Università degli studi di Modena e Reggio Emilia

Dipartimento di Ingegneria Enzo Ferrari

Operating System Design

Anno Accademico 2024/25

Indice

Capitolo 1

Definizioni

1.1 I/O Device

Capitolo 2

Processi e Thread

2.1 Processi

Processo: è l'*astrazione* di un programma in esecuzione. Il processo è l'astrazione più elementare e più importante che ci può fornire il sistema operativo. Riuscendo ad emulare il comportamento di esecuzione concorrente nonostante la presenza di una singola CPUs.
Un altro paio di definizioni:

- **algoritmo**: in matematica ed informatica ci si riferisce ad algoritmo ad una sequenza finita di istruzioni rigorose matematiche che hanno l'obiettivo di risolvere una determinata classe di un problema specifico o di risolvere un calcolo.
- **programma**: è la sequenza o l'insieme di istruzioni in liguaggio macchina che può essere eseguito.

I moderni computer possono eseguire diverse operazioni nello stesso istante. Descrivendo in maniera rigida quello che effettivamente succede, però, è che ogni CPU in *ogni istante di tempo* esegue **uno e un solo** processo. Tendendo a 0 il tempo riservato a ogni singolo processo è però possibile simulare **parallelismo** definito anche come: ***pseudoparallelism***, che però va in contrasto con il vero parallelismo hardware (multi-CPUs).
Il ***Process Model*** definisce che tutti gli eseguibili del computer, a volte includendo il sistema operativo, vengano organizzati in una serie di **processi sequenziali**. Il pro-

cesso è stato definito come l'istanza di un programma in esecuizione nel quale viene anche incluso il suo ***PCB*** (***Process Control Block***). Il **PCB**, anche noto come *process descriptor* è una struttura che permette di salvare tutte le informazioni che riguardano un determinato processo, ad esempio: *program counter*, i registri e le variabili. Concettualmente possiamo visualizzare che ad ogni processo è associata una CPU virtuale.

> ***Process Switching***
>
> È quando l'*OS* cambia processo in esecuzione sulla CPU.

Per ora considerermo che esista un'**unica CPU**. Questa assunzione non tiene normalmente conto dei moderni *chip* che sono spesso multi-core.
Possiamo visualizzare inizialmente il processo come una tupla che contiene: il programma, degli input, degli output e uno **stato**. Un singolo processore può essere condiviso da n processi con un algoritmo di *scheduling* (*scheduler algorithm*) che viene utilizzato per determinare quando interrompere un processo (se può farlo) e servirne un altro.

> **Processo vs. Programma**
>
> Un programma è qualcosa che può essere salvato su disco, statico; mentre un processo è qualcosa di dinamico e che varia ad ogni sua istanza.
> Un programma può essere eseguito da più processi che però sono distinti l'uno dall'altro.

La **creazione di un processo** può essere indotta da:

- inizializzazione di sistema
- un processo in esecuzione compie una *system call* che inizializza un nuovo processo
- un utente richiede l'esecuzione di un nuovo processo

I processi possono essere eseguiti in *foreground*, ovvero con i quali un utente può interagire, oppure in *background*, che sono "nascosti" all'utente e rispondono a certe specifiche funzioni. Su linux sono presenti decina di processi in background, alcuni anche noti come *daemons*.

In **UNIX** è presente una solo *system call* per creare un nuovo processo: **fork**. Dopo l'esecuzione della *syscall* i due processi, il padre e il figlio, hanno la stessa immagine della memoria, le stesse stringhe di environment e gli stessi file aperti. Normalmente, dopo il figlio, esegue **execve** o una *system call* simile per cambiare l'area di memoria ed eseguire un nuovo programma.
Alcuni implementazioni di **UNIX** condividono la sezione *.text* tra i due visto che non può essere modificata. In alternativa altre implementazioni possono condividere tutta la memoria del padre, in questo caso la memoria è condivisa in maniera ***copy-on-write***, ovvero ogni volta che uno dei due vuole modificare parte della memoria, quel specifico *chunck* viene copiato prima della modifica in una locazione privata della memoria

Un processo una volta creato, inizia la sua esecuzione e in un certo istante di tempo termina la sua esecuzione per una delle seguenti condizioni:

- *normal exit*, il processo termina correttamente. È un tipo di condizione volontaria.
- *error exit*, il processo termina per un errore riscontrato durante la sua esecuzione, gestito. È comunque un tipo di condizione volontaria.
- *fatal error*, il processo termina per un errore riscontrato durante la sua esecuzione che non è stato gestito (una divisione per 0). È una condizione di interruzione involontaria.
- *killed* da un altro processo, ovviamente involontario.

I processi, possono avere una **gerarchia**, ovvero un processo può avere solo un padre, ma n figli. In alcuni sistemi un processo padre e uno figlio sono comunque sempre associati in una certa maniera.

In **UNIX** un processo e tutti i suoi figli sono associati in una certo modo, formando un *process group*. Se un utente invia un segnale CTRL+C, questo segnale è inviato a tutti i membri della gerarchia di processi che, normalmente, sono attivi nella *current window*. Individualmente, ogni processo, può catturare il segnale, ignorarlo o utilizzare l'azione di default che è quello di interrompere il processo.

Ogni **processo** può avere diversi **stati**:

1. ***running***: il processo sta venendo eseguito e quindi occupa la CPU in quell'istante di tempo.
2. ***ready***: il processo è eseguibile, ma temporaneamente bloccato per permettere ad un altro processo di eseguire.
3. ***blocked***: impossibilità di eseguire il processo finché non avviene un'azione esterna.

Figura 2.1: ***Stati*** *di un processo*

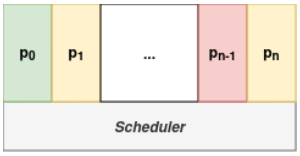
Un processo può essere **bloccato** per due principali motivi: è in attesa di un input che però non è ancora disponibile oppure perché, anche se il processo è **pronto** e potrebbe essere eseguito, il sistema potrebbe aver deciso di allocare la CPU ad un altro processo. Queste due tipologie di sospensioni sono completamente diverse, nel primo caso la sospensione è **inerente** al problema, nel secondo caso è il *OS*.
Facendo riferimento agli stati, gli stati **1** e **2** sono simili, in entrambi i casi è pronto ad eseguire, ma nel secondo stato non è c'è disponibilità di CPU, mentre il **3**rzo stato è sostanzialmente differente, il processo non può eseguire anche se la CPU è *idle*.
Analizzando ora le transizioni da uno stato all'altro, abbiamo:

1. il processo sa che il processo non può continuare la sua esecuzione. In alcuni sistemi il processo può eseguire una *syscall*, come **pause**, per entrare nello stato *blocked*.

   > In **UNIX** quando il processo deve leggere dalla *pipe* o un file speciale (tipo il terminale) e non ci sono input disponibili, il processo è automaticamente bloccato.

2. (è gestito dallo *scheduler*). Avviene quando lo *scheduler* decide che il processo in esecuzione ha eseguito per troppo tempo e vuole permettere ad un altro processo di ottenere il possesso della CPU.
3. quando tutti i processi hanno eseguito per una porzione di tempo equa, il primo processo può riprendere il possesso della CPU per eseguire nuovamente (non è sempre detto che sia il primo, dipende dallo *scheduler* - esistono molti algoritmi di *scheduling* che cercano di bilanciare la richiesta di utilizzo di CPU in maniera efficente (*efficienty*) per il sistema e equa (*fairness*) per il processo).

4. quando l'evento esterno che il processo stava aspettando, avviene (come la scrittura su qualche input da parte di un utente). A quel punto se la CPU è in uno stato di *idle* il processo eseguirà subito la transizione numero **3**, se no dovrà aspettare nello stato *ready* fintanto che la CPU non tornerà disponibile.



Il livello più basso del sistema operativo è lo *scheduler* con tutta la varietà dei processi sopra di lui. Tutti i gestori degli *interrupt* e i dettagli dei processi avviati e stoppati sono nascosti sono nascosti in quello che viene chiamato *scheduler*, mentre il resto del sistema operativo è strutturato e rappresentato tramite l'astrazione di un processo. **Molti pochi dei reali sistemi operativi sono strutturati in questa maniera**.

Per implementare il ***process model***, il sistema operativo deve mantenere in memoria una tabella (un *array* di strutture), chiamata ***process table***, dove ogni processo è una voce di questa tabella (è chiamata anche **PCB**). Al suo interno sono salvati informazioni (Tabella 2.1) che permettono, al processo, di passare dallo stato di *ready* a quello di *running* (e viceversa) come se non si fosse mai interrorto.

gestione di processi	**gestione della memoria**	**gestione dei file**
registri	puntatore al *text segment*	*root directory*
program counter	puntatore al *data segment*	*working directory*
program status word	puntatore al *stack segment*	*file descriptor*
stack pointer		*user id*
process state		*group id*
priorità		
scheduling param		
process id		
processo padre		
process group		
signal (CTRL+C)		
timestamp di inizio		
CPU time		
CPU time dei figli		

Tabella 2.1: Contenuto del *Process Control Block*

Se consideriamo di avere un processo p_0 che però per l'80% della sua esecuzione è in attesa di un I/O, allora quella CPU verrà utilizzata solo al 20%. Come possiamo ottimizzare il tempo di *idle* della CPU. Se dovessimo mettere altri 4 processi p_1, p_2, p_3, p_4 che hanno lo stesso pattern di esecuzione di p_0, idealmente avremo che la CPU verrebbe utilizzata al 100%, ma un modello del genere sarebbe non realisticamente ottimistico, perché si aspetta che i 5 processi non aspettino l'I/O nello stesso istante in maniera sequenziale. Un modello per avere **pseudo-parallelismo** è quello di osservare l'utilizzazione della CPU da un punto di vista probabilistico. Supponiamo che il processo p spende una frazione di p in attesa del completamento di un'azione di I/O, la probabilità che n processi aspettino l'azione di I/O is p^n. Allora l'utilizzazione della CPU è data dalla formula:

$$CPU_{utlization} = 1 - p^n$$

La Figura 2.2 ci mostra come a diversi valori della frazione di p per l'attesa dell'I/O, l'utilizzazione della CPU sia funzione del numero di processi in memoria n, questa

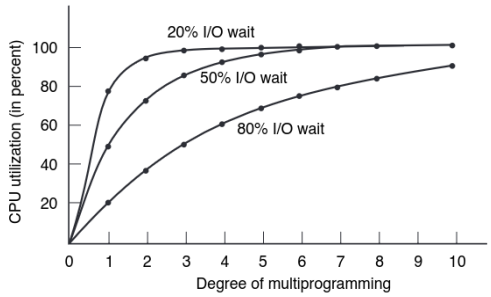


Figura 2.2: *Funzione di utilizzazione della CPU*

condizione è nota come ***degree of multiprogramming***.

Dalla figura è chiaro che se i processi spendono l'80% del loro tempo di esecuzione in attesa, ci devono essere in memoria almeno $n = 10$ per avere un *CPU waste* del 10%. È giusto specificare che quello che descrive il modello probabilistico è un'approssimazione. Assume, implicitamente, che i gli n processi sono indipendenti, il che significa che è accettabile per un sistema con 5 processi in memoria di averne 3 di questi in esecuzione (*running*) e due in attesa (*ready*), ma con un'unica CPU è impossibile avere 3 processi attivi contemporaneamente. Un modello più accurato è possibile costruirlo utilizzando la teoria delle code (***queueing theory***).