# Appunti di Sistemi Operativi

Andrea Franchini

January 9, 2019

# 1 Programmazione concorrente

## 1.1 Processo

Ogni processo è una macchina virtuale indipendente, con eventuali funzioni che gli permettano di intereagire con altri processi. Un processo può avere sottoprocessi, e tutti i processi hanno un padre. Quando un processo termina, terminano tutti i child processes.
Un processo si compone di tre segmenti: *codice*, *dati*, *sistema*.

### 1.1.1 Primitive di un Processo

- `pid_t PID` identifica un processo.
- `pid_t getpid()` restituisce sempre il PID del processo, anche all'interno di un thread.
- `pid_t fork()` clona il parent process, duplicando i segmenti *dati* e *sistema*. Restituisce al parent il PID del child, e al child 0.
- `void exit(int)` termina il processo, restituendo al parent l'intero passato.
- `pid_t wait(int*)` sospende il parent e attende che un qualsiasi child termini. Restituisce il PID del child terminato. Il parametro è l'indirizzo del valore restituito dalla exit del child.
- `pid_t waitpid(pid_t pid, int* status, ...)` sospende il parent e aspetta che finisca il child col PID indicato.
- `execl(char* path, char* argv)` sostituisce i segmenti *codice* e *dati* del processo con quelli del programma specificato. Il processo rimane lo stesso. `path` individua il programma e `argv` è un array di parametri, di cui il primo (`arg0`) deve essere il nome del programma e `arglast` deve essere `NULL`.
- *fork-exec* è una prassi per evitare che la terminazione di un codice lanciato con `execl` termini il processo, visto che in realtà lo sostituisce. Si esegue la `execl` solo se si è nel child process dopo una fork.

## 1.2 Thread

Detti anche *processi leggeri*, più thread possono esistere contemporaneamente all'interno di un processo. I thread di un processo condividono lo stesso spazio di indirizzamento. Se termina un processo terminano anche i suoi thread. A differenza dei processi, i thread **non** hanno un ordine gerarchico parent-children (sono tutti allo stesso livello).

### 1.2.1 Primitive di un Thread

- `pthread_t TID` identifica un thread.
- `pthread_create(pthread_t TID, void* attrs, function, void* param)` equivalente di `fork()`. TID è un indirizzo, attrs è un puntatore agli attributi del thread, di default `NULL`, `param` è l'indirizzo del parametro da passare. In caso di più parametri, si passa una struct.
- `pthread_join(pthread_t TID, void* status)` equivalente di `waitpid()`, TID è il thread di cui aspettare la terminazione e status è l'indirizzo della variabile che la funzione del thread deve restituire.

## 1.3 Tipi di Esecuzione

L'esecuzione può essere sequenziale (deterministica) o concorrente (non-deterministica). Se il processore è single-core *concorrente* è sinonimo di *parallelo*. Su processori multi-core l'esecuzione può essere *simulata* o *reale*. Il processore permette a ciascun processo di funzionare per un po' di tempo, dopodichè lo sospende e passa ad un altro processo.

## 1.4 Costrutti/Tecniche

### 1.4.1 Sequenza

Una sequenza è una successione sequenziale di operazioni eseguite da un thread. Una sequenza si dice *critica* quando le istruzioni in essa devono essere eseguite in successione senza che avvenga alcuna sospensione. Si cerca di garantire la mutua esclusione per tali sequenze.

### 1.4.2 Istruzione Atomica

Un'istruzione atomica è indivisibile a livello di codice macchina. Un'istruzione di un linguaggio di alto livello viene tradotta come sequenza critica in codice macchina.

### 1.4.3 MutEx (Mutual Exclusion)

è un costrutto particolare che permette di evitare che più thread accedano contemporaneamente allo stesso contenuto, bloccando la sequenza critica. Gli altri thread che cercano di accedere al blocco vengono messi in pausa.

- `pthread_mutex_t` è la variabile che tiene conto dello stato del mutex.
- `pthread_mutex_init(pthread_mutex_t*, NULL)` inizializza il mutex.
- `pthread_mutex_lock(pthread_mutex_t*)`
- `pthread_mutex_unlock(pthread_mutex_t*)`

### 1.4.4 Deadlock

Il deadlock è una situazione in cui due thread devono bloccare due risorse A e B ma le bloccano in ordine inverso, finendo così in un loop. Succede quando due mutex sono annidati e bloccano le risorse in ordine inverso:
`lock(A) > lock(B) > unlock(B) > unlock(A)`

### 1.4.5 Sincronizzazione e Semafori

Per avere una relazione temporale deterministica tra due thread, si utilizza il *semaforo*.

- `sem_t` è il tipo della variabile, è un intero che può avere valori positivi, negativi o nulli.
- `sem_init(sem_t*)` inizializza il semaforo a un valore positivo.
- `sem_wait(sem_t*)` decrementa il valore del semaforo. Finché il semaforo è $\geq 0$ l'esecuzione continua; quando è $< 0$ il thread viene messo in attesa.
- `sem_post(sem_t*)` incrementa il valore del semaforo, eventualmente sbloccandolo.

Generalmente, l'ordine di sblocco è di tipo FIFO. Il modulo di `sem_t`, se esso è negativo, indica quanti thread sono stati messi in attesa, altrimenti quanti thread possono utilizzare le risorse senza venire bloccati. In POSIX, il semaforo è solo $\geq 0$, dove 0 corrisponde allo stato di attesa.

# 2 Linux

## 2.1 Processi

Linux è un OS multiprogrammato con time-sharing, dove a ciascun task è assegnato un quanto di tempo, entro il quale viene sospeso.

### 2.1.1 Categorie di Processi

- **Processo Leggero:** equivale a un thread.
- **Processo Normale:** equivale a un processo non leggero.
- **Processo o Task:** equivale indifferentemente a un processo o a un thread.

#### 2.1.2 Primitive

- `gettid()` restituisce il TID del processo (unico per thread).
- `getpid()` restituisce il PID del processo, più thread nello stesso processo hanno lo stesso PID.

#### 2.1.3 Context Switch

La commutazione di processo, cioè la sostituzione del processo in esecuzione con un altro è detta *context switch*.

#### 2.1.4 Scheduler

L'entità che regola quali processi vengono eseguiti e quali sospesi si chiama *scheduler*, e opera la context switch. I processi più importanti vengono eseguiti prima di quelli meno importanti.
Il codice del Kernel di Linux è *non pre-emptable*, cioè non interrompibile quando viene eseguito.

# 3 Gestione dei Processi

## 3.1 Hardware

L'OS deve anche garantire che i task non possano compiere azioni dannose, regolando pertanto l'accesso a processore, memoria e periferiche. Per gestire le periferiche, esistono i driver, librerie che permettono l'astrazione dell'interfaccia con la periferica.

## 3.2 Strutture Dati

Il contesto di un processo è l'insieme di tutte le informazioni che lo riguardano. Esiste anche una parte detta *contesto hardware*, che riguarda i registri della CPU del processo. Quando un processo viene sospeso, tali registri vanno salvati come parte della seguente struttura:

- *Descrittore di Processo:* struttura dati fondamentale del processo.
- *Indirizzo del Descrittore:* fornisce un ID unico per il processo.
- *sPila* : rappresenta la pila del sistema operativo, unica per processo.

#### 3.2.1 Codice del Descrittore di Task

```
struct task_struct {
    // identificatori del task
    pid_t pid, tgid;
    // -1 unrunnable, 0 runnable, >0 stopped
    volatile long state;
    // puntatore alla sPila del task
    void* stack;
    // struct per il contesto HW
    struct thread_struct thread;
}

struct thread_struct {
    // puntatore alla base della sPila del processo
    unsigned long sp0;
    // puntatore alla posizione corrente della sPila del processo
    unsigned long sp;
    // punt. alla pila di modo U (uPila)
    unsigned long usersp;
}
```

## 3.3 Modi di funzionamento della CPU

Il processore può funzionare in due modi: **U** (user, non privilegiato) o **S** (system, privilegiato o kernel). La modalità S può eseguire ogni comando e accedere a tutta la memoria, mentre in modo U si ha accesso solo alla propria porzione di memoria e ad un set limitato di istruzioni.

**PSR:** Program Status Register, contiene le informazioni relative alla situazione del processore.

#### 3.3.1 System Call

Esiste un'istruzione speciale chiamata `SYSCALL`, non privilegiata, che permette di fare un salto all'OS (passare in modalità S).

#### 3.3.2 System Return

Esiste un'istruzione speciale chiamata `SYSRET`, privilegiata, che permettere di ritornare in modalità U. `SYSRET` è eseguita alla fine della `system_call()`

### 3.4 Modello di Memoria

La memoria di un x64 è di $2^{64}$ byte, cioè $2^{24}$ TByte.
Lo spazio in modo U occupa da `0x0000.0000.0000.0000` a `0x0000.7FFF.FFFF.FFFF` cioè $2^{47}$ byte.
Lo spazio in modo S occupa da `0xFFFF.8000.0000.0000` a `0xFFFF.FFFF.FFFF.FFFF`, cioè $2^{47}$ byte.
Gli spazi intermedi sono detti non-canonici e non sono accessibili. Oltre `0x0000.7FFF.FFFF.FFFF` in modo U si ha un errore.

### 3.5 Commutazione di Pila

#### 3.5.1 SYSCALL U $\rightarrow$ S

1. Salva in USP il valore corrente di SP
2. Copia in SP il valore presente in SSP (così SP punta a sPila)
3. Salva su sPila il valore del PC del programma chiamante a cui ritornare.
4. Salva su sPila il valore del PSR del programma chiamante.
5. Carica in PC e PSR i valori dell'array di syscall.

#### 3.5.2 SYSRET S $\rightarrow$ U

1. Ripristina in PSR il valore di sPila.
2. Ripristina in PC il valore di sPila.
3. Copia in SP il valore presente in USP (così punta su uPila).

### 3.6 Interrupt

Esiste un sistema di eventi che vengono rilevati dall'hardware a cui è associata una *routine di interrupt*, che interrompe l'esecuzione del processo e esegue la funzione associata all'interrupt. Quando questa termina, si ritorna all'esecuzione del programma che è stato interrotto.
L'istruzione che esegue il ritorno da interrupt è `IRET`.
Esiste una *tabella degli interrupt* che contiene coppie di <PC, PSR>.
Non tutti gli interrupt vengono eseguiti subito, ma viene confrontato con la priorità (contenuta nel PSR) del processo in corso (che diminuisce nel tempo).

### 3.7 Gestione dello Stato dei Processi

#### 3.7.1 Stato del Processo

Un processo può avere uno dei seguenti stati, contenuto nel Descrittore:

- Attesa: deve aspettare il verificarsi di un certo evento per andare in esecuzione. Un processo viene messo in stato di attesa quando esegue un servizio di sistema, oppure in caso di *preemption*, cioè quando lo scheduler decide di sospenderne l'esecuzione per dare la precedenza ad un altro processo.
- Pronto: può essere messo in esecuzione dallo scheduler. Il processo in esecuzione è comunque in stato di pronto e si chiama *current process*.

### 3.7.2 Contesto del Processo

Un processo può avere due contesti di esecuzione:

- modo U: generalmente, un processo è per la maggior parte del tempo in questo stato.
- modo S: quando si fa uso di una funzione dell'OS oppure avviene un interrupt, il processo è in modo S, ovvero è in esecuzione il sistema operativo.

### 3.7.3 Scheduler

- Gestisce la *politica di scheduling*, cioè la regolazione del tempo di esecuzione dei singoli processi.
- Esegue la *context switch*, svolta dalla funzione `schedule()`.
- Gestisce la *runqueue*, una struttura dati del processore che ha due campi:
- – `RB`: lista di pointer a tutti i processi in stato di pronto.
  – `CURR`: pointer al *current process*.

### 3.7.4 Gestione dell'Interrupt

La gestione dell'interrupt è *trasparente*, cioè avviene senza disturbare il processo in esecuzione. Gli interrupt vengono eseguiti nel processo corrente, cioè non sostituiscono il processo in esecuzione.
Se l'interrupt avviene al verificarsi di un evento su cui un processo era in stato di attesa, l'interrupt risveglia il processo portandolo in stato di pronto.

### 3.7.5 Waitqueue

Ogni volta che un processo deve essere messo, in attesa, viene creata una *waitqueue*, che è una lista contenente i puntatori ai descrittori dei processi in attesa.

# 4 Scheduler

## 4.1 Politche di Scheduling

Ogni task ha un campo nel suo descrittore che descrive la politica di scheduling.

### 4.1.1 Round robin (RR)

Il round robin (= a turno) è una politica di scheduling equa, che assegna a ciascu task lo stesso *quanto di tempo* per l'esecuzione. Il fatto che ogni task abbia lo stesso quanto di tempo fa sì che nessun task rimanga fermo indefinitivamente.
Vale la seguente precedenza: FIFO > RR > NORMAL.
FIFO e RR sono generalmente usati per task *soft real-time*. La loro priorità è detta *statica*, perché è un valore da 0 a 99 e non viene aggiornata una volta assegnata.

- **First In First Out (FIFO):** I task vengono eseguiti per intero dopo essere appena selezionati, finché non terminano o si sospendono da soli.
- **Round Robin (RR):** Ogni task ha un quanto di tempo per l'esecuzione. Ci sono code per ogni i-esimo livello che vengono eseguite solo se la (i+1)-esima coda è vuota.
- **Normal (NORMAL):** I task vengono gestiti dinamicamente in tempo reale.

FIFO e RR sono generalmente usati per task *soft real-time*. La loro priorità è detta *statica*, perché è un valore da 0 a 99 e non viene aggiornata una volta assegnata.

## 4.2 Gestione della Runqueue

Lo scheduler interviene per decidere quale task rimettere in esecuzione tra quelli in stato di pronto nella runqueue. Viene scelto il task con il *diritto di esecuzione maggiore*.
Lo scheduler deve scegliere un nuovo task da eseguire quando:

- un task si autosospende o termina.
- un task in stato di attesa viene risvegliato, aumentanto la runqueue (perché potrebbe avere priorità maggiore).
- scade il quanto di tempo (se si usa round robin).

### 4.2.1 Completely Fair Scheduler (CFS)

Il CFS è lo scheduler per i task NORMAL.
Lo scopo ideale è, dato un processore di potenza 1, dare a ogni task di N tasks 1/N tempo di esecuzione, che equivale ad avere una CPU di potenza 1/N. Il vantaggio di CFS è:

1. determinare velocemente la durata del quanto di tempo.
2. assegnare la priorità ai task, dando più tempo ai task importanti.
3. permettere a un task rimasto in attesa molto a lungo di tornare in esecuzione aumentando la sua priorità.

| | |
|---|---|
| NRT | Numero di task in runqueue |
| PER | Periodo di scheduling, in cui tutti i task vengono eseguiti |
| Q | (Q=PER/NRT) Quanto di tempo assegnato al task |

# 5 Memoria Virtuale

## 5.1 Indirizzi