Sistemi Operativi

Ionut Zbirciog

19 October 2023

1 Processi

1.1 Definizione

Un processo è un programma in esecuzione.

In un sistema multiprogrammato, ciascuna CPU passa rapidamente da un processo all'altro, eseguendo ognuno per decine o magari centinaia di millisecondi. La CPU in realtà esegue un solo processo alla volta (la CPU viene assegnata a turni a diversi processi), ma nel corso di 1 secondo può elaborarne parecchi e dare l'illusione del parallelismo (**pseudoparallelismo**). Mentre si parla di **multiprocessore** quando ci sono 2 o più processori a livello hardware.

1.1.1 Il modello del processo

In questo modello tutto il software eseguibile sul computer, incluso il sistema operativo, è organizzato in un certo numero di processi. Un processo, che è un programma in esecuzione, include i valori attuali del contatore di programma, dei registri e delle variabili. Concettualmente, ogni processo ha la sua CPU, ma nella realtà la CPU passa da un processo all'altro, dando la sensazione all'utente di esecuzione in parallelo. Questo rapido passare avanti e indietro è detto multiprogrammazione.

1.1.2 Come funziona il processo

- Esecuzione sequenziale: Un unico program counter, ogni processo è in un'unica posizione, e la CPU, essendo una sola, passa da un processo all'altro in modo sequenziale.
- Esecuzione parallela: Ogni processo ha un proprio flusso di controllo (propri puntatori, propria area di memoria). Ogni volta che si passa da un processo all'altro, si salva il contatore di programma e le variabili nella tabella dei processi del primo processo e si ripristina il contatore di programma del secondo. All'interno del sistema operativo esiste un modulo (scheduler) che, in base alla priorità e a dei criteri detti policy di scheduling, decide quale processo deve essere assegnato prima o dopo alla CPU. Tutto questo avviene perché l'unico obiettivo del sistema operativo è di ottimizzare al massimo le risorse a disposizione, dando l'illusione di essere super veloce. Lo scheduler può prendere decisioni anche controintuitive, come per esempio scegliere un processo che sta per terminare.

In linea di principio, i processi multipli sono reciprocamente indipendenti, quindi hanno bisogno di mezzi espliciti per dialogare tra loro. Il sistema operativo normalmente non offre garanzie di

tempestività o di ordine, per esempio, non offre la garanzia che ogni n secondi verrà assegnata la CPU ad ogni processo per t secondi. È un meccanismo molto complesso che può dipendere anche dal carico effettivo della macchina.

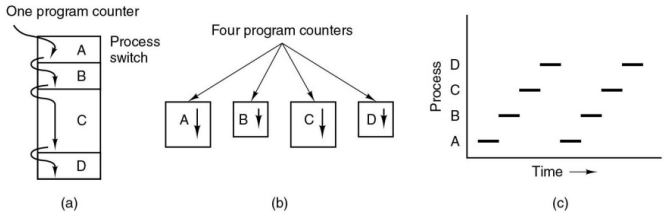

Figure 1: Esecuzione di processi: sequenziali e concorrenti

1.1.3 Informazioni associate a un processo

- PID (Process ID), UID (User ID), GID (Group ID)
- Spazio degli indirizzi di memoria
- Registri hardware (Program Counter)
- File aperti
- **Segnali**
- **Interrupt**

1.2 Gerarchia di Processi

Generalmente, i processi sono legati da una stretta gerarchia, ognuno è responsabile di altri processi perché devono sempre risalire a chi è responsabile, in quanto è l'unico modo per assicurarsi che una volta che uno termina, terminano tutti gli altri generati da quel processo. Avremo sempre la gerarchia Parent - Child. In UNIX, un processo speciale chiamato init (PID 1) è presente nell'immagine di boot. Quando comincia la sua esecuzione, esso legge un file che indica quanti terminali sono presenti. Poi esegue il fork di un nuovo processo per ogni terminale. Questi processi attendono che qualcuno esegua il login. Se il login avviene correttamente, il processo di login esegue una shell che accetta i comandi. Questi comandi possono far partire altri processi e così via. Nei moderni sistemi init, avvia kthreadd (PID 2), un processo per la gestione dei thread.

1.3 Creazione del Processo

La creazione di un processo può essere caratterizzata da 4 eventi principali:

1. Inizializzazione del sistema: All'avvio del sistema operativo vengono creati parecchi processi. Alcuni sono processi attivi, che interagiscono con gli utenti, alcuni sono processi in background (servizio per email, server web, ecc.), chiamati anche demoni.

2. Esecuzione di una chiamata di sistema per la creazione di un processo da parte di un processo in esecuzione (fork()).
3. Richiesta dell'utente di creare un nuovo processo (per esempio tramite bash).
4. Avvio di un lavoro in modalità batch (tramite script sh).

1.4 Termine di un Processo

Condizioni tipiche che terminano un processo:

1. Uscita normale (volontaria).
2. Uscita a causa di un errore (volontaria), ad esempio in C, 'return 0'.
3. Errore 'fatale' (involontario), ad esempio in C, 'segmentation fault'.
4. Ucciso da un altro processo (involontario).

Comandi per la gestione di un processo:

- 'fork': crea un processo, il figlio è un clone "privato" del genitore e condivide alcune risorse del genitore (program counter, registri).
- 'exec': esegue un nuovo processo, viene utilizzato in combinazione con 'fork' (in C, 'exec' esegue comandi, quindi non eseguire mai comandi con 'sudo' in C).
- 'exit': causa la terminazione volontaria del processo, lo stato di uscita viene restituito al processo genitore.
- 'kill': invia un segnale a un processo (o a un gruppo), può causare la terminazione involontaria di un processo.

1.5 Gli Stati di un Processo

1. Running (In esecuzione): sta effettivamente utilizzando la CPU in quel momento.
2. Ready (Pronto): eseguibile, temporaneamente fermo per consentire l'esecuzione di un altro processo.
3. Blocked (Bloccato): non può essere eseguito fino a quando non si verifica un evento esterno (ad esempio, sta aspettando una risorsa). È uno stato fondamentale per evitare uno spreco dell'utilizzo della CPU a causa della lentezza del mondo fisico.

Dal punto di vista logico, il primo e il secondo stato sono simili. Il terzo stato è diverso dai primi due perché il processo non può essere eseguito, neache se la CPU non ha niente da fare. Fra i 3 stati sono disponibili 4 transizioni.

1. Il processo si blocca in attesa di input, in alcuni sistemi il processo può eseguire una chiamata di sistema tipo pause(), per entrare in stato bloccato.
2. E' causata dallo scheduler, infatti lo scheduler può decidere di finirie l'esecuzione di un processo per sceglierne un altro, per esempio un processo con priorità maggiore.

3. E' causata dallo scheduler, infatti lo scheduler sceglie il processo che è in stato di pronto per eseguirlo.

4. Accade quando si verifica l'evento esterno di cui un processo era in attesa (come l'arrivo di un input). Se nessun altro processo è in esecuzione in quel momento, sarà innsescata la 3 transiazione è il processo inzierà ad essere eseguito. Altrimenti dovrà attendere nello stato di pronto.

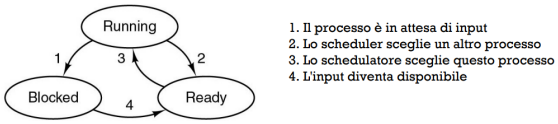

Figure 2: Gli stati di un processo

1.6 Signals vs Interrupts

Sono meccanismi utilizzati nei sistemi operativi e nelle applicazioni per gestire eventi asincroni (eventi indipendenti dal flusso di esecuzione del software).

1.6.1 Interrupts

- Origine: Dispositivi hardware (notificano l'avvenimento di eventi legati all'hardware, ad esempio quando si preme un tasto sulla tastiera).
- Gestione: Tramite routine di servizio di interrupt (ISR), routine specifiche e programmate all'interno del sistema operativo.
- Uso: Comunicazione tra hardware e software.
- Asincronia: Eventi che si verificano in modo asincrono e devono essere gestiti immediatamente.

Interrupt Vector: Associato a ciascun dispositivo di I/O e linea di interrupt, che contiene l'indirizzo dell' ISR (Interrupt Service Routine). Quando avviene un interrupt, per esempio dal disco, il program counter del processo, lo stato attuate del processo, e qualche registro vengono spinti sullo stack (viene salvato lo stato del processo). Il computer poi salta all'indirizzo specificato nel vettore di interrupt e viene eseguita la ISR rispettiva.

Tutti gli interrupt inziano salvando i registri del processo attualmente in esecuzione. Questa procedura non può essere espressa in linguaggi come C, perciò sono eseguite da una piccola routine in assembly, generalemente la stessa per tutti gli interrupt.

Quando la routine è terminata, viene chiamata una procedura C per completare il lavoro per lo specifico tipo di interrupt. Quando ha finito il suo lavoro, magari rendendo pronto qualche processo viene chiamato lo scheduler per stabilire quale processo eseguire. A questo punto, il controllo viene restituito al codice in linguaggio assembly che carica i registri e la mappa della memoria del processo attuale e inzia ad eseguirlo.

Un processo può essere interrotto anche migliaia di volte durante la sua esecuzione, ma il concetto fondamentale è che dopo ogni interrupt il processo torna esattamente allo stato in cui si trovava prima che avvenisse l'interrupt.

1.6.2 Signals

- Origine: Eventi software (modo di comunicare tra i processi), generati da un processo o dal sistema operativo.
- Gestione: Gestori di segnali personalizzati o comportamento predefinito (ad esempio, CTRL + C per la terminazione di un processo o CTRL + Z per la sospensione di un processo).
- Uso: Gestione condizioni eccezionali nelle applicazioni.
- Asincronia: Inviati asincronamente ma possono essere gestiti in modo sincrono.

Un segnale, è un impulso asincrono trasmesso da un processo ad un altro, ed è uno degli strumenti di comunicazione tra processi. Tipicamente nessun dato viene trasmesso assieme al segnale. Ogni tipo di segnale ha un comportamento standard come SIGINT (interruzione del processo), SIGTERM (Terminazione del programma, inciato dal comando kill se non diversamente specificato.), SIGKILL (Terminazione immediata (kill) del processo.). La gestione dei segnali può essere complicata, con alcune condizioni che sono specifiche per un thread, mentre altre no.

Ecco un esempio di gestione di un segnale in C:

```
void signalHandler(int signum){
  printf("Interrupt signal %d received\n", signum);
  exit(signum);
}

int main(){
  signal(SIGINT, signalHandler); // Chiamata di sistema che imposta una funzione per gestire il segn
  while(1){
    printf("Going to sleep....\n");
    sleep(1);
  }
  return 0;
}
```

2 Thread

Nei sistemi operativi tradizionali, ogni processo dispone di uno spazio degli indirizzi e di un singolo thread di controllo. Tuttavia, ci sono frequenti situazioni in cui è utile avere più thread di controllo in esecuzione nello stesso spazio di indirizzi, quasi in parallelo. La multithread execution implica che un processo può avere N thread in esecuzione.

Perché consentire più thread per processo? Innanzitutto, i thread sono dei miniprocessi leggeri, poiché a differenza dei processi, non hanno un loro PID, un loro spazio di indirizzamento, ecc. (lightweight processes). Inoltre, consentono un parallelismo efficiente in termini di spazio e di tempo, in quanto non devono essere gestiti dallo scheduler ma è il processo stesso che li deve gestire.

Un esempio per dimostrare l'utilità dei thread è la gestione delle pagine web da parte di un web server. Ipotizzando che ad ogni richiesta di un client, il web server avvia un nuovo processo per la gestione della richiesta, se ci fossero milioni di richieste, allora il web server dovrebbe avviare

milioni di processi simultanei, un numero molto elevato e che lo scheduler non sarebbe in grado di gestire. Invece di avviare un processo per ogni richiesta, il web server attribuisce un thread ad ogni richiesta, che è molto più leggero di un processo e che viene gestito direttamente dal web server e non dallo scheduler. Così facendo, il processo durante il tempo in cui è allocato alla CPU riesce a rispondere a N richieste.

I thread si trovano tutti nella stessa area di memoria del processo che li ha generati, quindi sono in grado di scrivere e leggere dalla stessa memoria condivisa. Questa caratteristica dei thread comporta anche dei problemi di sincronizzazione tra thread, poiché se un thread sta scrivendo su una variabile e un secondo thread legge e modifica la stessa variabile, il thread iniziale quando andra a leggere sulla variabile si aspettera di leggere quello che ha scritto lui, invece troverà qualcos'altro. Le uniche informazioni personali di ciascun thread sono: stack, registri, memoria, stato. Ciascun thread inoltre può chiamare qualsiasi chiamata di sistema supportata dal sistema operativo per conto del processo a cui appartiene.

I thread in POSIX:

1. `pthread_create: crea un nuovo thread`
2. `pthread_exit: termina il thread chiamante`
3. `pthread_join: attende l'uscita di un specifico thread`
4. `pthread_yield: rilascia la CPU per consentire l'esecuzione di un altro thread`
5. `pthread_attr_init: crea e inizializza la struttra di attributi di un thread`
6. `pthread_attr_destroy: rimuove la struttura di attributi di un thread`

2.1 Implementazione dei Thread nello Spazio Utente (JAVA)

Pro

- Sono gestiti dal kernel come processi a singolo thread.
- Possono essere eseguiti su sistemi operativi che non supportano direttamente i thread.
- Sono gestiti tramite una libreria.
- Offrono l'abilità di personalizzare l'algoritmo di scheduling per ogni processo e una maggiore scalabilità.

Contro

- Problemi con le chiamate di sistema bloccanti.
- Se un thread fa una chiamata che lo blocca, tutti gli altri thread nel processo vengono fermati.
- Gli errori di pagina, dove un programma accede a memoria non presente, possono bloccare l'intero processo quando sono causati da un thread a livello utente.
- I thread nello spazio utente non hanno un interrupt del clock, rendendo impossibile uno scheduling di tipo round-robin.
- Sebbene i thread a livello utente siano più veloci e flessibili, sono meno adatti per applicazioni in cui i thread si bloccano frequentemente, come i web server multithread.

2.2 Implementazione dei Thread nello Spazio Kernel

Pro

- Il kernel che gestisce i thread elimina la necessità di un sistema run-time per processo.
- Le chiamate di sistema che potrebbero bloccare un thread vengono implementate come chiamate di sistema.
- Alcuni sistemi riciclano i thread per ridurre i costi, invece che terminarli.
- La programmazione con thread richiede cautela per evitare errori.

Problemi aperti

- Molte procedure di libreria possono causare conflitti se un thread sovrascrive dati cruciali per un altro.
- L'implementazione di wrappers può evitare conflitti ma limita il parallelismo.
- La gestione dei segnali è complicata, con alcuni specifici per un thread e altri no.