La Bibbia di Sistemi operativi

Mario Petruccelli
Università degli studi di Milano

A.A. 2018/2019

Sommario

1 Virtualization

1.1 Introduzione

Processi Un processo, informalmente, è un programma in esecuzione. Un programma a sua volta, è una sequenza finita di istruzioni scritte in un linguaggio comprensibile all'esecutore (CPU). L'esecuzione di un programma da parte del processore è:

- **Fetch** Prelievo istruzione dalla memoria.
- **Decode** Decodifica dell'istruzione.
- **Execute** Esecuzione dell'istruzione.

1.1.1 Virtualizzazione

La virtualizzazione consiste nel prendere una risorsa fisica e trasformarla in una più generale, potente e facile da adoperare forma virtuale di se stessa.

Virtualizzazione della CPU L'illusione consiste nel far credere che il sistema abbia un elevato numero di cpu virtuali. Avere più CPU permeterrebbe a più programmi di essere eseguiti in **parallelo** nonostante il processore fisico effettivo sia uno solo. Se due processi vogliono essere eseguiti entrambi ad un certo tempo, oppure vogliono accedere alla stessa periferica, quale dei due ha la priorità? La risposta viene data con l'introduzione delle politiche di priorità (**politiche di scheduling**).

Virtualizzazione della memoria Consiste nel fabbricare l'illusione che ogni processo abbia il proprio spazio di indirizzi virtuali privato (**address space**) al quale accede e sarà il sistema operativo ad occuparsi di mappare nella memoria fisica.

Con la virtualizzazione è fondamentale riuscire a distinguere i processi in esecuzione. Per fare ciò viene associato un **PID** (process id) ad ogni job. il PID è un numero univoco.

1.1.2 Concorrenza

Si riferisce a tutta quelle serie di problemi che sorgono, e che vanno risolti, quando all'interno dello stesso programma più entità lavorano in parallelo. Le entità in questione si chiamano **threads**.

1.1.3 Persistenza

La persistenza è legata alla memorizzazione dei dati all'interno della memoria. La non volatilità delle memorie ha introdotto la possibilità di memorizzare dati in modo persistente. Il software nel sistema operativo che generalmente gestisce i dischi è chiamato **file system**.

1.1.4 Protezione ad anelli

Un modello di potezione implementato dal sistema operativo è quello ad anelli. Ci sono 5 livelli e 3 anelli differenti. A ciascun anello corrisponde un relativo livello di sicurezza.

- **Level 1** *Hardware level* qui vengono eseguiti, ad esempio, i device drivers visto che essi richiedono accesso diretto all'hardware dei dispositivi (microcontroller).
- **Level 2** *Firmware level* Il firmware sta in cima al livello elettronico. Contiene in software necessario dal dispositivo hardware e dal microcontroller.
- **Level 3: ring 0** *Kernel level* Questo è il livello dove opera il kernel, dopo la fase di bootload siamo qui.
- **Level 4: ring 1 e 2** *Device drivers* I device drivers passano atraverso il kernel per accedere all'hardware.
- **Level 5: ring 3** *Application level* Qui è dove viene eseguito normalemente il codice utente.

1.2 Processi

Sistema multiprogrammato Sistema nel quale è possibile eseguire più programmi contemporaneamente, idea alla base della virtualizzazione.

1.2.1 Multiprogrammazione

Time sharing prevede che il tempo di CPU sia equamente diviso fra i programmi in memoria.

Real time sharing La politica di scheduling è differente. Alcuni processi vanno serviti prima di altri.

1.2.2 Virtualizzazione della CPU

L'illusione consiste nel rendere indipendenti il numero di processi dal numero di processori. Si vuole disaccoppiare le entità logiche (*processi*), dalle entità fisiche (*processori*), in modo tale che ad ogni processo venga assegnato un processore logico mappato su processore fisico.

I concetti fondamentali alla base della virtualizzazione sono:

- **Time sharing** Meccanismo mediante il quale il tempo di CPU viene diviso equamente fra i processi.
- **Context switch** Meccanismo che consente di interrompere l'esecuzione di un processo in corso sulla CPU fisica e assegnare quest'ultima ad un nuovo processo.

1.2.3 Processi

Un processo è un programma in esecuzione, il sistema operativo deve fornire alcune interfacce (**APIs**) per la gestione dei processi che permettano di fare:

- **Create** Creazione di un nuovo processo.
- **Destroy** Eliminazione forzata di un processo. Molti processi termineranno per conto loro, ma l'utente potrebbe voler eliminare processi non ancora terminati.
- **Wait** Mette in attesa un processo.
- **Miscellaneous control** Sospensione di un processo per farlo ripartire dopo un certo tempo.
- **Status** Interfacce che restituiscono lo stato e altre informazioni di un processo.

Creazione di un processo La prima cosa che deve fare il sistema operativo per eseguire un programma è caricare il suo codice ed eventuali dati statici da disco a memoria, nell'address space del processo.

- **Allocazione dello stack** Un po' di memoria deve essere creata per lo stack del programma (*variabili locali, parametri delle funzioni e indirizzi di ritorno*).
- **Allocazione dello heap** Un po' di memoria deve essere creata per lo heap del programma (*dati allocati dinamicamente*).
- **Inizializzazione I/O** Standard input, output ed error.
- **Salto ed esecuzione** Salto all'entry point ed esecuzione. (*main*)

Stato di un processo

- **Running** È in esecuzione sul processore.
- **Ready** In attesa di essere eseguito dal processore.
- **Blocked** In stato di block, il processo sta esegundo qualche operazione (*es: I/O*).

Strutture dati Il sistema operativo deve tenere traccia delle informazioni fondamentali di un processo per poter ripristinare l'esecuzione di un processo interrotto. Esse sono:

- Porzioni di memoria coinvolte.
- Valori dei registri di CPU usati dal processo.
- Stato dei dispositivi di I/O usati dal processo.

Questi dati sono organizzati in strutture chiamate **Process Control Block (PCB)**, salvate in un per-process **kernel stack**, il quale risiede nel kernel space.

1.2.4 Process API

La creazione di un processo avviene tramite la `fork()`, la quale genera un processo identico a quello in esecuzione. Tale processo prende il nome di padre, quello generato viene chiamato figlio. L'esecuzione del processo figlio parte dall'istruzione successiva alla `fork()`. La `fork()` ritorna al figlio 0, al padre il **PID** del figlio e **-1** in caso di errore. Il processo figlio avrà il **proprio** address space, registri, PC, ecc...

La `wait()` è una funzione che forza il padre ad aspettare che il processo figlio termini la propria esecuzione. Senza, l'output potrebbe essere **non-deterministico** e potrebbero crearsi processi orfani o zombie. Esiste anche la `waitpid` che viene usata se si ha a che fare più di un figlio.

Per eliminare un processo esiste la funzione `kill()`. Solo il padre può distruggere il figlio. Ciò può portare alla creazione di processi **zombie** (processi terminati la cui **PCB** è ancora in memoria).

La `exec()` serve per generare un processo che fa qualcosa di diverso da quello padre. `exec(nome_programma, arg)` prende il nome di un eseguibile e alcuni argomenti, carica il codice e i dati statici di quell'eseguibile, sovrascrivendo il code segment corrente all'interno del PCB del figlio. Heap, stack e altre parti di memoria vengono re-inizializzate. Rimane la relazione padre-figlio.

1.3 Context Switch

1.3.1 Shell

Come mai `fork()` ed `exec()` sono due system call separate? Per rispondere introduciamo la **shell**.

La shell è un programma del sistema operativo `Unix` il cui compito è riconoscere ed eseguire altri programmi; si può dire che essa sia il genitore di tutti i processi che vengono mandati in esecuzione. Nello specifico, essa esegue una `fork()`, cambia il file descriptor se richiesto, ed infine invoca la `exec()`. Poi si mette in attesa che il programma abbia terminato prima di tornare in attesa di istruzioni. Esistono due tipi di shell, grafica (*terminale*) e interattiva (*aprire programmi col mouse*).

La separazione di `fork()` ed `exec()` è dovuta alla presenza della shell, con la quale possiamo andare ad effettuare alcune modifiche dopo la `fork()` e prima dell'`exec()`, come ad esempio la sostituzione del file descriptor.

```
$> wc file.c > n.txt
```

La shell esegue la `fork()` per poter mandare in esecuzione il programma `wc`. Prima di sostituire il codice del padre all'interno del PCB del figlio, sostituisce il file descriptor relativo allo standard output con n.txt. Successivamente esegue l'`exec()` producendo l'output desiderato all'interno di n.txt. Queste manipolazioni non sarebbero possibili se `fork()` ed `exec()` fossero un'unica system call perchè non si avrebbe accesso al PCB del figlio prima dell'exec().

1.3.2 Direct execution

Il concetto di direct execution è semplice: il programma viene eseguito direttamente sulla CPU fisica.

Quando il sistema operativo desidera iniziare l'esecuzione di un programma, viene fatto quanto segue:

- Crea una entry nella lista dei processi.
- Alloca la memoria per il programma.
- Carica il programma in memoria.
- Imposta lo stack con `argc/argv`.
- Pulisce i registri.
- Esegue la chiamata a `main()`.
  Si ha un salto dalla zona kernel al `main`. Il processo a questo punto deve:
- Eseguire il codice del `main()`.
- Ritornare dal main a fine esecuzione.
  Dal processo si torna alla zona kernel. Il sistema operativo infine:
- Rimuove la entry dalla lista dei processi.

Tuttavia la direct execution solleva alcune problematiche:

- Il sistema operativo non può assicurarsi che un programma in esecuzione non faccia qualcosa che non dovrebbe fare.
- Il sistema operativo non può fermare un processo in esecuzione.

Il primo problema si risolve con l'introduzione dello **user mode**. Il codice che viene eseguito in questa modalità di elaborazione è limitato in termini di istruzioni eseguibili. Nasce quindi anche la **kernel mode**, modalità in cui opera il sistema operativo e che consente di eseguire tutte le istruzioni privilegiate.

Per permettere ad un processo di eseguire istruzioni privilegiate vengono introdotte delle **system call**. Per eseguirle, un programa deve eseguire un'istruzione **trap** (*interrupt via software*). Questa istruzione salta nel kernel, aumenta i privilegi a kernel mode, esegue le operazioni privilegiate e ritorna al processo scalando i privilegi tramite un'istruzione **return-from-trap**. Durante questo procedimento bisogna assicurarsi di salvare i registri del chiamante. Per sapere dove la trap deve saltare, il kernel imposta una **trap table** al boot time. Non è il processo utente a specificare l'indirizzo dei **trap handlers** perchè potrebbe saltare ovunque nel sistema.
Per specificare la system call, generalmente viene assegnato un **system-call-number** che solitamente viene inserito in un registro appropriato.

1.3.3 Switch tra processi

Cooperative approach Soluzione via software che consiste nel programmare il processo in modo che, dopo un certo numero di secondi di utilizzo della CPU, il comando torni al sistema operativo. Il problema è che se vengono creati loop infiniti nel programma, la CPU non verrebbe mai condivisa.

Time interrupt Soluzione via hardware che consiste nel creare una nuova componente che genera un segnale elettrico (**time interrupt**) dopo un certo lasso di tempo. Ci sarà quindi un orologio interno che invierà un segnale al piedino del microprocessore. L'hardware deve inoltre fermare l'esecuzione del processo corrente, salvarne lo stato per dare il controllo allo **scheduler**, che nel caso decidesse di cambiare processo, farà eseguire al sistema operativo codice a basso livello che prende il nome di **context switch**.

Context switch Ciò che deve fare il sistema operativo è salvare alcuni valori dei registri per il processo in corso di esecuzione (*nel kernel stack*) e ripristinarne altri per il processo scelto. Viene eseguita una return-from-trap per mandare in esecuzione il processo scelto.
Interrupt, system call ed eccezioni sono eventi che inducono il mode switch.

1.4 Scheduling policy

Dati n processi, a quale assegno il processore? La scelta è fatta dallo **scheduler**, un modulo del sistema operativo che implementa una politica decisionale.

CPU burst è l'intervallo di tempo in cui viene usata intensamente la CPU.

I/O burst è l'intervallo di tempo in cui viene usato intensamente I/O.

CPU bound processi con CPU burst lunghi, ad esempio compilatori, simulatori, calcolo del tempo, ecc...

I/O Bound processi con I/O burst lunghi, ciò comporta maggiore interattività con l'utente.

Stato di IDLE è lo stato in cui è una risorsa accesa e funzionante ma non utilizzata.

Un processo in esecuzione si trova o in CPU burst o in I/O burst. Lo scheduler, per essere efficiente, deve ottimizzare l'uso delle risorse in modo tale che, se la CPU è occupata con l'esecuzione di un processo, i dispositivi di I/O lo sono con un altro e viceversa. L'ottimizzazione della CPU viene dunque portata mediante lo scheduler. Per valutare la bontà di un algoritmo di scheduling si devono introdurre delle metriche di valutazione.

$$T_{turnaround} = T_{termine} - T_{arrivo}$$

$$T_{response} = T_{first-exec} - T_{arrivo}$$

$$T_{wait} = T_{turnaround} - T_{job}$$

1.4.1 Algoritmo FIFO

L'algoritmo FIFO (*First In First Out*) mette in esecuzione il primo processo arrivato. Il problema a cui può portare questo algoritmo è l'**effetto convoglio**, ovvero quando un certo numero di piccoli consumatori di una risorsa vengono messi in coda dietro un enorme consumatore.

1.4.2 Algoritmo SJF

L'algoritmo SJF (*Shortest Job First*) mette in esecuzione il processo con CPU burst minore. In questo modo si evita l'effetto convoglio, ma solo se i processi arrivano allo stesso istante.

1.4.3 Algoritmo STCF

L'algoritmo STCF (*Shortest Time to Completion First*) ogni volta che arriva un processo, lo compara il processo in esecuzione e lascia il processore a quello che ha CPU burst minore.

SJF e STCF funzionano molto male per quanto riguarda il tempo di risposta (spesso possono indurre anche al verificarsi della starvation). Inoltre non conoscono a priori il CPU burst di un processo, perciò sono solo algoritmi teorici.

1.4.4 Round Robin

L'algoritmo **Round Robin** assegna un **quanto di tempo** ad ogni processo. Viene inizializzato un timer che, una volta arrivato a zero, forza un context switch. Il quanto di tempo va scelto bene, altrimenti si hanno troppi context switch se è troppo piccolo, o degenera in FIFO se è troppo grande.

1.5 Multilevel feedback scheduler

Il problema che **MLFQ** (*MultiLevel Feedback Queue*) cerca di risolvere è:

- Ottimizzare il $T_{turnaround}$.
- Aumentare l'interattività utente/sistema, minimizzando il $T_{response}$.

L'approccio che si usa consiste nell'avere un certo numero di **code** distinte, ognuna assegnata ad un diverso **livello di priorità**. MLFQ sfrutta i diversi livelli di priorità per decidere quale processo eseguire: viene scelto quello all'interno della coda di priorità maggiore. Se ci sono più processi all'interno di una certa coda, viene usato **RR**.

- 1. If priority(A) > priority(B), A runs (B doesn't).
- 2. If priority(A) = priority(B), A & B run in RR.
- 3. Quando un processo entra nel sistema, viene posizionato nella coda di priorità massima.
- 4a. Se un processo utilizza tutto il lasso di tempo a disposizione durante l'esecuzione, la sua priorità viene ridotta.
- 4b. Se un processo libera la CPU prima di terminare il lasso di tempo a disposizione, il livello di priorità rimane invariato.
- 5. **Priority boost** Dopo un certo periodo di tempo, tutti i processi vengono spostati nella coda di priorità più alta. (*Evita la starvation dei long running jobs e il monopolio della CPU se qualche processo la rilascia poco prima del lasso di tempo.*)

1.5.1 Better accounting

La scelta del tempo è cruciale, se settato troppo grande, i long running jobs potrebbero ancora andare in starvation, se impostato troppo piccolo, i processi interattivi potrebbero non avere una porzione adeguata dela CPU. Per evitare che possa essere raggirato l'algoritmo di scheduling, lo scheduler tiene tracia di quanto tempo ha consumato un processo in un certo livello di **MLFQ**. Le regole 4a e 4b diventano:

- 4. Una volta che un processo ha usato il tempo a disposizione in un certo livello (indipendentemente da quante volte ha rilasciato la CPU), la sua priorità viene ridotta.

2 Concurrency

3 Persistence

4 JOS