Università Ca' Foscari Venezia - Corso di Laurea in Informatica
Sistemi Operativi

Problemi di riepilogo sulla prima parte del corso

Prof. Augusto Celentano, anno accademico 2009-2010

Questa raccolta è un campionario di domande e esercizi proposti nel corso degli ultimi anni nelle esercitazioni in aula o come temi d'esame. Per le domande a risposta libera non vengono proposte risposte perché non sarebbero univoche: le domande prevedono una breve trattazione di un argomento che può svolgersi in molti modi. I testi di sistemi operativi, in particolare quelli consigliati, trattano gli argomenti oggetto delle domande con sufficiente chiarezza e profondità.

Parte I - Domande a risposta libera

Gestione dei processi e scheduling della CPU

1. Illustrare la struttura e i benefici di un'architettura di sistema operativo a microkernel.

2. Spiegare il significato del termine "preemption" e illustrare le differenze tra le politiche di scheduling "con preemption" e "senza preemption".

3. Illustrare il problema dell'attesa indefinita (*starvation*) negli algoritmi di scheduling, fornendo almeno un esempio di algoritmo che presenta questo problema, e almeno un esempio di algoritmo che non lo presenta (o che lo risolve, illustrando come).

4. Illustrare i principi di funzionamento di un algoritmo di scheduling della CPU di tipo Shortest Process Next (SPN), discutendone gli aspetti positivi e negativi in termini di equità di trattamento tra i processi, comportamento nei confronti dei processi CPU-bound e I/O-bound e rischio di starvation.

5. Illustrare, anche con opportuni esempi, le differenze di comportamento tra un algoritmo di scheduling real time a priorità e un algoritmo di scheduling a divisione di tempo (time sharing) senza priorità. Si considerino in particolare questi due eventi e si spieghi come si comportano i due tipi di algoritmo:

- viene segnalata la fine dell'esecuzione di un'operazione di I/O su cui un processo era in attesa
- viene creato e mandato in esecuzione un nuovo processo

Gestione della memoria centrale

1. Discutere in che cosa consistono e che obiettivo hanno gli algoritmi di allocazione della memoria centrale:
   a First fit
   b Best fit
   c Worst fit

2. Illustrare le differenze tra la gestione di memoria virtuale a segmenti e a pagine.

3. Descrivere la tecnica di paginazione su richiesta (*demand paging*) e dire quali vantaggi e quali svantaggi comporta nei sistemi multiprogrammati. Illustrare le attività principali necessarie a realizzare tale tecnica, evidenziando quali di queste attività sono svolte a livello hardware (unità di gestione della memoria, MMU) e quali a livello software (sistema operativo).

4. Descrivere l'algoritmo denominato "clock" (algoritmo dell'orologio) per la sostituzione delle pagine fisiche in un sistema a memoria paginata, e discutere le differenze rispetto all'algoritmo LRU.

5. Descrivere le basi concettuali su cui poggia la strategia working set per l'allocazione di memoria centrale ad un processo, e illustrare come è possibile in pratica implementare tale strategia.

File system e dispositivi memoria secondaria

1. Descrivere lo schema di allocazione dei file su disco dei sistemi Windows basato su *File Allocation Table* (FAT), discutendone vantaggi e svantaggi rispetto ad altri schemi (allocazione contigua, concatenata, indicizzata). Descrivere come la dimensione degli elementi della FAT influenza l'efficienza della gestione e la velocità di accesso ai blocchi dei file.

2. Illustrare i sistemi di gestione dello spazio libero di un disco basati sull'uso di bitmap e sull'uso di una lista dei blocchi liberi, discutendone i vantaggi e gli svantaggi in termini di spazio occupato, efficienza e affidabilità.

3. Descrivere vantaggi e svantaggi (o problemi) dell'allocazione contigua dei file su memoria di massa. Ci sono situazioni in cui è sicuramente preferibile agli altri tipi di allocazione (ad es., indicizzata, FAT)?

4. Descrivere lo schema di allocazione indicizzata multilivello dei blocchi dei file su disco, discutendone vantaggi e svantaggi rispetto ad altri schemi (allocazione concatenata, allocazione contigua). Motivare la necessità di utilizzare più di un livello di indice discutendo anche in quali casi significativi l'introduzione di più livelli di indice peggiora le prestazioni del file system rispetto ad altri metodi di allocazione.

5. Illustrare in che modo un'architettura RAID può migliorare il tempo di accesso ai dischi, discutendo sia gli aspetti generali del problema, sia in particolare questi due casi:
   a) un programma accede sequenzialmente ad un solo file
   b) più programmi accedono a parti diverse di uno stesso file durante lo stesso intervallo di tempo
   c) più programmi accedono a file differenti durnte lo stesso intervallo di tempo

Unix

1. Descrivere in dettaglio le operazioni eseguite dal sistema operativo Unix per creare un processo attraverso la funzione fork, illustrando come evolvono le strutture dati del sistema che riguardano i processi (tabella dei processi), e che cosa avviene in memoria centrale durante queste operazioni.

2. Illustrare la differenza tra un link "hard" e un link simbolico ("symlink") nei sistemi operativi Unix-like. Illustrare come la presenza di link hard e simbolici influenza l'operazione di cancellazione di un file.

3. Illustrare l'esecuzione dell'operazione di cancellazione di un file da una directory nell'ipotesi che il file system consenta l'esistenza di più hard link e symbolic link ad un file, come avviene in Unix. Che cosa succede se si cancella un link simbolico ad un file? e se si cancella un file cui punta un link simbolico?

4. Si considerino queste due situazioni:
   a: il file /usr/tizio/fileA è anche identificato attraverso il link *hard* /usr/tizio/fileB
   b: il file /usr/tizio/fileA è anche identificato attraverso il link simbolico /usr/tizio/fileB

Discutere che cosa succede ai file e ai loro link, in ciascuna delle due situazioni qui sopra, a seguito dell'esecuzione del comando
   1. mv /usr/tizio/fileB /usr/tizio/fileC oppure del comando
   2. mv /usr/tizio/fileA /usr/tizio/fileC

5. Con riferimento ad una struttura di directory di tipo gerarchico, tipica dei sistemi Unix, illustrare in che cosa consiste l'operazione di montaggio (*mount*) di un file system, che scopo ha, che effetto produce sulla struttura dei file system e qual è il suo impiego con i volumi rimovibili.

Parte II - Domande a risposte chiuse

1. Il descrittore di processo (*process control block*) include al suo interno:

(a) l'identificatore del processo e quello del processo padre;
(b) l'identificatore del processo ma non quello del processo padre;
(c) l'identità dell'evento di cui il processo è in attesa;
(d) una copia dello stato del processore al momento dell'ultimo prerilascio o sospensione del processo;
(e) le aree dati, codice e stack del processo;
(f) la sola area stack del processo, utilizzando puntatori alle page table per il resto;
(g) i puntatori alle page table che descrivono la memoria virtuale assegnata al processo;
(h) le variabili condivise con altri processi nelle sezioni critiche;
(i) nulla di quanto elencato sopra.

Risposte corrette: (a), (c), (d), (g).

Gli elementi tipici di un PCB possono essere raggruppati in:

1. identificatori del processo
2. informazioni sullo stato del processore (al momento dell'ultimo prerilascio o sospensione)
3. informazioni sul controllo del processo.

Segue direttamente che l'affermazione (d) è vera. Tra gli identificatori del processo abbiamo l'identificatore (ID) del processo, l'ID del processo parent e l'ID dell'utente. L'affermazione (a) è quindi corretta mentre la (b) è errata. Tra le informazioni di controllo del processo un sottoinsieme di queste descrivono lo stato e la schedulazione: stato del processo (ad esempio in esecuzione, pronto o in attesa), priorità nella schedulazione, informazioni relative alla schedulazione dipendenti dall'algoritmo di scheduling ed infine l'identità dell'evento di cui il processo è in attesa prima di passare nello stato di pronto. L'affermazione (c) è quindi corretta.
Altre informazioni incluse tra le informazioni di controllo riguardano: privilegi del processo, risorse utilizzate (ad esempio i file aperti), gestione della memoria (include ad esempio puntatori a segmenti e/o page table della memoria virtuale del processo). L'affermazione (g) è quindi vera.
Tra le informazioni di controllo del processo possono essere inoltre presenti puntatori per collegare il processo in particolari strutture e flag/messaggi per l'interprocess communication. Il PCB non contiene informazioni in merito alle variabili condivise con altri procesi nelle sezioni critiche; l'affermazione (h) è falsa. Il PCB, insieme al programma eseguito dal processo, i dati utenti e lo stack di sistema compone la cosiddetta immagine del processo. Aree dati, codice e stack non sono quindi contenute nel PCB: sono false le affermazioni (e) ed (f). Da quanto detto segue che l'affermazione (i) è falsa.

2. In un sistema time sharing un processo può essere interrotto in modo forzato prima della scadenza del suo quanto di tempo

(a) no, mai
(b) sì, se ci sono altri processi in stato di attesa
(c) solo se c'è un processo pronto con priorità più alta
(d) solo se c'è un processo pronto con priorità più bassa

Risposte corrette: (a) o (c) in fuzione dell'assenza o presenza di priorità.

In un sistema time sharing puro l'esecuzione viene ripartita a turno tra i processi della stessa priorità. In un sistema senza priorità è vera l'affermazione (a) e sono false le altre, ma un sistema senza prioità è una semplificazione didattica non realistica.

In un sistema con priorità i processi più prioritari hanno precedenza. Quindi se un processo a priorità p è in esecuzione e durante il suo quanto di tempo un processo di priorità $q > p$ entra nello stato di pronto (ad es. perché ha terminato un'attesa su evento esterno) il processo in esecuzione viene interrotto. E' perciò vera l'affermazione (c).

3. In un sistema operativo UNIX le chiamate di sistema della famiglia *exec* (execl(), execv(), etc. ...):

(a) sono gli unici meccanismi di creazione di nuovi processi;
(b) sono i principali meccanismi di creazione di nuovi processi;
(c) causano la terminazione del processo in corso e l'avvio di un nuovo processo;
(d) causano la sostituzione del processo in corso con uno nuovo;
(e) riportano come risultato il PID del nuovo processo;
(f) riportano come risultato il valore restituito dalla funziona main() dell'eseguibile specificato;
(g) nessuna delle affermazioni precedenti è corretta.

Risposta corretta: (g).

Le risposte (a) e (b) non sono corrette in quanto nel sistema operativo UNIX la creazione di un processo avviene con la chiamata di sistema *fork*(). La *fork*()crea una copia del processo chiamante (parent process). Restituisce al processo *parent* il PID del processo creato (*child process*) mentre restituisce 0 al processo *child*.
La chiamata ad una funzione della famiglia *exec* comporta la sostituzione dell'immagine del processo chiamante con l'eseguibile specificato dagli argomenti, lasciandone inalterato *pid* e processo *parent*. Il processo non viene quindi terminato: l'affermazione (c) è falsa.
Inoltre, essendo sostituiti solo alcuni elementi del processo, e non il processo stesso, anche la risposta (d) è falsa.
Se hanno successo le funzioni della famiglia *exec* non effettuano ritorno al chiamante (il codice chiamante non esiste più, essendo sostituito dal nuovo eseguibile). In caso di errore, cioè se il nuovo eseguibile non ha potuto sostituire il codice del chiamante) viene restituito il valore -1. Quindi anche le risposte (e) ed (f) sono errate. Segue che solo la risposta (g) è vera.

4. Dopo l'esecuzione di questo frammento di programma appartenente al codice del processo *P*, considerando anche l'esecuzione dei processi eventualmente creati:

```
i = fork();
j = fork();
```

(a) ci sono due processi
(b) ci sono tre processi
(c) ci sono quattro processi

Risposta corretta: (c).

Dopo l'esecuzione della prima istruzione i = fork () viene creato un secondo processo *P'* che esegue lo stesso codice del processo *P*. Entrambi i processi proseguono con l'esecuzione dell'istruzione j = fork(), perciò entrambi creano un nuovo processo. *P* crea un processo *P''* e *P'* crea un processo *P'''*. Se non si verificano errori, i processi alla fine dell'esecuzione del frammento di programma sono quattro.

Parte III - Esercizi

1. In un sistema time-sharing con politica di scheduling *round robin con priorità statiche* sono presenti quattro processi P1-P4 nel seguente stato:

P1 in esecuzione, P2 e P3 pronti, P4 in attesa di una operazione di I/O.

Le priorità dei processi sono in questa relazione:

P1 = P2 > P4 > P3

Descrivere, motivando la risposta, come cambia lo stato del sistema (cioè come cambia lo stato dei processi) se a partire dalla situazione data si verificano nell'ordine tutti e soli i seguenti eventi:

a) trascorre un quanto di tempo
b) termina l'operazione di I/O del processo in attesa
c) il processo in esecuzione chiede una operazione di I/0
d) trascorre un quanto di tempo
e) il processo in esecuzione chiede una operazione di I/0
f) trascorre un quanto di tempo
g) il processo in esecuzione termina

Soluzione

Il sistema evolve in questo modo:

a) Essendo P1 e P2 della stessa priorità, P2 va in esecuzione al posto di P1, che va in stato di pronto; P1 si posiziona nella coda dei processi pronto davanti a P3 perché ha priorità maggiore.
b) Il processo P4 va in stato di pronto, posizionandosi davanti a P3 perché meno prioritario. In esecuzione rimane il processo P2.
c) Il processo P2 dallo stato di esecuzione va in stato di attesa; P1 è il primo processo pronto per priorità e va in esecuzione.
d) Il processo P1 rimane in esecuzione perché gli altri processi pronti, P3 e P4, hanno priorità minore.
e) Il processo P1 dallo stato di esecuzione va in stato di attesa; nella coda dei processi pronti ci sono i processi P3 e P4; il più prioritario è il processo P4, che va in esecuzione.
f) Il processo P4 rimane in esecuzione perché l'altro processo pronto, P3, ha priorità minore.
g) Il processo P4 termina e va in esecuzione P3, unico processo nella coda dei processi pronti.

L'evoluzione del sistema è riassunta in questa tabella:

Evento	*Esecuzione*	*Pronto*	*Attesa*
Stato iniziale	P1	P2 > P3	P4
a	P2	P1 > P3	P4
b	P2	P1 > P4 > P3	–
c	P1	P4 > P3	P2
d	P1	P4 > P3	P2
e	P4	P3	P2, P1
f	P4	P3	P2, P1
g	P3	–	P2, P1

2. Quattro processi (da P1 a P4) entrano nello stato di pronto a distanza di 100 mS l'uno dall'altro e nell'ordine da P1 a P4. I processi hanno stessa priorità. Si supponga che il tempo stimato del prossimo CPU-burst di ogni processo sia rispettivamente di 600, 300, 500 e 200 mS. Dopo questo CPU burst inizia per ogni processo un'operazione di I/O molto lunga. Per le politiche di scheduling elencate di seguito, determinare il tempo di attesa per ciascun processo nello stato di pronto, (considerando solo il CPU burst) e il tempo di attesa medio dei quattro processi:

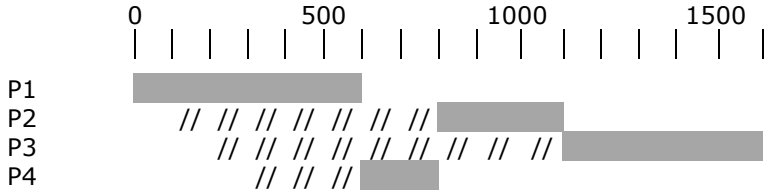
a. Shortest Process Next
b. Round-robin con un quanto di tempo di 400 mS
c. Round-robin con un quanto di tempo di 200 mS
d. First Come First Served

Soluzione

Negli schemi seguenti le parti in grigio rappresentano i processi in esecuzione, le parti tratteggiate rappresentano i processi nello stato di pronto.

a. *Shortet Process Next*: manda in esecuzione i processi dando precedenza a quelli con il CPU burst minore.

Il primo processo a partire è sempre il processo P1, anche se non è il più breve, poiché al tempo 0 gli altri processi non sono ancora entrati nello stato di pronto. Al tempo 600mS, quando P1 termina il suo CPU burst, gli altri processi sono tutti nello stato di pronto per cui verranno eseguiti nell'ordine dal più breve al più lungo:



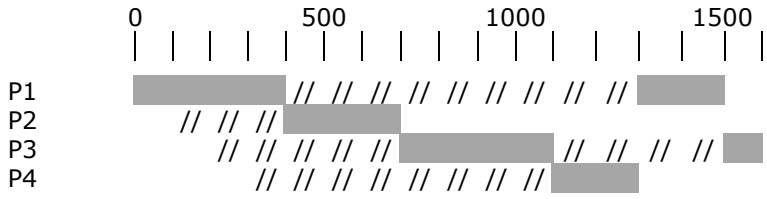
da cui i seguenti tempi di attesa nello stato di pronto:

P1 = 0
P2 = 700 mS
P3 = 900 mS
P4 = 300 mS

tempo di attesa medio = (0+700+900+300)mS/4 = 1900mS/4 = 475 mS

b. *Round-robin* con un quanto di tempo di 400 mS: alterna i processi in una coda circolare allo scadere del quanto di tempo.

Per il calcolo del tempo di attesa è necessario considerare tutti i periodi trascorsi nello stato di pronto, quindi anche quelli che intercorrono tra un quanto di tempo e l'altro. L'evoluzione del sistema è la seguente:



da cui i seguenti tempi di attesa nello stato di pronto:

P1 = 900 mS
P2 = 300 mS
P3 = 900 mS
P4 = 800 mS

tempo di attesa medio = (900+300+900+800)mS/4 = 2900mS/4 = 725 mS

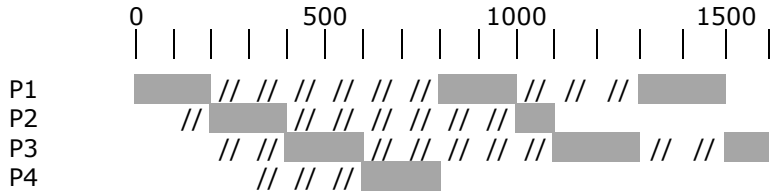
c. *Round-robin* con un quanto di tempo di 200 mS

In questo caso si deve osservare che il tempo di ingresso nella coda di pronto del processo P3 coincide con lo scadere del primo intervallo di tempo di 200 mS su cui si basa il meccanismo di round-robin. Nella realtà, i due eventi non saranno contemporanei, e il verificarsi dell'uno prima dell'altro modifica la situazione della coda dei processi pronti, e quindi l'evoluzione del sistema. Entrambe le evoluzioni sono considerate corrette ai fini della soluzione del problema.

Se il processo P3 entra nello stato di pronto prima che lo scheduler interrompa l'esecuzione del processo P1, portandolo nello stato di pronto, la situazione della coda dei processi pronti è la seguente:

--> | P1 | P3 | P2 | -->

Quindi andrà in esecuzione il processo P2, P3 diventerà il primo processo nella coda e sarà eseguito al terzo turno di round-robin. L'esecuzione complessiva sarà la seguente:



da cui i seguenti tempi di attesa nello stato di pronto:

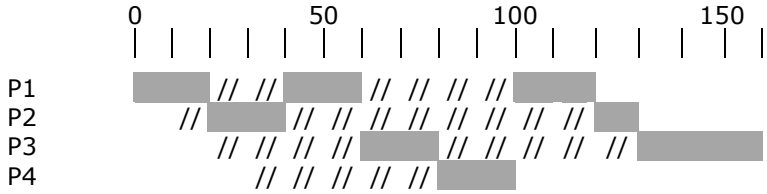
P1 = 900 mS
P2 = 700 mS
P3 = 900 mS
P4 = 300 mS

tempo di attesa medio = (900+700+900+300)mS/4 = 2800mS/4 = 700 mS

Se invece il processo P3 entra nella coda dei processi pronti dopo che lo scheduler ha interrotto il processo P1, la situazione della coda sarà la seguente:

--> | P3 | P1 | P2 | -->

Quindi andrà in esecuzione il processo P2, ma diventerà primo processo nella coda dei processi pronti il processo P1, interrotto e riportato nella coda un istante prima dell'ingresso in coda del processo P3. L'esecuzione complessiva sarà quindi la seguente:

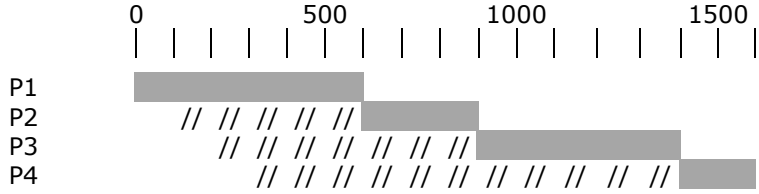


da cui i seguenti tempi di attesa nello stato di pronto:

P1 = 600 mS
P2 = 900 mS
P3 = 900 mS
P4 = 500 mS

tempo di attesa medio = (600+900+900+500)mS/4 = 2900mS/4 = 725 mS

d. *First Come First Served*: manda in esecuzione i processi nell'ordine in cui arrivano, per l'intera durata del loro CPU burst.



da cui i seguenti tempi di attesa nello stato di pronto:

P1 = 0
P2 = 500 mS
P3 = 700 mS
P4 = 1100 mS

tempo di attesa medio = (0+500+700+1100)mS/4 = 2300mS/4 = 575 mS

3. Si consideri un processo che fa rifermento a 5 pagine virtuali nel seguente ordine:

1, 2, 3, 2, 3, 4, 2, 3, 5, 1, 3, 5

Si consideri una memoria fisica inizialmente vuota di 3 pagine e si illustri il comportamento dei seguenti algoritmi di sostituzione delle pagine, evidenziando i page fault che vengono generati:

1. Ottimo
2. FIFO (First In First Out)
3. LRU (Least Recently Used)

La sequenza di sostituzione delle pagine è rappresentata su una tabella, le cui righe rappresentano le pagine, le colonne le successive configurazioni della tabella delle pagine a seguito dei successivi riferimenti alla memoria. Sotto la tabella sono riportati i casi in cui si genera un page fault. La tabella delle pagine è inizialmente vuota.

Algoritmo ottimo: sostituisce la pagina che richiederà meno page fault in futuro. Quando viene richiesta la pagina 4, viene sostituita la pagina 1 perché è quella che verrà riutilizzata più avanti nel tempo. Quando viene richiesta la pagina 5, sia la pagina 2 sia la pagina 4 non verranno più utilizzate, quindi entrambe sono candidate alla sostituzione. Si hanno perciò due soluzioni equivalenti:

	1	2	3	2	3	4	2	3	5	1	3	5
	1	1	1	1	1	4	4	4	5	5	5	5
		2	2	2	2	2	2	2	2	1	1	1
			3	3	3	3	3	3	3	3	3	3
	F	F	F			F			F	F		

	1	2	3	2	3	4	2	3	5	1	3	5
	1	1	1	1	1	4	4	4	4	1	1	1
		2	2	2	2	2	2	2	5	5	5	5
			3	3	3	3	3	3	3	3	3	3
	F	F	F			F			F	F		

FIFO: sostituisce la pagina che è stata caricata da più tempo. Quando viene richiesta la pagina 4 viene sostituita la pagina 1, successivamente vengono sostituite la pagina 2 e la pagina 3, che sono le pagine caricate per prime, in quest'ordine. Quando viene richiesta nuovamente la pagina 3 (penultimo passaggio) viene sostituita la pagina 4, presente in memoria da più tempo rispetto alle pagine 1 e 5.

	1	2	3	2	3	4	2	3	5	1	3	5
	1	1	1	1	1	4	4	4	4	4	3	3
		2	2	2	2	2	2	2	5	5	5	5
			3	3	3	3	3	3	3	1	1	1
	F	F	F			F			F	F	F	

LRU: sostituisce la pagina che è inutilizzata da più tempo. Quando viene richiesta la pagina 4 viene sostituita la pagina 1 che non è più stata utilizzata dopo il suo caricamento iniziale. Quando viene richiesta la pagina 5 viene sostituita la pagina 4, perchè le pagine 2 e 3 sono state utilizzate negli ultimi due accessi. Quando viene nuovamente richiesta la pagina 1, tra quelle presenti viene scaricata la pagina 2 perché non è utilizzata da più tempo rispetto alle pagine 3 e 5. La strategia LRU in questo caso coincide con uno dei due casi della strategia ottima.

	1	2	3	2	3	4	2	3	5	1	3	5
	1	1	1	1	1	4	4	4	5	5	5	5
		2	2	2	2	2	2	2	2	1	1	1
			3	3	3	3	3	3	3	3	3	3
	F	F	F			F			F	F		

4. Si consideri un processo che utilizza solo tre pagine fisiche (*frame*) di memoria, inizialmente vuote. Il processo accede a locazioni di memoria contenute in sei pagine virtuali, che vengono referenziate nel seguente ordine:

1, 2, 3, 2, 6, 4, 2, 1, 5, 3, 6, 5

Si illustri il comportamento degli algoritmi LRU (Least Recently Used) e FIFO (First In First Out) per la sostituzione delle pagine fisiche evidenziando i page fault che vengono generati e commentando la ragione della loro scelta.

Si confronti quindi il comportamento dei due algoritmi con un ipotetico algoritmo di sostituzione ottima della pagine fisiche.

Soluzione

La sequenza di sostituzione delle pagine è rappresentata su una tabella avete una riga per ogni frame, le cui colonne rappresentano le configurazioni della tabella delle pagine a seguito dei riferimenti alla memoria. Sotto la tabella sono riportati i casi in cui si genera un page fault. La tabella delle pagine è inizialmente vuota

LRU: sostituisce la pagina che è inutilizzata da più tempo. Quando viene richiesta la pagina 6 viene sostituita la pagina 1 che non è più stata utilizzata dopo il suo caricamento iniziale. Quando viene richiesta la pagina 4 viene sostituita la pagina 3, perché le pagine 2 e 6 sono state utilizzate negli ultimi due accessi. Quando viene nuovamente richiesta la pagina 1, tra quelle presenti viene scaricata la pagina 6 perché non è utilizzata da più tempo rispetto alle pagine 2 e 4. E così via. Il numero di page fault è elevato anche se la strategia LRU è ritenuta molto efficace perché la sequenza degli accessi non segue una regolarità che ci si attenderebbe in base al principio di località, e perché il numero di pagine logiche accedute è elevato rispetto a l numero di pagine fisiche disponibili.

Pagina	1	2	3	2	6	4	2	1	5	3	6	5
Frame												
1	1	1	1	1	6	6	6	1	1	1	6	6
2		2	2	2	2	2	2	2	2	3	3	3
3			3	3	3	4	4	4	5	5	5	5
	F	F	F		F	F		F	F	F	F	

FIFO: sostituisce la pagina che è stata caricata da più tempo. Quando viene richiesta la pagina 6 viene sostituita la pagina 1, prima nell'ordine di caricamento. Successivamente per caricare le pagine 6 e 4 vengono sostituite le pagina 2 e 3, in quest'ordine, perché sono le pagine caricate da più tempo. Quando viene richiesta nuovamente la pagina 2 viene generato un ulteriore page fault rispetto all'algoritmo FIFO.

Pagina	1	2	3	2	6	4	2	1	5	3	6	5
Frame												
1	1	1	1	1	6	6	6	1	1	1	6	6
2		2	2	2	2	4	4	4	5	5	5	5
3			3	3	3	3	2	2	2	3	3	3
	F	F	F		F	F	F	F	F	F	F	

Algoritmo ottimo: è un algoritmo teorico di riferimento, che sostituisce la pagina che sarà utilizzata più avanti nel tempo, posticipando quindi il momento di un successivo page fault per richiamarla. Quando viene richiesta la pagina 6, viene sostituita la pagina 3 perché sarà riutilizzata solo dopo numerosi passi, mentre le pagine 1 e 2 saranno utilizzate nuovamente a breve termine. Quando viene richiesta la pagina 4, per lo stesso motivo viene scaricata la pagina 6. Quando viene richiesta la pagina 5, le pagine presenti in memoria non saranno più utilizzate in futuro quindi ognuna di esse può essere scelta come pagina vittima. Nella soluzione scegliamo di seguire l'ordine sequenziale per numero di pagina fisica.

Pagina	1	2	3	2	6	4	2	1	5	3	6	5
Frame												
1	1	1	1	1	1	1	1	1	5	5	5	5
2		2	2	2	2	2	2	2	2	3	3	3
3			3	3	6	4	4	4	4	4	6	6
	F	F	F		F	F			F	F	F	

5. Un processo in esecuzione utilizza quattro pagine virtuali e tre pagine fisiche di 8 Kbyte ciascuna. La tabella delle pagine è la seguente, dove tutte le numerazioni sono decimali, i tempi sono espressi in unità convenzionali ed esprimono il tempo assoluto del caricamento o dell'ultimo accesso alla pagina fisica:

# pagina virtuale	*valida*	*usata*	*modificata*	*# pagina fisica*	*tempo del caricamento*	*tempo dell' ultimo accesso*
0	1	1	0	12	73	76
1	0	0	0	–	–	–
2	1	1	0	3	60	90
3	1	1	1	5	70	70

Per la sequenza di accesso in lettura ai due indirizzi virtuali: 9.118 e 20.096, nell'ordine, dire, per gli algoritmi di sostituzione di pagina FIFO e LRU:

- se l'accesso genera o no un page fault;
- se sì, qual è la pagina vittima individuata, e se deve essere scritta su disco o no.
- qual è l'indirizzo fisico corrispondente, eventualmente dopo la risoluzione del page fault.

Soluzione

I due indirizzi virtuali sono allocati così: 9.118, pagina 1, offset 926; 20.096, pagina 2 offset 3.712

Strategia FIFO

L'accesso all'indirizzo 9.118 causa un page fault perché la pagina virtuale 1 non è valida. Viene scelta come pagina vittima la pagina fisica caricata da più tempo, cioè la pagina fisica 3 (tempo assoluto di caricamento = 60). Non essendo modificata non deve essere scritta su disco. L'indirizzo fisico corrispondente è (3 x 8.192) + 926 = 25.502.

La tabella delle pagine si modifica in questo modo (X è il tempo in cui avviene la sostituzione, X > 90):

# pagina virtuale	*valida*	*usata*	*modificata*	*# pagina fisica*	*tempo del caricamento*	*tempo dell' ultimo accesso*
0	1	1	0	12	73	76
1	1	1	0	3	X	X
2	0	0	0	–	–	–
3	1	1	1	5	70	70

Il successivo accesso all'indirizzo 20.096 causa un page fault perché la pagina virtuale 2 non è più valida. Viene scelta come pagina vittima la pagina fisica caricata da più tempo, cioè la pagina fisica 5 (tempo assoluto di caricamento = 70). Essendo modificata, deve essere scritta su disco. L'indirizzo fisico corrispondente è (5 x 8192) + 3.712 = 44.672.

La tabella delle pagine si modifica in questo modo (Y è il tempo in cui avviene la sostituzione, Y > X)

# pagina virtuale	*valida*	*usata*	*modificata*	*# pagina fisica*	*tempo del caricamento*	*tempo dell' ultimo accesso*
0	1	1	0	12	73	76
1	1	1	0	3	X	X
2	1	1	0	5	Y	Y
3	0	0	0	–	–	–

Strategia LRU

COme sopra, l'accesso all'indirizzo 9.118 causa un page fault. Viene scelta come pagina vittima la pagina fisica non utilizzata da più tempo, cioè la pagina fisica 5 (tempo assoluto di ultimo accesso = 70), che deve essere scritta su disco. L'indirizzo fisico corrispondente è (5 x 8.192) + 926 = 41.886.

La tabella delle pagine si modifica in questo modo (X è il tempo in cui avviene la sostituzione, X > 90):

# pagina virtuale	*valida*	*usata*	*modificata*	*# pagina fisica*	*tempo del caricamento*	*tempo dell' ultimo accesso*
0	1	1	0	12	73	76
1	1	1	0	5	X	X
2	1	1	0	3	60	90
3	–	–	–	–	–	–

Il successivo accesso all'indirizzo 20.096 non causa un page fault perché la pagina virtuale 2 è valida. L'indirizzo fisico corrispondente è (3 x 8192) + 3.712 = 28.288.

6. Si consideri la struttura di indicizzazione del file system di Unix, con dimensione del cluster di allocazione su disco di 4096 byte e puntatori di 4 byte. Dato un file di 512 Mbyte (1 Mbyte = 2^{20} byte), rispondere alle seguenti domande, motivando la risposta:

a) quanti blocchi di indice, oltre all'i-node, sono necessari per indicizzare il file;
b) quanti accessi a disco sono necessari per leggere con accesso diretto il primo e l'ultimo byte del file, supponendo che il file sia già aperto e quindi il suo i-node sia già in memoria centrale.

Soluzione

4096 / 4 = 1024 puntatori per ogni blocco indice.

Dimensione file: 512 M / 4096 = 128 K blocchi di dati, richiedono 128 K (128 * 1024) indici così organizzati, oltre all'i-node che indicizza direttamente i primi 12 blocchi:

- un blocco indice di primo livello per gli indici dei blocchi da 13 a 1024+12 = 1036
- un blocco indice di primo livello per indirizzare gli indici di secondo livello
- 127 blocchi indice di secondo livello per i blocchi dati da 1037 a 128 K; l'ultimo blocco indice è occupato da 1024 - 12 indici.

Complessivamente servono 129 blocchi di indice.

Per leggere il primo byte serve solo un accesso a disco per leggere il primo blocco dati dal momento che il suo indice è contenuto nell'i-node; per leggere l'ultimo byte sono necessari 3 accessi a disco: l'indice di 1° livello, l'indice di 2° livello e il blocco dei dati.

7. Si consideri l'esecuzione di questo programma nel sistema operativo Unix, supponendo che il processo che lo esegue abbia *process-id* uguale a 1001 e che i processi creati successivamente abbiano *process-id* consecutivi e crescenti:

```
main()
{ int i, j;
  i = fork();  j = fork();
  if (i == j)
      visualizza sul terminale "Caso i = j: i, j = " seguito dal valore di i;
  else if (i > j)
      visualizza sul terminale "Caso i > j: i, j = " seguito dai valori di i e j;
  else
      visualizza sul terminale "Caso i < j: i, j = " seguito dai valori di i e j;
}
```

Illustrare come evolve il sistema durante l'esecuzione: quali processi sono creati, che *process-id* hanno, che gerarchie padre-figlio si stabiliscono e che cosa viene visualizzato sul terminale da parte di quale processo. Si assuma che non si verificano errori.

Nota. In genere l'ordine di esecuzione dei processi generati dalla system call *fork* non è definito a priori: si ipotizzi un ordine di esecuzione qualunque, purché plausibile, e si proceda nell'analisi dell'esecuzione in modo coerente.

Soluzione

Si identifichino i processi con i nomi A, B, etc., e sia A il processo iniziale. Si assume che l'ordine di esecuzione dei processi corrisponda con l'ordine dei *process-id*. Dopo l'istruzione i = fork() si hanno i seguenti processi:

A (processo iniziale):	process-id = 1001, i = 1002, j = indefinito
B (creato da A):	process-id = 1002, i = 0, j = indefinito

Dopo l'istruzione j = fork(), che viene eseguita da entrambi i processi, la situazione è la seguente:

A (processo iniziale):	process-id = 1001, i = 1002, j = 1003
B (creato da A):	process-id = 1002, i = 0, j = 1004
C (creato da A):	process-id = 1003, i = 1002, j = 0
D (creato da B):	process-id = 1004, i = 0, j = 0

Sul terminale vengono quindi visualizzate le seguenti informazioni:

processo A:	Caso i < j: i, j = 1002, 1003
processo B:	Caso i < j: i, j = 0, 1004
processo C:	Caso i > j: i, j = 1002, 0
processo D:	Caso i = j; i, j = 0