Esame 2018/09/10

Blascovich Alessio

Domanda 1

MAX PUNTI: 4
Domanda:
Spiega in dettaglio come funziona il meccanismo della Tabella delle Pagine Invertita.
Soluzione:
Si basa su una tabella unica per tutto il sistema, quindi non per i singoli processi.
Questa tabella contiene una tupla con <pid,numero-pagina> con:

- PID: è l'identificativo del processo che detiene la pagina.
- numero-pagina: l'indirizzo logico della pagina

Per cercare un indirizzo bisogna quindi scorrere tutta la tabella per trovare la tupla che contiene la combinazione pid e numero-pagina cercata. Gli indirizzi vengono tradotti cercando la tupla nella tabella e tenendo conto in una variabile `i` di che posizione occupa nella tabella la nostra tupla (simile a dire "a che indice `i` sta la nostra tupla").
Viene poi fatto un append tra `i` e l'offset `d`, l'indirizzo ottenuto è l'indirizzo fisico cercato.

Domanda 2

MAX PUNTI: 4
Domanda:
Descrivere le differenze tra semafori normali e spinlock, fornire poi un esempio di utilizzo di semafori spin-lock da parte del kernel.
Soluzione:
I semafori implementati con spinlock(busy waiting) delegano la CPU ad un continuo controllo dello stato della sezione critica.
Questo li porta ad essere molto CPU-intensive e adatti a contesti dove c'è bisogno di una risposta immediata, come nel caso di accessi alla memoria.
Sono moto facili da implementare ed anche molto scalabili.
I semafori sono usate dal kernerl per sincronizzare tra di loro processi/thread.

Domanda 3

MAX PUNTI: 5
Domanda:
Data la stringa di riferimenti alla memoria 1, 2, 3, 4, 1, 2, 5, 1, 2,3, 4, 5, si determinino il numero di page fault che si avranno usando gli algoritmi FIFO, LRU e ideale.
Si supponga di avere inizialmente 3 frame vuoti.
Soluzione:
Si consideri la sigla pf come l'acronimo di page-fault.

- *FIFO*:
  Il metodo *FIFO* gestisce la memeoria come una semplice coda, il primo segmaneto ad essere entrato è il primo ad essere fatto uscire.

1	null	null	pf=1 ⟹	1	2	null	pf=2 ⟹	1	2	3	pf=3 ⟹	4	2	3	pf=4 ⟹	4	1	3	pf=5 ⟹	4	1	2	pf=6 ⟹	5	1	2
pf=6 ⟹	5	1	2	pf=6 ⟹	5	1	2	pf=6 ⟹	5	1	2	pf=6 ⟹	3	1	2	pf=7 ⟹	3	4	2	pf=8 ⟹	3	4	5	pf=9 ⟹		

  Termino con un totale di 9 page fault.

- *LRU*:
  L'algoritmo *LRU* tiene associato ad ogni frame il clock della CPU nell'istante in cui quel frame è stato usato per l'ultima volta, verrà tolto il frame che non viene usato da più tempo.
  Nella prima riga sono indicati gli istanti, mentre nella seconda il numero del frame.

1	0	0
1	null	null

pf=1 ⟹

1	2	0
1	2	null

pf=2 ⟹

1	2	3
1	2	3

pf=3 ⟹

4	2	3
4	2	3

pf=4 ⟹

4	5	3
4	1	3

pf=5 ⟹

4	5	6
4	1	2

pf=6 ⟹

7	5	6
5	1	2

pf=7 ⟹

7	8	6
5	1	2

pf=7 ⟹

7	8	9
5	1	2

pf=7 ⟹

10	8	9
3	1	2

pf=8 ⟹

10	11	9
3	4	2

pf=9 ⟹

10	11	12
3	4	5

pf=10 ⟹ Termino con un totale di 10 page fault.

- *Ideale*:
  Come dice il nome questo è un algoritmi perfetto, perchè dovrebbe prevedere il futuro, infatti dovrebbe sapere quali saranno le prossime richieste che un pogramma potrebbe fare.

1	null	null

pf=1 ⟹

1	2	null

pf=2 ⟹

1	2	3

pf=3 ⟹

1	2	4

pf=4 ⟹

1	2	4

pf=4 ⟹

1	2	4

pf=4 ⟹

1	2	5

pf=5 ⟹

1	2	5

pf=5 ⟹

1	2	5

pf=5 ⟹

3	2	5

pf=6 ⟹

3	4	5

pf=7 ⟹

3	4	5

pf=7 ⟹ Termino con un totale di 7 page fault

Domanda 4

MAX PUNTI: 6
Domanda:
Data la seguente tabella contenente un insieme di processi:

Processo	*Burst*	*Tempo di arrivo*
1	3	0
2	1	1
3	2	3
4	4	4
5	8	1

Disegnare lo schema di arrivo dei processi senco gli algoritmi HRRN e RR con quanti di tempo pari a 2.
Per quanto riguarda il RR si immagini che i processi vengano inseriti nella *ready queue* in un tempo tale per cui il tempo di risposta è minimo.
Calcolare tempo di attesa, risposta e turnaround per ogni processo.
Soluzione:

- *HRRN*: E' un algoritmo a priorità non-preempitive, la priorità viene calcolata come R=1+$\frac{T_{attesa}}{T_{burst}}$, questo valore va ricalcolato ogni qual volta un processo termina la propria esecuzione oppure quando un processo termina e nel mentre sono arrivati altri processi.

Processo	t=0	t=3	t=4	t=6	t=14	t=18
1	$1+\frac{0}{3}=1$					
2		$1+\frac{2}{1}=3$				
3		$1+\frac{0}{2}=1$	$1+\frac{1}{2}=\frac{3}{2}$			
4			$1+\frac{0}{4}=1$	$1+\frac{2}{4}=\frac{3}{2}$	$1+\frac{10}{4}=\frac{7}{2}$	fine
5		$1+\frac{2}{8}=\frac{5}{4}$	$1+\frac{3}{8}=\frac{11}{8}$	$1+\frac{5}{8}=\frac{13}{8}$		

  Ora dobbiamo calcolare i tempi, definiti come:

  - T_{attesa}: Tempo in cui il processo è stato nella coda dei processi pronti prima di essere eseguito la prima volta.
  - $T_{risposta}$: Tempo totale in cui un processo è stato nella coda dei processi pronti.
  - $T_{turnaround}$: Tempo tra la sottoposizione del processo e la sua comclusione, può essere espresso anche come $T_{risposta} + burst$.

Processo	T_{attesa}	$T_{risposta}$	$T_{turnaround}$
1	0	0	3
2	2	2	3
3	1	1	3
4	10	10	14
5	5	5	13

- *RR(Round Robin)*: Algoritmo basato su un time-out.
  Viene dichiarato un quanto di tempo, durante questo quanto un processo può usare la CPU ma allo scadere del quanto il processo deve tornare nella coda dei processi.
  In partica è un *FCFS* preempitive, perchè la coda viene popolata in base all'ordine di arrivo.

Tabella della coda

Il colore rosso indica quando un processo nel quanto di tempo, oppure prima termina la propria esecuzione

Exec	1	2	5	1	3	4	5	4	5
Queue	2	5	1	3	4	5	4	5	
	5	1	3	4	5				
		4	5						

Tabella dell'esecuzione dei processi

Il colore rosso indica quando un processo detiene la CPU.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1																		
2																		
3																		
4																		
5																		

Tabella dei tempi

Processo	T_{attesa}	$T_{risposta}$	$T_{turnaround}$
1	0	3	6
2	1	1	2
3	3	3	5
4	4	6	10
5	2	9	17

Domanda 5

MAX PUNTI: 7

Domanda:

Si descriva i processi che portano da programma a processo, quindi si spiegino il concetto di binding e si enuncuino le differenti tipologie di linking e loading.

Soluzione:

Il binding è la traduzione che viene fatta da indirizzi simbolici usati nel programma in indirizzi fisici.
Il binding può essere fatto in tre moenti diverse:

- *Compile time*: Bisogna conoscere in precedenza la locazione di memoria che il programma andrà ad occupare, se la locazione cambia bisogna ricompilare.
- *Load time*: Genero gli indirizzi in base a dove inizia il programma, quindi posso anche riposizionare il processo, se però cambia l'indirizzo di riferimento devo ricompilare.
- *Run time*: Il binding è posticipatro così da permettere di poter spostare il processo durante l'esecuzione, per motivi di efficienza questo binding richiede supporto hardware.

Il linking può essere fatto in due modi diversi:

- *Statico*: Tutti i riferimenti sono definiti prima dell'esecuzione, e il processo dentro di se contine una copia di tutte le librerie usate.
- *Dinamico*: Il codice del processo non contine il codice delle librerie, ma contiene un puntatore(stub) alla funzione della libreria chiamata.
  Un esempio sono le DLL(*Dynamic Linked Library*) usate in Windows.

Il loading può avvenire in due modi diversi:

- *Statico*: L'intero processo viene caricato in memoria.
- *Dinamico*: Il processo è diviso in moduli che vengono caricati all'evenienza, tecnica usata in casi molto particolari.

Domanda 6

MAX PUNTI: 7
Domanda:

Considerando il seguente codice:

```
Risorse condivise
semaphore S=1, T=1, U=0;
int x=0;

Processo P1{
    down(&S);
    if (x=0) then up(&T)
    else up(&U);
    x:=3;
    write(x);
}

Processo P2{
    down(&T);
    x:=1;
    up(&S);
}

Processo P3{
    down(&U);
    x:=10;
    up(&S);
}
```

Si supponga che i processi siano eseguiti in modo concorrente sulla stessa CPU .

a Possono verificarsi race conditions su `x`?

b Quali sono gli output del programma?

c Cosa succede se inserisco `down(&S)` nel processo `P1` subito prima di `write(x)`?
In modo tale che sia:

```
...
x:=3;
down(&S);
write(x);
...
```

Soluzione:

a Si, perchè `P1` e `P2` possono accedere contemporaneamente ad `x`.

b Per via delle race conditions il comportamento non è sempre uguale:

- `P1`, `P2` e `P3` si avviano ma P3 si blocca e aspetta il semaforo `U` allora `P2` viene seguito tutto e risulta `x=1`, ora `P1.else` setta `U=1` così `P3` può continuare , nel mentre `P1` ha assegnato `x=3`, ma il valore viene sovrascritto da `P3` con `x=10` quindi viene stampato 10.
- Partendo dal caso precedente se `P1` è l'ultimo a modificare `x` allora verrà stampato 3.
- Può succedere che le istruzioni per la modifica dei semafori siano lente quindi viene eseguito tutto `P1` e stampa 3.
- Può succedere addirittura che `P3` non venga eseguito in tempo, così da eseguire prima `P1` poi prima del `write` viene seguito `P2` che modifica `x=1` così verrà stampato 1.

c Se il codice fosse stato quello il `write` sarebe stato eseguito dopo che `P2` e `P3` hanno eseguito un up nei rispettivi semafori.
Però rimangono le race conditions su `x` quindi gli output rimangono 1, 3 e 10.