo. Il processo parent, dopo aver chiamato la funzione `fork()` viene duplicato. La funzione inoltre ritorna un valore intero, se questo valore è maggiore di zero vuol dire che ci troviamo all'interno del processo genitore; se invece il valore è zero vuol dire che il codice eseguito è all'interno del child. L'unico modo per distinguere se il processo è parent o child è attraverso il valore di ritorno di `fork`.
- ◇ `exec()`: è una funzione utilizzata dal processo figlio nel caso in cui è necessario far partire un processo completamente diverso dal parent;

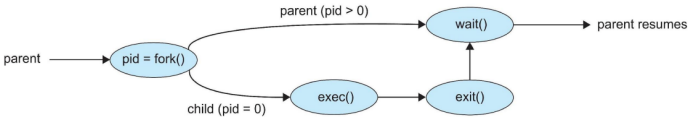


Figura 6: Rappresentazione delle 3 *system calls* fondamentali.

◇ `wait()`: è una systam call utilizzata dal genitore al fine di aspettare il termine dell'esecuzione del figlio.

1.2.3 L'albero dei processi in Linux

Come fa il sistema operativo a generare tutti i processi di cui ha bisogno? Ci deve sempre essere un processo iniziale, un programma all'inizio da cui tutti si genera. In particolare in Linux il primo processo da cui tutto è generato è chiamato `systemd` ed è il processo padre di tutti gli altri processi, quello il quale `pid` vale 1. Da `systemd`, *forkando* processo dopo processo vengono generati tutti i processi necessari all'avvio del sistema, come il terminale, generando quindi un albero (figura 7).

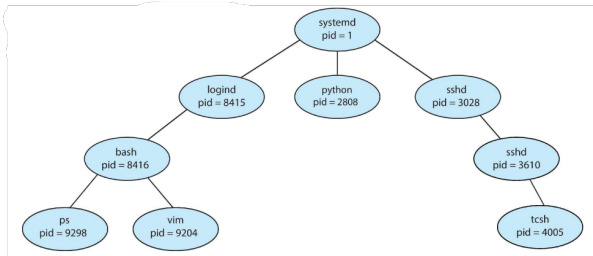


Figura 7: L'albero dei processi generato da `systemd`.

1.2.4 Terminazione di un processo

Naturalmente, un processo può anche terminare. La terminazione del processo può essere spontanea (attraverso la *system call* `exit`) e quindi il processo viene deallocato dal sistema operativo, oppure il processo può essere terminato dal genitore attraverso la system call `abort`. Questo di solito avviene quando il processo figlio supera il limite delle risorse allocate, quando la task che sta completando non è più richiesta oppure nel momento in cui il genitore termina e di conseguenza il sistema operativo termina anche i figli.

Come abbiamo visto in precedenza, esiste una funzione (`wait()`), che serve per evitare che il processo parent termini prima del processo child: la funzione infatti obbliga il parent ad aspettare che il child termini. Inoltre, se al termine di un processo child non c'è nessun processo parent che stava aspettando il termine del child, ci troviamo davanti ad un **processo zombie**. Infine, se il processo parent termina senza aspettare la terminazione del child, quest'ultimo è chiamato **orfano** e verrà terminato dal sistema operativo.

1.3 Comunicazione tra processi (IPC)

Passiamo ora a discutere i diversi modi per comunicare tra diversi processi. In alcuni casi avremo a che fare con processi indipendenti, altre volte invece necessiteremo di processi **cooperanti**. Per questi ultimi è necessario un modello di *Inter-Process Communication*, chiamata anche **IPC**. In questo paragrafo ci occuperemo di due modelli: comunicazione tramite memoria condivisa e tramite il passaggio di messaggi.

1.3.1 Memoria condivisa

Il primo modello di cui ci occuperemo è la tecnica di memoria condivisa. In questo modello, come possiamo notare anche dalla figura 8, l'unica cosa di cui si fa carico il sistema operativo è l'assegnazione di una memoria che è condivisa tra i processi A e B. Ciò significa che la comunicazione è molto veloce tra i due processi in quanto non

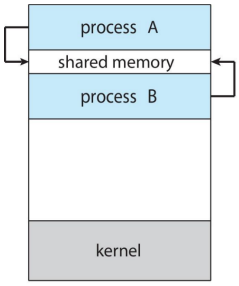


Figura 8: Il modello di memoria condivisa per IPC.

c'è nessun intermediario tra i due. Allo stesso tempo però è più facile che si generino errori come la sovrascrittura di valori e in genere problemi di sincronizzazione con i dati (come la *race condition*, vedi capitolo **??**) in quanto il sistema operativo non ha alcun tipo di controlla sulla memoria secondaria.

Partiamo ora da un esempio di questo modello al fine di ottenere informazioni più dettagliate: stiamo infatti parlando del problema `Producer - Consumer`. Ipotizziamo quindi che due processi abbiano dello spazio in memoria condiviso; la comunicazione attraverso questa memoria può avvenire in due modi:

- **unbounded**, ovvero che il produttore continua a generare dati da mettere nell'area condivisa, e che il consumatore continua ad utilizzare quei dati fino a che non finiscono (al più attende la creazione di altri dati).
- **bounded**, dove si ha un **buffer** che entrambi devono aspettare: il consumer attende che ci siano dati nel buffer e il producer aspetta nel momento in cui il buffer è pieno.

1.3.2 Passaggio di messaggi

In questo secondo caso invece il sistema operativo si prende carico di gestire la coda dei messaggi (**message queue**) che vengono scambiati tra i due processi (figura 9). In questo caso è il kernel che fa da intermediario tra i due e di conseguenza la velocità di comunicazione sarà ridotta dall'**overhead**. Allo stesso tempo però il kernel garantisce

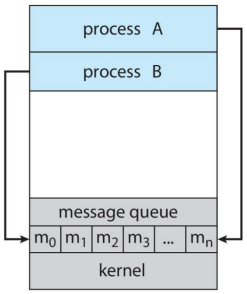


Figura 9: Il modello di memoria condivisa per IPC.

la sincronizzazione e la correttezza tra i messaggi scambiati e di conseguenza è un modello più sicuro.

Discutiamo ora più dettagliatamente questo modello. In particolare, sono fornite due operazioni fondamentali: `send(message)` e `recive(message)`. Ciò nonostante la comunicazione di tali messaggi fa sorgere diversi dubbi e domande legate alla progettazione: come sono stabilite le connessioni? qual è la sua capacità? è unidirezionale o bidirezionale? la dimensione del messaggio è fissa o variabile? un collegamento è associato solo a 2 processi o a più? Per rispondere a queste domande andiamo a vedere l'implementazione di questo modello che può essere di due tipi.

Quando si parla di comunicazione **diretta**, si intende che i processi specifichino esplicitamente il destinatario del messagio: `send(Q, message)` e `recive(P, message)`. Nella comunicazione **indiretta** invece si ha a che fare con un **buffer**, chiamato anche porta, il quale ha un ID unico tra gli altri. In questa implementazione due processi si possono parlare solo se condividono questo buffer (**mailbox**). Se più processi condividono la stessa mailbox si ha quindi un link che collega diversi processi, inoltre attraverso questa porta il collegamento è bidirezionale in quanto un processo può sia spedire un messaggio che riceverlo. Con la comunicazione indiretta le primitive però sono diverse: una volta creata la porta (chiamiamola `A`, per comodità), al fine di scambiare i messaggi è necessario utilizzare le operazioni `send(A, message)` e `recive(A, message)`.

2 Threads

Partiamo con il distinguere un thread da un processo. Il thread è un **filo di esecuzione**: in un processo ci possono essere diversi thread i quali possono avere dei diversi *pattern* per ciascuna esecuzione.

Perché?

I thread sono entità più semplici rispetto ai processi e sono quindi più facili da gestire. Grazie ai thread si ottengono diversi vantaggi:

- ◇ Il sistema è più **recettivo**;
- ◇ Dato che le risorse sono condivise è più facile gestirle;
- ◇ Richiede meno risorse rispetto alla creazione di un processo;

- ◇ Può utilizzare tutti i *core* messi a disposizione dal sistema (*multicore programming*).

Per esempio, al posto di far eseguire 4 processi differenti è molto meglio eseguire un processo con 4 thread differenti: in questo modo si evita di allocare in memoria 4 volte le stesse risorse che, grazie all'utilizzo dei thread, sono allocate solo una volta.

2.1 Concorrenza e parallelismo

Quando parliamo di **multicore programming** è necessario fare una netta distinzione tra il significato di concorrenza e parallelismo. Con **parallelismo** si intende che un sistema è in grado di preformare più di un compito in maniera simultanea (tipico dei sistemi multicore). Con **concorrenza** si intende la possibilità di far progredire più di un compito (non in maniera simultanea).

Come possiamo vedere della figura 10, quando parliamo di concorrenza ci riferiamo ad un singolo core che esegue a frammenti più thread diversi.

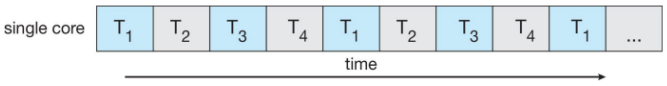


Figura 10: Esempio di concorrenza tra 4 processi.

Quando invece facciamo riferimento al parallelismo (figura 11) indichiamo la capacità del sistema di effettivamente riuscire ad eseguire parallelamente diversi thread. Si osserva che le due pratiche non sono esclusive: notiamo che il core 1 esegue

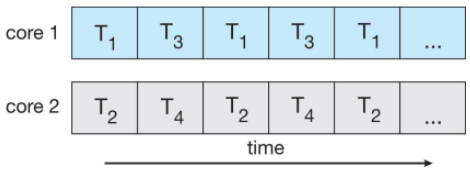


Figura 11: Esempio di parallelismo in un sistema dual core.

in maniera concorrente T1 e T3, mentre il core 2 esegue in maniera concorrente T2 e T4.

2.1.1 Tipi di parallelismo

Possiamo dividere i parallelismi in due tipi. Il primo, chiamato **data parallelism** (figura 12) implica un sottoinsieme preciso di dati sia distribuito per ogni core. In altre parole, avendo a che fare un un largo database, lo si suddivide in parti e ciascuna viene assegnata ad un core. Tale core potrà operare solo in quella porzione di dati
Il secondo tipo di parallelismo è detto **task parallelism**, rappresentato in figura 13. Questo tipo di parallelismo concede la memoria condivisa a ciascun core solo che ogni core ha un compito ben preciso: un core sarà ottimizzato per la scrittura, un altro sarà più veloce in lettura e così via.

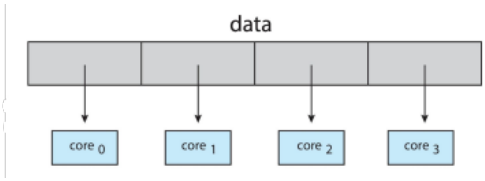


Figura 12: Esempio di parallelismo di dati.

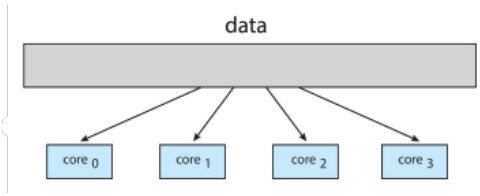


Figura 13: Esempio di parallelismo di compiti.

2.1.2 Legge di *Amdahl*

La legge di *Amdahl* è una funzione che mette in relazione due variabili importanti:

1. **S**, ovvero la percentuale di codice che non può essere parallelizzato, ovvero codice **seriale** (di conseguenza il numero di codice che può essere parallelizzato è $1 - S$);
2. **N**, che rappresenta il numero di core disponibili.

Con questi due dati abbiamo la possibili di calcolare lo ***speedup*** attraverso la seguente formula:

$$speedup \leq \frac{1}{S + \frac{1-S}{N}}$$

Osservando la formula osserviamo che se se la percentuale di codice seriale tende a zero e il numero di core tende a infinito, lo *speedup* sarebbe infinito. Questa però è una situazione utopica: non esistono casi in cui si è privi di codice seriale.

Osserviamo quindi il seguente grafico (figura 14) che mostra lo *speedup* all'aumentare del numero di core, essendo a conoscenza della percentuale di codice seriale. Notiamo che quando la percentuale di codice seriale è circa la metà non ha importanza il numero di core del sistema: lo speedup rimarrà pressoché invariato. Anche solo con il 5% di codice seriale notiamo un forte abbassamento rispetto allo speedup ideale. È evidente quindi che l'aumento di core non causa l'aumento di speedup, sopratutto in una situazione dove il codice seriale è molto.

2.2 Modelli multithreading

È importante fare una distinzione tra due classi principali di thread: **user threads** e **kernel threads**. La principale differenza tra i due è che i kernell threads hanno molti più privilegi rispetto a quelli utente. Esistono quindi dei modelli che cercano di associare agli user threads i kernell threads al fine di sfruttare al meglio il principio di *multithreading*.

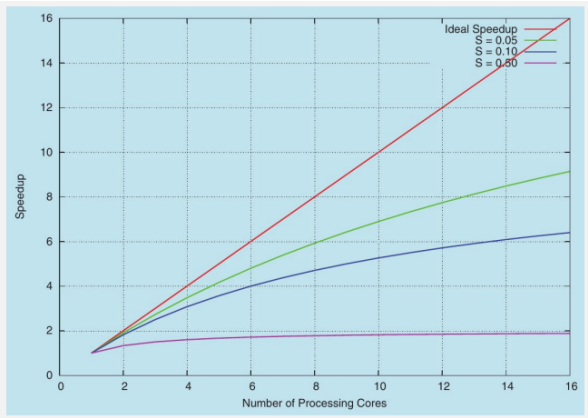


Figura 14: Il grafico che descrive lo *speedup* a seconda della percentuale di codice seriale.

2.2.1 Many-to-One

In questa prima architettura, il kernell mette a disposizione solo un thread che è collegato e deve soddisfare tutte le richieste di tutti gli user threads (figura 15). È

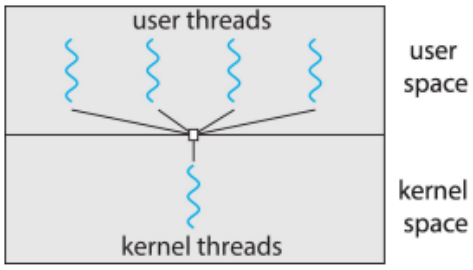


Figura 15: Il modello di *multithreading* molti a uno.

evidente che l'efficienza di questo modello non è il suo forte: sono presenti diversi thread utente ai quali deve rispondere solamente un thread del kernel. Basta pesare al fatto che se il kernel thread è in attesa di un input esterno, tutti gli user thread sono bloccati (**collo di bottiglia**). Questo modello è infatti poco utilizzato dato che non sfrutta le potenzialità del *multicore*.

2.2.2 One-to-One

In questa architettura (figura 16), quando viene creato uno user thread, il suo rispettivo kernel thread viene creato; così facendo esiste un kernell thread associato ad ogni user thread. Ad ogni modo ci possono essere alcune restrizioni in modo da evitare la creazioni di troppi kernel threads (e quindi limitare la creazione di user threads). È comunque evidente che questo modello è sicuramente più **efficace** del modello precedente in quanto fornisce la possibilità di sfruttare a piano un sistema multicore.

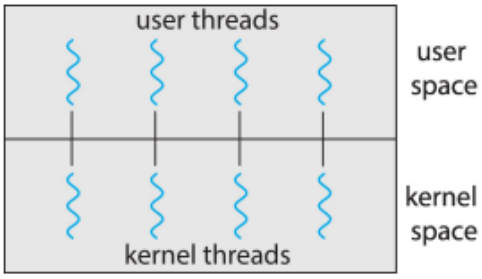


Figura 16: Il modello di *multithreading* uno a uno.

2.2.3 Many-to-Many

L'ultimo modello è un buon compromesso tra il modello *Many-to-One* e il modello *One-to-One*. Il modello molti a molti (figura 17) fornisce diversi vantaggi ed è più flessibile rispetto ai primi due. Non esiste infatti la corrispondenza univoca, generalmente si hanno più user threads che fanno riferimento ad alcuni kernell threads (è come se una sala da 20 clienti fosse gestita da 5 camerieri). È possibile infatti

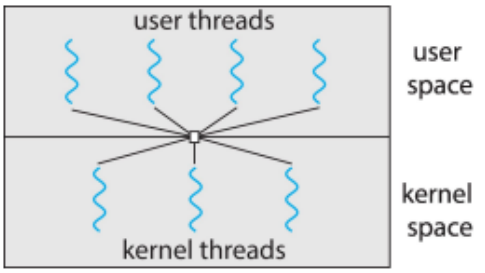


Figura 17: Il modello di *multithreading* molti a molti.

controllare in maniera più efficace i kernell thread e questo rende il modello *Many-to-Many* molto **robusto**. Ad ogni modo la sua implementazione è molto più complessa rispetto ai modelli precedenti.

Per questo, a volte, si fa riferimento ad un *ibrido* tra il modello *Many-to-Many* e il modello *One-to-One*: stiamo parlando del **Two-level model** che consente sia la corrispondenza *1 a 1* che la corrispondenza *N a M*.

2.3 Librerie di thread

In questa piccola sezione ci interessiamo alle libreire disponibili per la creazione e la gestione di threads. Queste possono essere di due tipi:

- ◇ Librerie in **user space**, dove i thread sono gestiti completamente a livello utente (come nel caso di `Pthreads`);
- ◇ Librerie di tipo **kernel-level** che sono supportate dal sistema operativo e si appoggiano al kernel attraverso le *system calls*; questo comporta un livello maggiore di complessità nel kernell ma il programmatore ha meno da implementare.

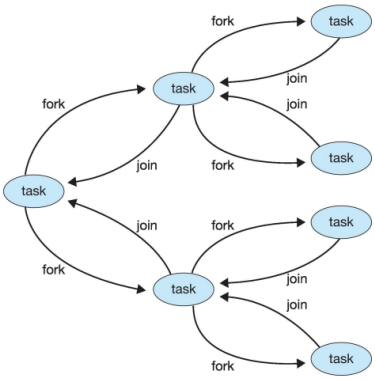


Figura 18: Rappresentazione grafica del modello fork-join per la risoluzione di un task.

Spendiamo due parole su `POSIX Threads`. Questa, non è propriamente una libreria ma è un insieme di specifiche (non implementazioni!) che aiuta con la creazione e la gestione di thread. `POSIX Threads` fornisce specifiche sia a livello di utente che a livello di kernel.

2.4 Threading implicito

Cerchiamo ora di cambiare approccio: attraverso le librerie l'approccio era esplicito, stava al programmatore l'implementazione e/o la gestione dei thread. Nel momento in cui si fa riferimento a più threads, diventa sempre più difficile controllare la correttezza del codice. Ecco che si è pensato ad un altro approccio: il threading **implicito**. Con questo approccio i thread sono gestiti maggiormente dal compilatore oppure da librerie *run-time* le quali si occupano di creare e gestire i thread. In questa sezione vedremo alcuni dei metodi utilizzati per ottenere il threading implicito.

2.4.1 Modello fork-join

Questo tipo di modello si ispira alla creazione di processi (paragrafo 1.2.2). In questo modello, diversi threads sono divisi e, una volta che sono terminati, vengono uniti (figura 18). La divisione dei threads è una scelta presa completamente dalla libreria. Come possiamo notare dalla figura 18, in questo caso, la libreria sceglie un task da far risolvere ad un thread dove, eventualmente verrà splittata in altri due thread (che vengono appositamente creati) e così via fino a che il task non sia completamente risolto.

2.4.2 Thread pools e OpenMP

Altri due modelli molto importanti sono *thread pools* e `OpenMP`. Nel primo caso la libreria mette a disposizione un numero di thread (*pool* di thread) che attendono un task da risolvere. In questo modo, dato che i thread sono già pronti, non si spreca tempo per l'effettivo processo di creazione. Inoltre il problema della limitazione dei

thread è risolto in quanto una volta creati quelli per la *pool*, non vengono creati altri thread.

Quando parliamo di `OpenMP` invece parliamo di una serie di **direttive** per il compilatore e un insieme di librerie per `C`, `C++` e `FORTRAN`. Questo modello fornisce tutte le risorse per la condivisione della memoria tra i thread e i parallelismi. Essendo delle direttive, queste devo essere proprio scritte nel codice; per esempio: `#pragm omp parallel`.

2.5 Problematiche

Come vedremo in questo paragrafo, anche i thread portano a delle problematiche che devono essere gestite come, per esempio, l'interruzione di tali e il modo di implementare l'eliminazione di un thread.

2.5.1 Semantica `exec` e `fork`

Il primo dubbio che è necessario chiarire è il comportamento durante la chiamata della funzione `exec()`: se si fa una chiamata alla funzione `exec()`, vengono rimpiazzati tutti i thread del processo oppure solo il thread su cui è stata chiamata `exec`? Generalmente la funzione `exec` cancella il processo in esecuzione e, di conseguenza, tutti i sui thread.

Il secondo dubbio, riguarda la funzione `fork()`: se si fa una chiamata a `fork()` ad un processo multithread, si effettua una copia a tutti i thread del processo oppure solo al thread su cui è stata chiama la funzione? La risposta a questa domanda è che dipende dall'obiettivo della funzione `fork()`:

- ◇ Se la funzione deve cambiare subito il codice attraverso `exec()`, non vale la piena copiare tutti i thread se tanto si si che verranno eliminati.
- ◇ Se invece il nuovo processo deve supportare anch'esso il multithreading, allora ha senso effettuare una coppia di tutti i thread e non solo del thread su cui è stata chiamata la funzione

2.5.2 Segnalazione ed eliminazione

I segnali sono usati per notificare un processo che un determinato evento è accaduto. Tali segnali però debbono essere gestiti: ecco che emerge la figura del **signal handler** che può essere di **default** oppure **user-defined**, ovvero definito dall'utente. Generalmente ogni segnale ha il suo specifico *default handler* che è utilizzato anche dal kernel. Cosa succede però nel caso in cui il sistema è multi-threading? Ci sono diverse opzioni:

1. Spedire il segnale al thread ad esso compatibile;
2. Propagare il segnale ad ogni thread del processo;
3. Mandare il segnale ad dei thread specifici del processo;
4. Assegnare ad uno specifico thread il compito di ricevere i segnali degli altri thread.

È inoltre importante riuscire a gestire la **cancellazione** dei thread. Questi però devono attivare la possibilità di essere cancellati: in altre parole, se un thread ha la cancellazione disabilitata non potrà essere cancellato fino a che non viene riattivata. Nel momento in cui la cancellazione è attivata, il thread può essere cancellato attraverso due tecniche:

- ◇ **Asincrona**, ovvero il thread termina immediatamente;
- ◇ **Deferred** che significa *posticipata*. Può ritornare utile nel momento in cui il thread sta compiendo un'operazione delicata (chiusura di un file) e non ha la possibilità di terminare immediatamente.

3 CPU Scheduling

In questa sezione ci occupiamo di tutti gli aspetti che concernono lo scheduling del processore, ovvero la pratica secondo la quale, attraverso dei precisi criteri, si sceglie quale processo all'interno della *ready queue* verrà eseguito.

3.1 Nozioni fondamentali

Prima di iniziare a discutere di scheduling vero e proprio è necessario avere chiari alcuni concetti importanti.

Burst. La prima è la nozione di burst. Chiamiamo **CPU burst** il periodo di tempo nel quale un processo esegue operazioni all'interno della CPU; diversamente, l'**I/O burst** è il tempo che il processo spende interfacciandosi con le periferiche di input e output. In particolare, un processo che occupa per molto tempo la CPU si dice *CPU bounded*, mentre nel caso di I/O si parla di un processo *I/O bounded*. In questo capito ci occupiamo solo di CPU burst.

CPU scheduler. Il CPU scheduler è il responsabile dell'organizzazione e dell'ordinamento della **coda dei processi** (figura 19). Le decisioni su quale processo deve essere eseguito prima stanno a lui. Questo tipo di decisioni avvengono generalmente quando un processo:

1. Passa dallo stato *running* a *waiting*;
2. Passa da *running* allo stato *ready*;
3. Passa dallo stato *waiting* allo stato *ready*;
4. Termina.

Osserviamo che per i punti 1 e 4 non è presente una vera e propria decisione: un nuovo processo deve essere messo in esecuzione. Questo non vale per i punti 2 e 3 dove si fa una decisione.

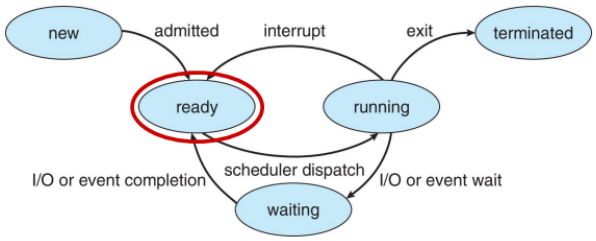


Figura 19: Lo scheduler entra in gioco nel passaggio da ready a running.

Preemption. Ci sono due macro gruppi di scheduling: preemptive e non. Uno scheduling è detto **non preemptive** nel momento in cui un processo non può essere fermato. In altre parole, quando la CPU è assegnata ad un processo, questo la utilizza fino a che non è terminato. Differentemente, uno scheduling è detto **preemptive** quando un processo può essere fermato per dare precedenza ad un altro per poi essere

fatto ripartire: questo tipo di scheduling è sicuramente più performante e moderno; è infatti utilizzato nei sistemi operativi più diffusi come Windows, MacOS, Linux e altri. Ciò porta comunque a delle situazioni indesiderate come le **race conditions**: poniamo il caso di avere due processi che condividono dei dati; immaginiamo che il primo processo stia aggiornando questi dati ma allo stesso tempo il secondo processo li stia utilizzando. Questo è un problema dato che il processo due sta utilizzando dei dati che non sono consistenti dato che sono in fase di aggiornamento dal processo uno. Come vedremo nel capitolo **??**, questo è un problema di sincronizzazione che va risolto.

Dispatcher. Nel momento in cui lo scheduler ha scelto quale processo verrà eseguito, il dispatcher si occupa di cambiare il processo nella CPU. In particolare viene effettuato il *context switch* (1.2.1), passa in *user mode* e va alla giusta locazione del programma per iniziare la sua esecuzione. Durante tutto ciò la CPU però non lavora: è importante quindi minimizzare questa latenza e fare in modo che non se ne effettuino un numero troppo elevato al fine di mantenere un alta percentuale di utilizzo della CPU.

3.2 Algoritmi non preemptive

In questo paragrafo ci occupiamo dei primi algoritmi non preemptive, ovvero gli algoritmi che non fermano i processi che sono in esecuzione.

3.2.1 First-Come First-Served (FCFS)

Questo è l'algoritmo più banale da implementare: il primo processo che entra nella coda sarà anche il primo ad essere eseguito. Questo algoritmo ha un approccio praticamente identico al **FIFO** (*First In First Out*). Attraverso un semplice esempio, possiamo osservare che questo algoritmo non è efficiente. Poniamo che entrino nella coda 3 processi: P_1, di durata 24 unità di tempo (in genere millisecondi) e P_2 e P_3 con durata di esecuzione 3. Osservando la figura 20 è banale notare che se P_1 fosse

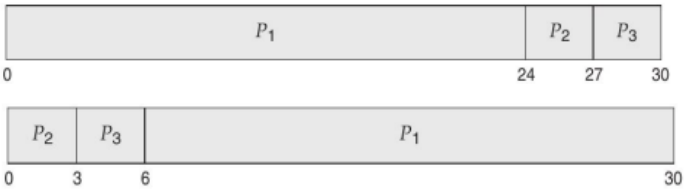


Figura 20: Diagramma di *Gantt* dell'algoritmo FCFS.

arrivato in coda per ultimo, i processi P_2 e P_3 avrebbero aspettato meno. Possiamo dimostrarlo anche in maniera più matematica, attraverso dei brevi calcoli. Nel primo caso P_1 ha aspettato 0 prima di essere eseguito, P_2 ha aspettato l'esecuzione di $P_1 = 24$ mentre P_3 ha aspettato l'esecuzione di $P_1 + P_2 = 24 + 3 = 27$. Se provassimo a fare una media del tempo di attesa otteniamo:

$$\langle T \rangle = \frac{0 + 24 + 27}{3} = 17$$

Nel secondo caso invece la situazione migliora notevolmente in quanto P_2 aspetta 0, P_3 aspetta solamente l'esecuzione di $P_2 = 3$ e infine P_1 attende l'esecuzione di $P_2 + P_3 = 3 + 3 = 6$. Attraverso la stessa espressione matematica otteniamo che l'attesa media in questo caso diventa:

$$\langle T \rangle = \frac{6 + 0 + 3}{3} = 3$$

Diversi sono i problemi di questo algoritmo. Prima di tutto può generare il **convoy effect** (effetto convoglio): se viene eseguito un processo *I/O bounded*, tutti gli altri processi in coda devono aspettare che questo si "sblocchi" generando quindi un rallentamento generale. In secondo luogo, questo algoritmo di scheduling dipende dall'ordine di entrata dei processi e di conseguenza **non** è nemmeno possibili analizzare in modo **deterministico** le prestazioni dell'algoritmo.

3.2.2 Shortest-Job-First (SJF)

Come abbiamo notato dalla figura 20, se i processi più bervi sono eseguiti prima, il tempo medio di attesa $\langle T \rangle$ si abbassa. Implementiamo quindi un algoritmo che dia la precedenza ai processi con il tempo di esecuzione più breve tra quelli che sono presenti in coda. In questo algoritmo stiamo assumendo che siamo a conoscenza del CPU burst time di ciascun processo: si osserva che stiamo facendo un'ipotesi, spesso questo dato non è a disposizione. Se tutti i CPU burst sono conosciuti, l'algoritmo fornisce il minor tempo medio di attesa di un insieme finito di processi.

Il funzionamento di questo algoritmo è molto semplice: in base ai processi arrivati in coda, questi, prima di essere eseguiti, vengono riordinati in base al loro tempo di burst. Per esempio, poniamo di avere in coda un processo P_1 con un burst time di 6, P_2 da 8, P_3 da 7 e P_4 da 3. Osservando la figura 21 notiamo che i processi

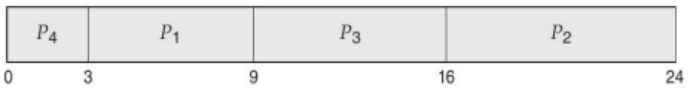


Figura 21: Diagramma di *Gantt* dell'algoritmo SJF.

sono stati riordinati in modo tale da minimizzare il tempo medio d'attesa. In questo esempio il processo P_4 attende 0, il processo P_1 attende l'esecuzione di $P_4 = 3$, il processo P_3 aspetta la conclusione di $P_4 + P_1 = 3 + 6 = 9$ e infine il processo P_2 aspetta $P_4 + P_1 + P_3 = 3 + 6 + 7 = 16$. Ecco che il tempo di attesa medio diventa:

$$\langle T \rangle = \frac{3 + 16 + 9 + 0}{4} = 7$$

Anche in questo caso però se durante l'esecuzione è in coda un processo con un burst molto alto e entrano solo processi con un burst basso, è probabile che si verifichi una situazione di attesa perenne (chiamata **starvation**); vedremo, nel corso di questo capitolo, come ciò può essere evitato (3.4).

3.2.3 Stima del *CPU burst time*

Come abbiamo affermato poco fa, quasi mai il burst time è a disposizione. Si è quindi trovato un modo per stimare al meglio il burst time di un processo, in base ai processi

che sono stati eseguiti in precedenza. Il metodo utilizzato è chiamato **exponential averaging** il quale, in essenza, dà peso maggiore ai processi eseguiti da poco tempo e, pian piano, più i processi sono remoti, meno influenza hanno sulla stima. Le variabili in gioco nella formula sono:

- t_i indica la durata del CPU burst del processo i-esimo, dove $i \in [0, n]$, dove n indica il numero di processi;
- τ_{n+1} rappresenta la stima, la predizione (*guess*), che si calcola sul processo che si sta per eseguire;
- α che è un coefficiente che indica quanto pesa la storia dei processi. Quando questo coefficiente è basso, la storia recente non conta, mentre quando è alto, la storia recente ha più peso (con $\alpha = 1$ si ha che solo l'ultimo processo influisce sulla stima). Generalmente si utilizza il valore $\alpha = \frac{1}{2}$.

Nel caso generale, con n processi si ha che per stimare il burts dell' $n + 1$-esimo:

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

Osserviamo ora la figura 22 che rappresenta come viene effettuata la *guess* in base alla storia dei processi ($\alpha = .5$). La prima colonna composta dalla coppia $\binom{6}{10}$ è la

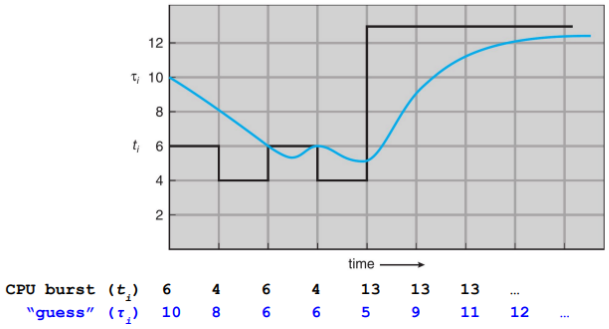


Figura 22: Grafico che indica come viene stimato il burst di un processo.

colonna "base", la partenza del nostro grafico. Con questi due dati, si calcola la seconda colonna $\binom{4}{8}$, dove $8 = \frac{10+6}{2}$; a questo punto si può calcolare la terza colonna $\binom{6}{6}$, dove il secondo $6 = \frac{4+8}{2}$. Così facendo si è in grado di calcolare una buona stima per il CPU burst time del processo corrente.

3.3 Algoritmi preemptive

In questo secondo paragrafo discutiamo invece di due algoritmi preemptive, ovvero algoritmi che possono fermare l'esecuzione di un processo per favorirne un altro.

3.3.1 Shortest-Remaining-Time-First (SRTF)

Il primo algoritmo che andremo a discutere è la versione preemptive del SJF: in questo caso viene servito per primo il processo al quale manca minor tempo per essere

completato. Quindi se si sta eseguendo un processo P e arriva un processo Q il quale burst time è minore rispetto al burst time che manca a P per terminare, quest'ultimo viene fermato per dare la precedenza a Q. Una volta terminata l'esecuzione di Q, il processo P riprede da dove era stato fermato in precedenza. Prendiamo in considerazione i seguenti 4 processi:

Processo	Tempo di arrivo	Stima burst time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Osservando la figura 23, cerchiamo di capire come si comporta questo algoritmo nel momento in cui i 4 processi sono inseriti all'interno della coda. Al tempo zero

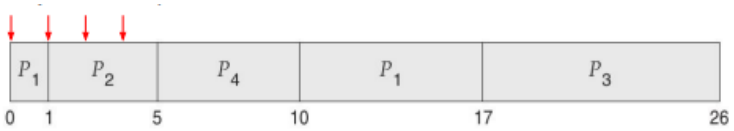


Figura 23: Diagramma di *Gantt* dell'algoritmo SRTF.

arriva in coda il processo P_1, di durata 8. Al tempo 1 arriva in coda il processo P_2 che ha una durata di 4. A P_1 rimangono ancora $8 - 1 = 7$ unità di tempo prima di terminare, mentre a P_2 ne servono solo 4. P_1 viene quindi fermato e P_2 inizia la sua esecuzione (si effettua un *depatching*). Al tempo 2 e al tempo 3 sono aggiunti alla coda P_3 e P_4 i quali però hanno un burst time maggiore rispetto a P_2 che quindi termina l'esecuzione al tempo $1+4 = 5$. A questo punto rimangono P_1, P_3 e P_4. Viene eseguito P_4 in quanto il suo burst (5) è minore rispetto a quello di P_1 (7) e P_3 (9). Terminata l'esecuzione di P_4 inizia quella di P_1 e poi quella di P_3.

Osserviamo che con questo algoritmo può capitare che sia in esecuzione un processo con un tempo di burst molto elevato e che poi continuino ad arrivare dei processi con un tempo di burst ridotto. In questo caso, il processo con il tempo maggiore verrebbe sempre interrotto dagli altri processi e non riuscirebbe mai a terminare andando quindi in una situazione di **starvation**. Come vedremo, una soluzione è questo problema è fornita dallo scheduling con priorità (3.4).

3.3.2 Round Robin (RR)

Passiamo ora ad un algoritmo un po' più sofisticato ed elegante: stiamo parlando del Round Robin. Alla base di questo algoritmo c'è il **quanto** di tempo (generalmente tra i 10 e i 100 millisecondi): ogni processo all'interno della coda, ha diritto ad essere eseguito per 1 quanto di tempo alla volta. Di conseguenza, ogni quanto di tempo viene effettuato un *context switch* e si prosegue ad un altro processo nella coda: si continua così in maniera ciclica finché ciascun processo viene eseguito completamente lasciando spazio ai nuovi.

Osserviamo che se $q \to \infty$ si ha un comportamento FIFO, molto simile all'algoritmo First-Come First-Served. Allo stesso tempo però q deve comunque essere maggiore del tempo che ci si impiega per effettuare un context switch (ordine

dei μs), altrimenti la CPU viene sprecata solo per fare i context switch al posto di effettivamente eseguire i processi.

Consideriamo il caso in cui $q = 4$ e in coda sono presenti i tre processi utilizzati nell'algoritmo FCFS:

Processo	Stima burst time
P_1	24
P_2	3
P_3	3

Il comportamento del Round Robin, in questo caso, è illustrato nella figura 24. Os-

Figura 24: Diagramma di *Gantt* dell'algoritmo RR.

serviamo che il RR, per il processo P_1, ogni $q = 4$, si ferma per fare spazio agli altri processi mentre, per i processi P_2 e P_3, che durano meno di un quanto, quando terminano il Round Robin, non aspetta la scadenza del quanto per eseguire un altro processo ma comincia subito, che è un comportamento ragionevolmente ovvio.

3.3.3 Reattività

Cerchiamo ora di capire il vantaggio che forniscono gli algoritmi di scheduling preempitive (in particolare il RR) rispetto agli algoritmi non preemptive. Prendiamo come esempio i seguenti processi:

Processo	Burst time
P_1	6
P_2	3
P_3	1
P_4	7

Poniamo ora che il quanto di tempo q sia 4. Calcoliamo ora il tournaround time medio tra questi processi: P_1 viene eseguti per 4 unità, dopo di chè viene fermato (gliene rimangono 2) e viene eseguito P_2 che terima ($T_{P_2} = 4 + 3 = 7$); viene quindi eseguito P_3 che termina anche lui ($T_{P_3} = 7 + 1 = 8$). A questo punto inizia l'esecuzione di P_4 che viene fermato dopo 4 unità (ne rimangono ancora 3) e viene fatto ripartire P_1 che termina ($T_{P_1} = 8 + 4 + 2 = 14$) e infine viene fatto terminare anche P_4 ($T_{P_4} = 14 + 3 = 17$). Per trovare il turnaround time medio si effettua la media dei 4 tournaround trovati.

$$\langle T \rangle = \frac{T_{P_1} + T_{P_2} + T_{P_3} + T_{P_4}}{4} = \frac{14 + 7 + 8 + 17}{4} = \frac{46}{4} = 11.5$$

Osserviamo però che se avessimo utilizzato l'algoritmo SJF (vedi 3.2.2) il tournaround time medio è 8 che è minore rispetto a quello fornito da RR. Ciò significa che utilzzare un algoritmo pre-emptive, in termini di tempistiche, non è necessariamente

la scelta migliore, è semplicemente un altro modo per schedulare l'esecuzione dei processi, ma non ne garantisce il miglioramento della prestazione. Allora perchè utilizzare questi algoritmo? Perchè migliora la reattività (**responsiveness**) del sistema. Ipotizziamo di avere un coda moltissimi processi che stanno in esecuzione. Un algoritmo non preempitve li esegue uno ad uno (secondo determinati criteri, più o meno efficienti) ma tutti i processi devono sempre stare in attesa che uno termini. Il RR invece garantisce che tutti i processi vengano eseguiti per almeno un certo quanto q di tempo: è quindi un algoritmo più **equo** rispetto agli algoritmi non preemptive.

3.4 Scheduling con priorità

Fino ad ora abbiamo trattato i processi in modo equo se non per la stima del *burst time*. Introduciamo ora una seconda informazione, la **priorità** che non è altro che un numero che indica quanto sia importante (urgente) l'esecuzione di un determinato algoritmo. La priorità può essere sia legata al CPU burst time ma può anche essere legata ad altri aspetti.

Arriva però un problema: la **starvation**. Anche in questo caso, come nel SRTF, può capitare che i processi che hanno una priorità di grado molto basso non vengano mai eseguiti in quanto sono sempre presenti processi con una priorità più alta. Con l'introduzione della priorità però, se un processo è da troppo tempo in coda, si aumenta di un grado la priorità al fine da mandarlo in esecuzione. Questa tecnica rappresenta allegoricamente l'invecchiamento (**aging**) del processo nella coda di attesa.

Vediamo ora un primo esempio di scheduling con priorità puro. Abbiamo a che fare con i 5 processi seguenti:

Processo	Stima burst time	Priorità
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Osserviamo che noi consideriamo come numero più basso la priorità più alta (di conseguenza P_2 è il processo con priorità più alta e P_4 quello con priorità più bassa). Detto ciò procediamo con il diagramma di Gantt (figura 25). Notiamo infatti

Figura 25: Diagramma di *Gantt* dell'algoritmo basato puramente sulla priorità.

che i processi sono eseguiti in ordine in base alla loro priorità: P_2, P_5, P_1, P_3 e P_4. Ovviamente, se durante l'esecuzione di P_1 (che ha priorità 3) fosse arrivato in coda un algoritmo con priorità 2, P_1 sarebbe stato interrotto per favorire l'esecuzione del nuovo processo. Inoltre, nel caso in cui due processi abbiano la stessa priorità si segue la dinamica FIFO, ovvero il primo processo che entra nella coda viene eseguito per primo rispetto ai processi con medesima priorità

3.4.1 Scheduling con pririt̀a e RR

Cerchiamo ora di raffinare un po' di più l'algoritmo fondendo la priorità con l'algoritmo di Round Robin. In particolare, nel momento in cui si hanno più processi che hanno lo stesso livello di priorità, al posto di seguire un approccio FIFO, si utilizza il Round Robin, garantendo quindi una maggiore reattività ai processi. Partiamo dal seguente set di processi:

Processo	Stima burst time	Priorità
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Come è possibile osservare dalla figura 26, notiamo che il processo 4, essendo che è l'unico processo con priorità 1, viene eseguito per primo e non viene interrotto da nessun'altro processo. Dopo di che si effettua l'algoritmo RR con $q = 2$ sui processi 2

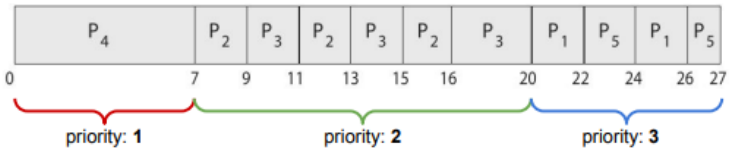
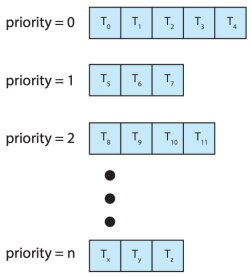


Figura 26: Diagramma di *Gantt* dello scheduling con priorità unito all'algoritmo RR per i processi con lo stesso grado di urgenza.

e 3 in quanto hanno la stessa priorità. Infine, si fa lo stesso procedimento che con i processi 1 e 5 che hanno priortà 3.

3.4.2 Coda multilivello



Il fatto che per ogni grado di priorità venga eseguito il Round Robin ci porta a creare uno scheduling multilivello dove ogni livello corrisponde ad un grado di priorità. Di conseguenza nell'esecuzione viene prima eseguito il primo livello attraverso il RR; dopo di che si passa al secondo livello e così via fino all'ultimo grado di priorità. In particolare, le priorità più alte sono assegnati a processi che hanno un bisogno **real-time** (come per il controllo di un braccio robotico) che poi sono seguiti dai processi di sistema etc.

Questo ci porta al caso più complesso: immaginiamo che i processi non solo debbano essere inseriti nella coda giusta ma che questi debbano anche essere in grado di sposarsi da una coda all'altra e quindi cambiando il loro grado di priorità. Stiamo infatti parlando delle **Multilevel Feedback Queue** (MFQ).

3.5 Scheduling multiprocessore

Fino ad ora abbiamo discusso di algoritmi tenendo conto del fatto che il sistema disponesse di un singolo processore; sappiamo bene però che nei sistemi moderni ormai una situazione del genere non si verifica più, con sistemi multicore.

Come possono essere gestiti diversi threads all'interno di queste architetture multiprocessore? Nel caso del **Syimmetric multiprocessing** (SMP)

Finisci introduzione CPU scheduling 2

3.5.1 Symmetric multiprocessing (SMP)

Partiamo dalla situazione più semplice, nel casi in cui abbiamo a che fare con un'architettura SMP. In questo caso infatti abbiamo *cores* che vengono gestiti in modo simmetrico (vedi paragrafo 2.1) e, disponendo di n corse, l'unico problema da risolvere è come questi possano andare a gestire n thread. Come è infatti possibile notare dalla figura 27, un modo può essere quello di utilizzare una *ready-queue* comune a tutti i cores oppure quello di creare una coda apposita ad ogni core per gestire i processi. Entrambe le soluzioni sono più che lecite anche se ad oggi i sistemi operativi

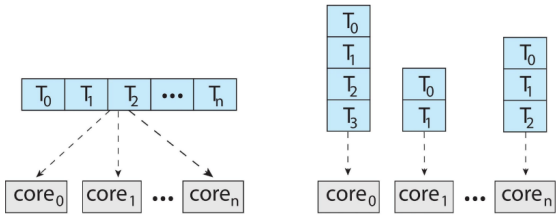


Figura 27: Ci sono diverse modi per gestire n threads con n cores.

moderni tendono ad usare la seconda in quanto la prima soluzione, essendo si che si ha a che fare con una coda condivisa è complicato gestire la condivisione di tale risorsa tra i cores (problemi di *race conditions*).

3.5.2

3.6 Scheduling real-time

3.7 Valutazione di un algoritmo

Finisci CPU scheduling: parte 2