

Sistemi Operativi

Alessandro Trigolo

29 febbraio 2024

Indice

I	Gestione dei Processi	3
1	Processi	3
1.1	Allocazione in memoria	3
1.1.1	Process Control Block (PCB)	4
1.2	Stati di un processo	5
1.2.1	Context switch	5
1.2.2	Creazione di un processo	6
1.2.3	L'albero dei processi in Linux	7
1.2.4	Terminazione di un processo	7
1.3	Comunicazione tra processi (IPC)	8
1.3.1	Memoria condivisa	8
1.3.2	Passaggio di messaggi	8
2	Threads	9
2.1	Concorrenza e parallelismo	10
2.1.1	Tipi di parallelismo	10
2.1.2	Legge di <i>Amdahl</i>	11
2.2	Modelli multithreading	11
2.2.1	Many-to-One	12
2.2.2	One-to-One	12
2.2.3	Many-to-Many	13
2.3	Librerie di thread	13
2.4	Threading implicito	14
2.4.1	Modello fork-join	14
2.4.2	Thread pools e OpenMP	14
2.5	Problematiche	15
2.5.1	Semantica <i>exec</i> e <i>fork</i>	15
2.5.2	Segnalazione ed eliminazione	15

Elenco delle figure

1	Spazio in memoria allocato per il processo dal sistema operativo. . . .	3
2	Rappresentazione del contenuto di un generico PCB.	4
3	Lista concatenata che mantiene tutti i PCB dei processi (task) in Linux.	4
4	Gli stati della vita di un processo.	5
5	Il context switch.	6
6	Rappresentazione delle 3 <i>system calls</i> fondamentali.	7
7	L'albero dei processi generato da <code>systemd</code>	7
8	Il modello di memoria condivisa per IPC.	8
9	Il modello di memoria condivisa per IPC.	9
10	Esempio di concorrenza tra 4 processi.	10
11	Esempio di parallelismo in un sistema dual core.	10
12	Esempio di parallelismo di dati.	11
13	Esempio di parallelismo di compiti.	11
14	Il grafico che descrive lo <i>speedup</i> a seconda della percentuale di codice seriale.	12
15	Il modello di <i>multithreading</i> molti a uno.	12
16	Il modello di <i>multithreading</i> uno a uno.	13
17	Il modello di <i>multithreading</i> molti a molti.	13
18	Rappresentazione grafica del modello fork-join per la risoluzione di un task.	14

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

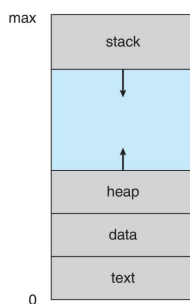


Figura 1: Spazio in memoria allocato per il processo dal sistema operativo.

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 programma¹.

¹Come 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 ??), come la priorità del processo;
5. Informazioni per la gestione della memoria e dell'I/O.

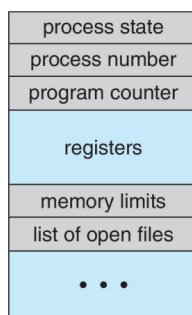


Figura 2: Rappresentazione del contenuto di un generico PCB.

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

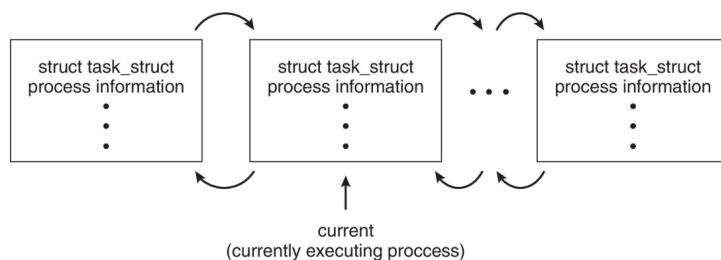


Figura 3: Lista concatenata che mantiene tutti i PCB dei processi (task) in Linux.

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 ?? 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 ritorna nello stato ready;
5. **Terminated**: una volta che il processo finisce la sua esecuzione, questo ovviamente termina e viene rimosso dalla CPU.

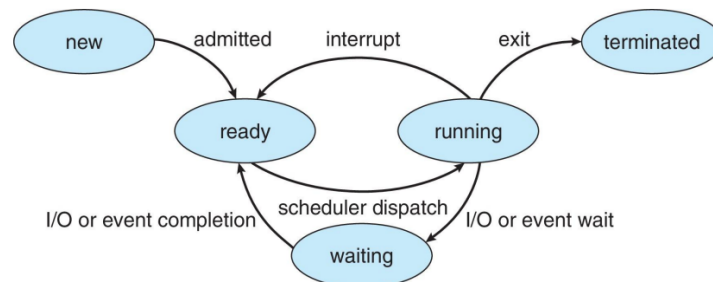


Figura 4: Gli stati della vita di un processo.

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 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 ??). È importante notare che il context switch tipicamente richiede anche un aggiuntivo utilizzo della

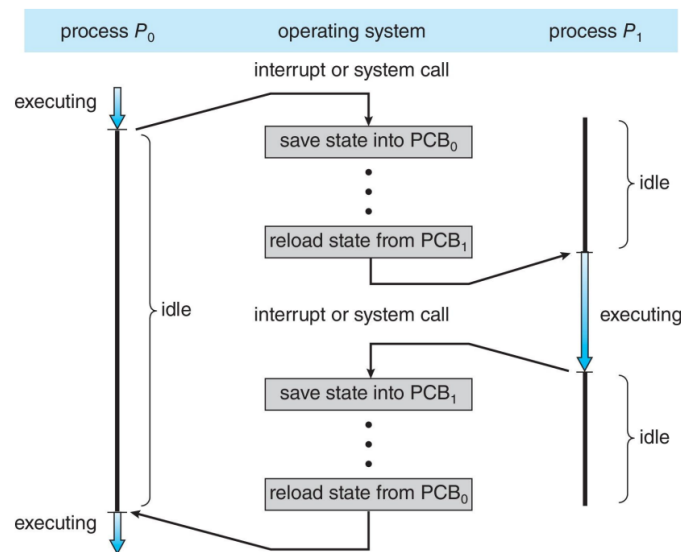


Figura 5: Il context switch.

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;

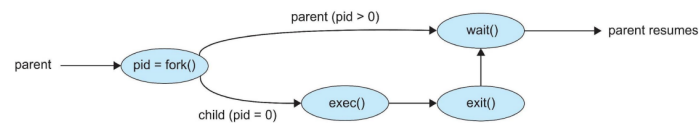


Figura 6: Rappresentazione delle 3 *system calls* fondamentali.

- ◇ `wait()`: è una *system 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).

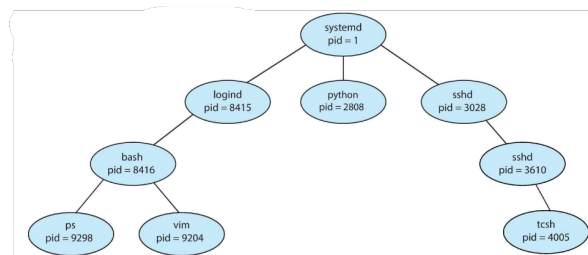


Figura 7: L'albero dei processi generato da `systemd`.

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

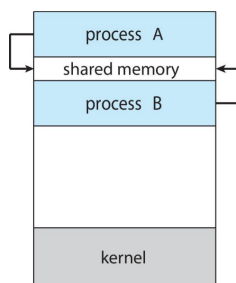


Figura 8: Il modello di memoria condivisa per IPC.

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 controllo 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

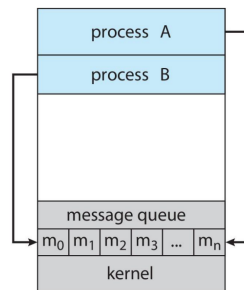


Figura 9: Il modello di memoria condivisa per IPC.

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 `recv(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 specificano esplicitamente il destinatario del messaggio: `send(Q, message)` e `recv(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 `recv(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

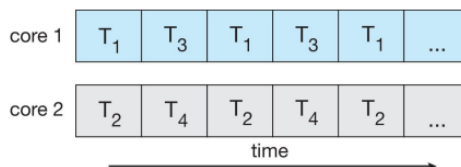


Figura 11: Esempio di parallelismo in un sistema dual core.

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.

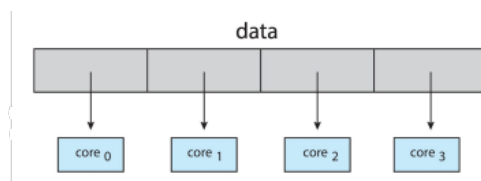


Figura 12: Esempio di parallelismo di dati.

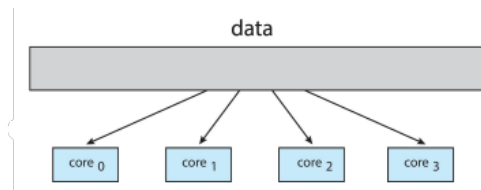


Figura 13: Esempio di parallelismo di compiti.

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 possibilità di calcolare lo *speedup* attraverso la seguente formula:

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

Osservando la formula osserviamo che 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*, soprattutto 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 kernel threads hanno molti più privilegi rispetto a quelli utente. Esistono quindi dei modelli che cercano di associare agli user threads i kernel threads al fine di sfruttare al meglio il principio di *multithreading*.

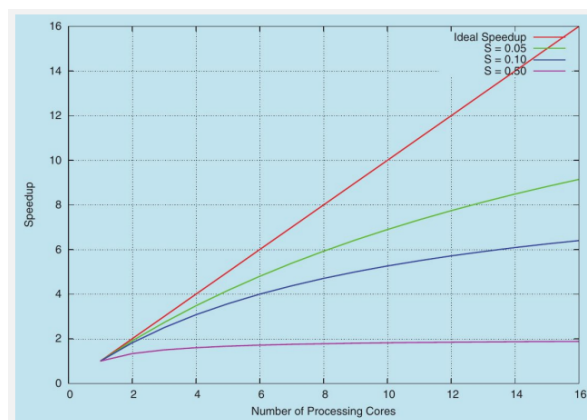


Figura 14: Il grafico che descrive lo *speedup* a seconda della percentuale di codice seriale.

2.2.1 Many-to-One

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

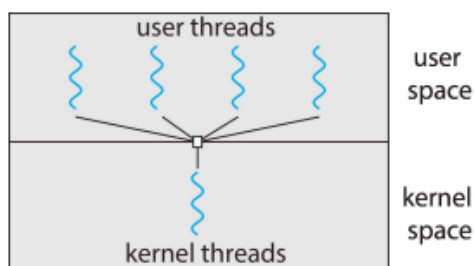


Figura 15: Il modello di *multithreading* molti a uno.

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 pensare 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 kernel thread associato ad ogni user thread. Ad ogni modo ci possono essere alcune restrizioni in modo da evitare la creazione 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 pieno un sistema multicore.

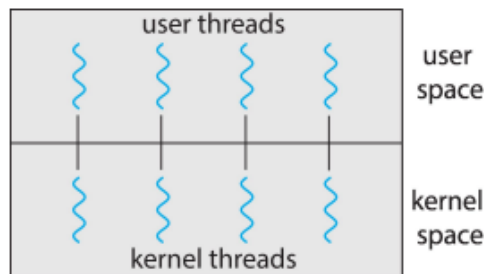


Figura 16: Il modello di *multithreading* uno a uno.

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

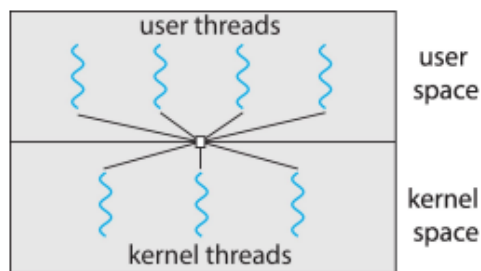


Figura 17: Il modello di *multithreading* molti a molti.

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 librerie 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 kernel ma il programmatore ha meno da implementare.

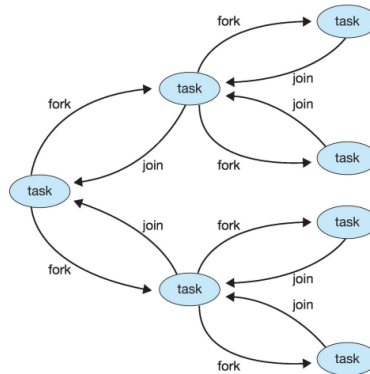


Figura 18: Rappresentazione grafica del modello fork-join per la risoluzione di un task.

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 devono essere proprio scritte nel codice; per esempio: `#pragma 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 suoi 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 chiamata 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 pena copiare tutti i thread se tanto si che verranno eliminati.
- ◊ Se invece il nuovo processo deve supportare anch'esso il multithreading, allora ha senso effettuare una copia 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.