Titolo

Daniele De Micheli

2019

Indice

Parte I
Prima parte

1 Processi

1.1 Concetto di processo

I processi rappresenta la prima e più importante astrazione a livello software per un sistema operativo. Un SO esegue infatti un certo numero di programmi contemporaneamente; ogni programma rappresenta un **processo**, e questi processi vengono eseguiti in maniera sequenziale. Un processo è composto da diverse parti:

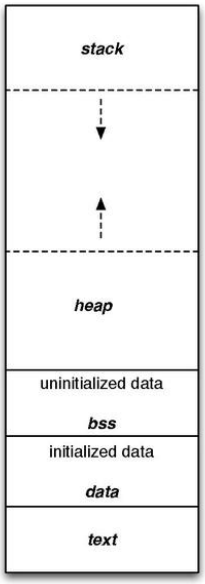
- Lo stato dei registri del processore, incluso il program counter.
- Il codice del programma (*text section*) - PID -.
- Lo **stack** delle chiamate, contenente parametri, variabili locali e indirizzo di ritorno (compreso lo *stack pointer*).
- La **data section**, contenente le variabili globali.
- Lo **heap**, contenente la memoria allocata dinamicamente durante l'esecuzione. Per esempio, in Java viene indicata con *New*, in C con *malloc*.
- Altre risorse acquisite (es. file aperti).

Un porgramma è un'entità passiva (file eseguibile su disco), un processo è un'entità attiva(è un programma in esecuzione). Un progrmma "diventa" un processo quando viene caricato nella memoria centrale. Esso può generare diversi processi:

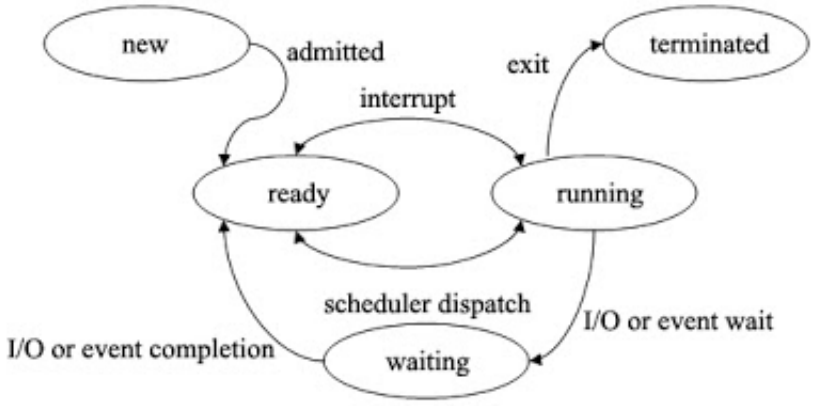
1. Molti utenti eseguono lo stesso programma
2. Uno stesso programma ...

La memoria di un processo è divisa tra stack e heap. Dopo lo heap c'è la sezione **data** (e in linux anche la sezione **bss**) e successivamente la sezione **text**.

Durante l'esecuzione un processo può trovarsi in diversi *stati*. Gli stati possibili sono:



- Nuovo (new): il processo è creato, ma non è ancora ammesso all'esecuzione.
- Pronto (ready): il processo può essere eseguito.
- In esecuzione (running): le sue instruzioni sono in esezuzione su un processore.
- In attesa (waiting): il porcesso non è esecuzione perchè sta aspettando un evento (es. input utente..).
- Terminato (terminated): il processo ha terminato l'esecuzione.



1.1.1 Process Control Block

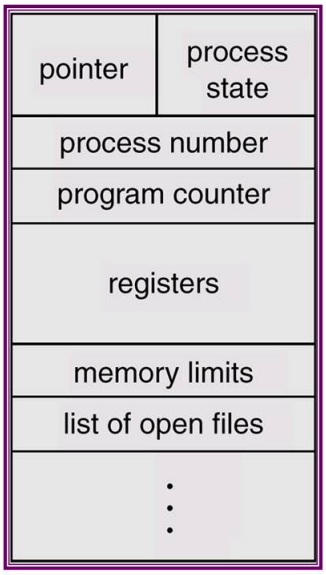
Detto anche "Task Control Block", contiene le informazioni relative ad un processo:

- Process state: ready, running...
- Program number (o PID): identifica il processo
- Program counter: contenuto del registro "istruzione successiva"
- Register: contenuto dei registri del processore
- Informazioni di scheduling: priorità, puntatori a code di scheduling..
- Informazioni relative alla gestione della memoria: memoria allocata al processo
- Informazioni di accounting: CPU utilizzta, tempo trascorso..
- Informazioni su I/O: dispositivi asseganti al processo, elenchi file aperti...

1.1.2 Threads

Fino ad ora abbiamo assunto che un processo abbia un singolo flusso di esecuzione sequenziale. Supponiamo che si possano avete molti program counter per un singolo processo:

1.



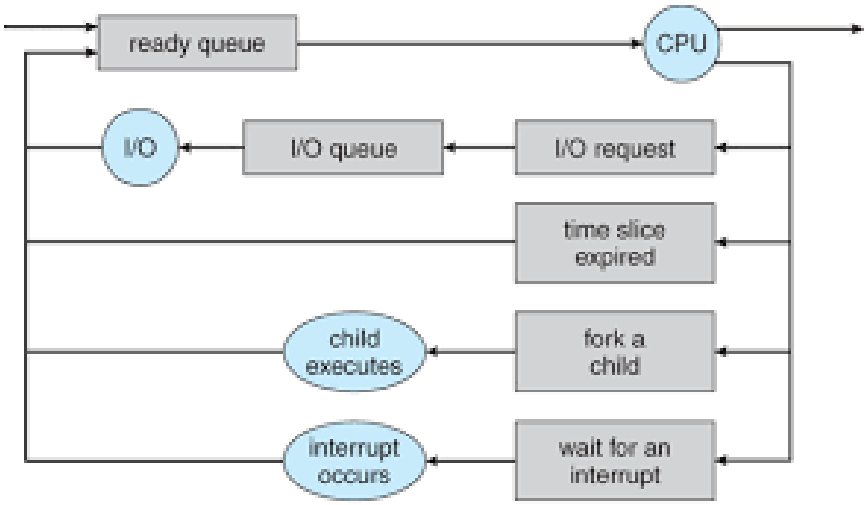
1.1.3 Scheduling dei processi

L'obiettivo dello scheduling dei processi è quello di massimizzare l'utilizzo della CPU. Una tecnica per fare questo è quella del *Time-sharing*:

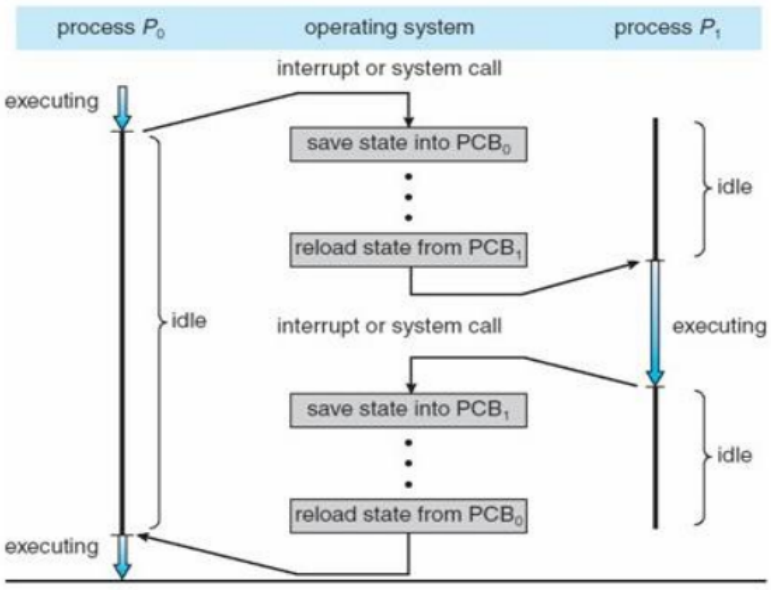
Lo scheduler dei processi sceglie il prossimo processo da eseguire tra quelli in stato ready. Ci sono diverse code di processi:

- Ready queue: processi residenti in memoria
- Wait queue: diverse code per i processi in attesa

Durante la loro vita i processi migrano tra una coda e l'altra.



Quando un SO decide che si deve cambiare processo, si ha la **commutazione di contesto** (o *context switch*). Quando la CPU passa ad eseguire un processo diverso, il sistema operativo deve salvare lo stato del processo precedente, e caricare lo stato salvato del processo da rieseguire attaverso un context-switch. Il PCB rappresenta il contesto di un processo. Il tempo necessario per il context switching è puro overhead: non viene eseguito alcun lavoro utile. Più è complesso l'SO, più è complesso cambiare processo per il context-switch.



Multitasking nei sistemi mobili Alcuni sistemi mobili (es. le prime versioni di iOS) permettevano solo ad un processo di essere in esecuzione. Da iOS4 è possibile avere un processo in esecuzione in **foreground** (ha lo schermo a disposizione) e un certo numero di processi in esecuzione in **background** (senza schermo), ma con dei limiti. Android ha molti meno limiti: i processi in background che vogliono effettuare delle elaborazioni devono creare opportuni *servizi*, che:

- non hanno interfaccia utente
- possono usare un ridotto contenuto di memoria
- possono continuare a funxionare anche quando l'app in backgorund è sospesa

L'aumento di potenza dei sistemi mobili rende i loro OS sempre più simili a quelli non mobili.

1.2 Operazioni sui processi

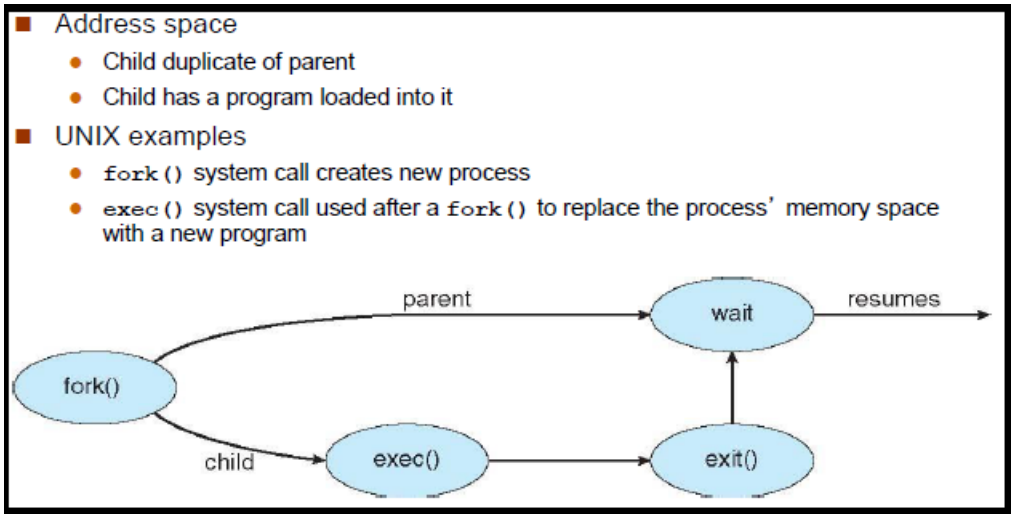
1.2.1 Creazione di processi

Di solito nei sistemi operativi i processi sono organizzati in maniera gerarchica:

- un processo (padre) può creare diversi processi (figli) fino a creare un *albero di processi*.
- PORCODDIO PERCHECAZZO VA COSI VELOCE

Sistemi operativi diversi creano processi in modo diverso. Possono esistere diverse politiche di condivisione (padre e figlio condividono le risorse, solo alcune, nessuna), diverse politiche di creazione di spazio di indirizzi (il figlio è un duplicato del padre (stessa memoria e programma, oppure il figlio deve eseguire qualcos'altro) e ancora politiche di coordinazione padre/figli (il padre è sospeso finchè i figli non terminano, oppure eseguono in maniera concorrente).

Esempio: sistema UNIX



2 lezione mancante

3 Scheduling della CPU

Simulazioni e modellazione lo devo fare io da E-Learning.

3.1 Algoritmi di scheduling della CPU

Come anche per il resto dell'informatica, non esiste un solo algoritmo per svolgere questo compito. Ogni algoritmo ha diverse strategie di azione, e una CPU utilizza in modo intelligente questi algoritmi combinandoli anche tra di loro.

Concetti fondamentali : lobiettivo della multiprograaazione è massimizzare l'utilizzo della cpu. GLI ALGORITMI di scheduling sfruttano il fatto che di norma l'esecuzione di un processo è una sequenza di:

- **Brust della CPU**:sequenza di operzaione di CPU.
- **Brust dell'I/O**: attesa del completamento di operazione I/O.

Un'efficente distribuzione dei brust è essenziale per la CPU.

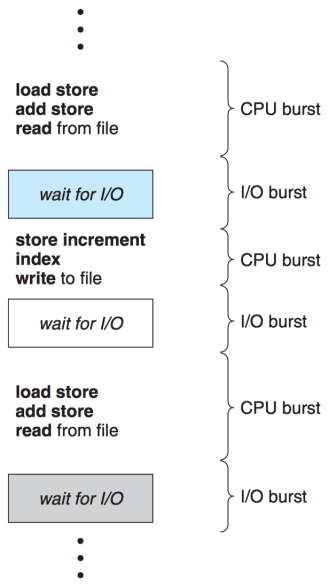


Figure 6.1 Alternating sequence of CPU and I/O bursts.

Distribuzione delle durate dei burst della CPU guardo slide per foto

Scheduler della CPU Lo scheduler della CPU, o *scheduler a breve termine*, seleziona un processo tra quelli nella ready queue (non per forza FIFO) ed alloca un core ad esso. I riassegnamenti della CPU possono essere effettuati in diversi momenti:

1. quando un processo passa da running a waiting.
2. qunado un processo passa da running a ready.
3. quando un processo passa da waiting a ready.
4. quando un processo termina.

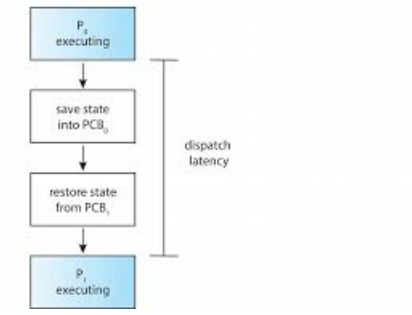
Non è detto che un sistema operativo faccia scheduling su tutte e quattro le situazioni. Se il riassegnamento viene fatto solo nelle situazioni 1 e 4, lo schema di scheduling è detto *senza prelazione* (**nonpreemptive** o cooperativo, altrimenti è detto *con prelazione* (**preemptive**. I processi cooperativi devono appunto collaborare: se volesse, un processo potrebbe tenersi il core occupato per tutto il tempo che desidera. Lo schema di scheduling preemptive è più complicato da implementare ma è anche più sicuro:

- che succede se due processi condividono dei dati?
- che succede se un processo sta eseguendo del codice in modalità kernel?
- che succede se un processo sta eseguendo un gestore degli interrupt?

Dispatcher Il dispatcher passa effettivamente (fisicamente) il controllo della CPU al processo scelto dallo scheduler a breve termine:

- Effettua il csmbio di contesto.
- Passa in modalità utente.
- Salta nel punto corretto del programma del processo selezionato (ossia dove era stato interrotto il processo).

La **latenza di dispatch** è il tempo impegato dal dispatcher per fermare un processo ed avviarne un altro.



3.2 Criteri di scheduling

Misure che servono per confrontare le caratteristiche dei diversi algoritmi (notare che non dioendono solo dall'algoritmo, ma anche dak tipo di carico). I rincipali criteri sono:

- **Utilizzo della CPU**: % di tempo in cui la CPU è attiva (dovrebbe essere tra 40% e il 90%).
- **Throughout**: # di processi che completano l'esecuzione nell'unità di tempo (dipende dalla durata dei processi).
- **Tempo di completamento**: tempo necessario per completare l'esecuzione di un certo processo (dipende da molti fattori: durata del processo, carico totale...).
- **Tempo di attesa**: tempo trascorso dal processo nella ready queue (meglio del tempo di compèletamento , meno dipendente dalla durata del processo e dell'I/O).
- **tempo di risposta**: negli ambienti time-sharing, tempo trascorso tra l'arrivo di una richiesta al processo e la produzione della prima risposta, senza l'emissoine di questa nell'output.

3.3 Algoritmi

3.3.1 Scheduling in ordine di processo

Chiamato anche **first-come-first-served** (o FCFS): la CPU viene assegnata al primo processo che la richiede.

Vantaggio : è molto semplice da implementare (coda FIFO).

Svantaggio : tempo di attesa medio può essere lungo (effetto "convoglio").

FCFS (Example)

Process	Duration	Oder	Arrival Time
P1	24	1	0
P2	3	2	0
P3	4	3	0

Gantt Chart :

P1 waiting time : 0
P2 waiting time : 24
P3 waiting time : 27

The Average waiting time :

(0+24+27)/3 = 17

Ma nella figura, scambiando l'ordine dei processi la situazione può cambiare molto. Quindi dipende molto dall'ordine in cui arrivano i processi.

3.3.2 Scheduling per brevità

Chiamato anche **shortest-job-first** (o SFJ): la CPU viene assegnata in base a l processo che ha il successivo CPU burst più breve.

Vantaggio : minimizza il tempo di attesa medio (è **ottimale**).

Svantaggio : non esiste modo per sapere qual è il processo che avrà il CPU burst più breve! Visto che questo non è possibile nella pratica, si può solo "stimare".

inserisco il paragone tra FCFS e SFJ

Come stimo il burst di un processo?

Idea: registrare la lunghezza dei CPU burst precedenti ed applicare una **media esponenziale**:

1. t_n = durata effettiva dell'n-esimo burst.

2. Sia α un valore compreso tra 0 e 1.

3. Sia $T_{k+1} = \alpha t_k + (1 - \alpha)T_k$

4. La media esponenziale è data da T_{n+1}

La versione preemptive è chiamata **shortest-remaining-time-first**. Il parametro α "bilancia" il peso della storia recente vs. storia passata (di solito si usa $T = 0.5$). Se $\alpha = 0$, la storia non conta, torna il FCFS Se $\alpha = 1$, conta solo la durata dell'ultimo burst.

Con il shortest-remainig-time-first abbiamo che il tempo di attesa medio di un processo è:

$$istante\ terminazione\ processo - (tempo\ di\ arrivo + durata\ burst) \quad (1)$$

ci ficco la foto delle slide

3.3.3 Scheduling circolare

In uno scheduling circolare, o **round-robin** (RR) abbiamo che:

- Ogni processo ottiene una piccola quantità fissata di tempo di CPU
- trascorso tale tempo il processo viene interrotto e messo in fondo alla ready queue
- la ready queue è trattata come un buffer circolare

Se ci sono n processi nella ready queue e il quanto temporale è q, allora ogni processo ottiene 1/n di tempo di CPU e nessun processo attende più di q*(n-1) unità di tempo nella ready queue. Per effettuare la prelazione del processo corrente si effettua un interrupt del timer ogni q di tempo.

foto dello scheduling circolare

Confronto tra algoritmi di scheduling

Experiments		Exp. 1	Exp. 2	Exp. 3	Exp. 4
Scheduling algorithm		FCFS	SJF	Priority	R.R.
Number of processes		50	50	50	50
FCFS ratio		100%	0%	0%	-
SJF ratio		0%	100%	0%	-
Priority ratio		0%	0%	100%	-
Wait time	Min	0	0	0	0
	Max	156	147	363	230
	Mean	34.7	23.4	28.52	41.12
	Sdev.	43.230	33.15	57.439	52.274
Response time	Min	0	0	0	0
	Max	156	147	363	4
	Mean	34.7	23.4	28.52	0.36
	Sdev.	43.230	33.15	57.439	0.8426
Turnaround time	Min	3	3	3	3
	Max	196	246	386	329
	Mean	65.32	54.02	59.14	71.74
	Sdev.	49.253	48.45	65.074	69.499