Sistemi Operativi
Compito Advanced 21 settembre 2005

1. (a) Quali tipi di scheduler sono presenti in un sistema? Si tracci il diagramma di migrazione dei processi tra le code dei dispositivi, indicando dove agiscono gli scheduler.

   

   (b) Cos'è la starvation? Con quali algoritmi di scheduling della CPU si può verificare?

   

2. Si consideri un sistema hard real-time con scheduler EDF (Earliest Deadline First). All'istante 0 sono presenti in coda ready i processi periodici A con periodo $T_A = 30$ msec e durata di ogni CPU burst di 10 msec, e B con periodo $T_B = 40$ msec e durata di CPU burst pari a 25 msec.

   (a) Fornire il diagramma di GANTT relativo alle esecuzioni dei due processi nei primi 150 msec (ignorando il tempo di latenza del kernel).

   (b) All'istante 150 msec arriva un nuovo processo C con periodo $T_C = 40$ msec e CPU burst di 5 msec. Che cosa succede?

3. Nel reparto verdure di un noto ipermercato udinese, i clienti portano a pesare i sacchetti presso 4 bilance etichettatrici. Ogni sacchetto viene appoggiato dal cliente su una delle bilance; un addetto identifica il prodotto, stampa l'etichetta e l'attacca sul sacchetto.

   1. Modellate questa situazione con i thread di Solaris: chi è il processore? chi sono i task? cosa sono i thread a livello utente? cosa sono i thread a livello kernel?
   2. In questa situazione, cosa può fare (e spesso fa) un cliente per diminuire il tempo speso per pesare tutti i suoi sacchetti? A cosa equivale, in termini di programmazione?

   1. Addetto = processore. Bilance = thread a livello kernel = LWP. Clienti = task. Singoli sacchetti = thread a livello utente.
   2. Pone i sacchetti su due o più bilance. Questo equivale ad aumentare il grado di multiprogrammazione fisico del task, ossia aumentare il numero di LWP associati al task. Di conseguenza, aumenta anche la quota di CPU assegnata dal S.O. al task nel suo complesso, e quindi il tempo di turnaround diminuisce.

4. Un ascensore viene realizzato (oltre alle parti meccaniche) mediante una scheda a processore sulla quale viene installato un semplice sistema operativo che fornisce multiprogrammazione e semafori.

   Il sistema di funzionamento è strutturato a processi. I motori vengono comandati dal processo `motors_controller` mentre la gestione dei tasti nell'ascensore è affidata al processo `elevator_keypad`. Inoltre ad ogni piano ci sono dei sensori di posizione che vanno ad attivare una apposita routine di interrupt `floor_sensor_int_service` che aggiorna la variabile globale `floor`.

   Si completi il seguente codice concorrente usando opportunamente le primitive `up(&semaph)` e `down(&semaph)`.

```
int floor = 0;

[dich. e inizializz. semafori/o]

void motors_controller(void) {
while (TRUE) {
    [ sez1 ]  // waits for some action
    newfloor = get_action();
    while( newfloor != floor ) {
        if ( newfloor > floor ) activate_motors(UP);
        else activate_motors(DOWN);
        [ sez2 ]  // waits for next floor
        }
    stop_motors();
```

```
    [ sez3 ]
    }
}

void floor_sensor_int_service(void) {
    floor = get_floor();
    [ sez4 ]  // signals new floor
}

void elevator_keypad(void) {
while (TRUE) {
    [ sez5 ]  // waits motors controller is idle
    put_action(read_floor_key());  // read_floor_key is blocking
    [ sez6 ]
    }
}
```

```
int floor = 0;

semaph wake = 0, sleep = 1, floor_sensor=0;

void motors_controller(void) {
while (TRUE) {
    down(&wake); // waits for some action
    newfloor = get_action();
    while( newfloor != floor ) {
        if ( newfloor > floor ) activate_motors(UP);
        else activate_motors(DOWN);
        down(&floor_sensor);  // waits for next floor
        }
    stop_motors();
    up(&sleep);
    }
}

void floor_sensor_int_service(void) {
    floor = get_floor();
    up(&floor_sensor);  // signals new floor
}

void elevator_keypad(void) {
while (TRUE) {
    down(&sleep);  // waits motors controller is idle
    put_action(read_floor_key());  // read_floor_key is blocking
    up(&wake);
    }
}
```

5. Si consideri un sistema ove sia in funzione un algoritmo di deadlock detection munito anche di un sistema automatico di "soluzione" che rimanda in esecuzione un opportuno processo coinvolto nel deadlock dopo avergli tolto tutte le risorse. Il meccanismo di gestione del sistema è in grado di fornire statistiche sull'overhead medio di sistema p_O (espresso in percentuale al giorno) dovuto al suo intervento, nonché del tempo pagato in termini di riesecuzione del processo sacrificato p_R e della frequenza di deadlocks p_D.

   Si vuole valutare se sia il caso di rimuovere il sistema di deadlock solution (per non pagarne l'overhead) stabilendo in funzione dei precedenti parametri quanto debba essere (mediamente) il limite

del tempo α (espresso in percentuale di giorno) affinché i tecnici si accorgano della presenza di un deadlock, lo risolvano manualmente e il sistema riesca poi a recuperare il tempo perso a causa di detto intervento.

Il sistema produce un lavoro utile di $1 - p_O - p_R$ ogni giorno. La percentuale di tempo sprecata senza di lui sarebbe αp_D, quindi il lavoro utile sarebbe $1 - \alpha p_D$ da cui si ricava $\alpha < \frac{p_O + p_R}{p_D}$

6. Il meccanismo di file servicing NFS della Sun è Stateful (a stato) o Stateless (senza stato)?

   Tale scelta compiuta dai progettisti è giustificata in termini di performance o dovuta ad altri fattori?

   Se è stata compiuta la scelta più performante, a quale altre caratteristiche si è rinunciato? Se è stata compiuta la scelta meno performante, quali accorgimenti sono stati adottati in modo da alleviare il problema?

   Il protocollo NFS fornisce controlli di coerenza? In caso non lo faccia come vengono fornite queste funzionalità?

   

7. Si consideri un sistema di paginazione a 3 livelli con pagine da 4K e 12 bit per ogni livello (quindi indirizzamento logico a 48bit) dove ogni entry nella page table occupi 32 bit, tanti quanti i bits dell'indirizzamento fisico.

   (a) Supponiamo che un processo acceda alle seguenti pagine del suo spazio virtuale: 0x780E20, 0x780F22, 0xFFFFF03FF e 0xFFFFFFFFF. Quanti frame devono essere allocati al processo per contenere queste pagine e quanto serve al sistema di paginazione?

   Le pagine della tabella hanno una dimensione di $32b * 2^{12} = 16KB = 4$ frames. In ogni frame ci sono 1024 entries della page table.

   Visto che i primi 12 bits dei 4 indirizzi sono tutti uguali (a 0) avrò 1 solo frame della pagina di primo livello (quello che contiene lo 0).

   I primi 2 indirizzi e gli altri 2 hanno i secondi 12 bits rispettivamente uguali fra loro, ma cadono in due parti diverse della pagina di primo livello (frame 0 e frame F) quindi avremo bisogno di 2 frames di detta pagina. Avrò inoltre 2 pagine di terzo livello.

   Per entrambe queste 2 pagine mi basta 1 frame ciascuna (frame 0 per l'entry 780 in un caso e frame F per le entries FF0 e FFF nell'altro).

   Per i primi 2 indirizzi, che hanno i terzi 12 bits uguali, avrò 1 frame per contenere i dati indirizzati.

   Per gli altri due, che hanno i terzi 12 bits diversi, avrò 2 frames per contenere i dati indirizzati.

   In totale 8 frames.

   (b) Si valuti la differenza di performance sull'accesso medio in memoria (EAT) fra questo sistema e uno con tabella delle pagine invertita. Si assuma un tempo di accesso in cache t_c, hit ratio α, accesso in RAM t_m, un'hash per la tabella invertita che mediamente trova il riferimento in 1.5 passi, page fault rate p e tempo di recupero pagina da disco t_d. Si assuma inoltre nel caso a livelli di avere una località da far mantenere in memoria principale tutti i primi 2 livelli della tabella che servono.

   pagine invertite Assumiamo (come accade di solito nel caso della tabella invertita) di avere la tabella tutta in memoria, quindi:

   - ad ogni accesso attendiamo che la cache tenti di risolvere l'indirizzo;
   - se va male, con frequenza $1 - \alpha$, dobbiamo accedere alla tabella per la ricerca e il tempo medio di ricerca in tabella è $t_r = 3t_m$ (2 accessi per passo);

- ora, se va male, con frequenza p dobbiamo accedere al disco;
- poi (in ogni caso) facciamo l'accesso in RAM.

Quindi $t_{EAT} = t_c + (1-\alpha)(t_r + pt_d) + t_m = t_c + (1 + 3(1-\alpha))t_m + p(1-\alpha)t_d$.

4 livelli Viste le dimensioni della tabella *non* possiamo assumere venga mantenuta *tutta* in memoria e le ipotesi dell'esercizio specificano che troverò direttamente in RAM le sole pagine che servono per i primi 2 livelli, quindi:

- ad ogni accesso attendiamo che la cache tenti di risolvere l'indirizzo;
- se va male, con frequenza $1-\alpha$, dobbiamo accedere alla tabella di primo livello per la ricerca, che sta in RAM;
- analogamente per il secondo livello;
- per il terzo livello non abbiamo la pagina necessariamente in RAM per cui con frequenza p dobbiamo prima fare l'accesso a disco per la pagina e, poi, in ogni caso andiamo a fare l'accesso in RAM.
- a questo punto con frequenza p dobbiamo fare l'accesso a disco per il frame coi dati
- infine (in ogni caso) facciamo l'accesso in RAM.

Quindi $t_{EAT} = t_c+(1-\alpha)(2t_m+pt_d+t_m+pt_d)+t_m = t_c+(1+3(1-\alpha))t_m+2p(1-\alpha)t_d$.
La differenza è quindi $p(1-\alpha)t_d$.

8. Si consideri un disco gestito con politica SSTF. Inizialmente, la testina è posizionata sul cilindro 50; lo spostamento ad una traccia adiacente richiede 1ms. Al driver di tale disco arrivano richieste per i cilindri 0, 63, 20, 30, rispettivamente agli istanti 0ms, 17ms, 24ms, 33ms. Si trascuri il tempo di latenza.
   (a) In quale ordine vengono servite le richieste?

   All'istante 0, la testina inizia a muoversi alla velocità di 1 traccia/ms verso il cilindro 0. Dopo 17ms, quando arriva la richiesta per il cilindro 63, la testina si trova sul cilindro 33 e quindi questa richiesta prende il soppravvento sull'altra, che viene accodata. All'istante 24, quando arriva la richiesta per il cilindro 20, la testina si trova sul cilindro 40 e quindi questa richiesta prende il soppravvento sull'altra, che viene accodata. All'istante 33, quando arriva la richiesta per il cilindro 30, la testina si trova sul cilindro 31 e quindi questa richiesta prende il soppravvento sull'altra, che viene accodata. All'istante 34 la richiesta del 30 viene evasa e si riprende la 20. All'istante 44 la 20 viene evasa e si riprende la 0. All'istante 64 la 0 viene evasa e si riprende la 63. All'istante 127 si evade, in fine, la 63. Quindi l'ordine è: 30, 20, 0, 63.

   (b) Qual è il tempo di attesa medio per le quattro richieste in oggetto?

   La media è $\frac{(64-0)+(127-17)+(44-24)+(34-33)}{4} = 48.75$ ms.

9. Si consideri un disco con velocità di rotazione pari a v_r rpm, tempo medio di seek t_s, con 1.5MB in ogni traccia, sul quale si impiega un File System EXT2 con blocchi da 4KB e puntatori a 4B.

   Quant'è il tempo per la lettura sequenziale completa di un file di 3.5MB allocato in modo contiguo per quanto possibile?

   Il file sarà messo su 3 traccie, due parti da 1.5MB (quindi $\frac{1.5 \cdot 2^{20} B}{4 \cdot 2^{10}} = 384$ blocchi) e l'altra da 512KB (128 blocchi).

   Sia t_l il tempo di latenza medio e t_b il tempo di lettura di un blocco.

   Iniziamo a leggere l'inode (che entra in cache) impiegando $t_s + t_l + t_b$.

Quindi andiamo a leggere i primi 10 blocchi diretti (che sono allocati sequenzialmente) impiegando $10t_b$. Ora dobbiamo andare a leggere il blocco del primo livello indiretto (che entra in cache) impiegando $t_s + t_l + t_b$. Visto che questo blocco contiene 1024 puntatori è abbastanza grande per tutto il resto del file. Andiamo quindi a leggere gli altri 374 blocchi della prima traccia contigua impiegando $t_s + t_l + 374t_b$. Ci portiamo quindi sull'altro troncone del file e lo finiamo di leggere impiegando $t_s + t_l + 384t_b$. Ci portiamo infine sull'ultimo troncone del file e lo finiamo di leggere impiegando $t_s + t_l + 128t_b$.

Totale $t = 5t_s + 5t_l + 898t_b$.

10. (a) Si illustrino le problematiche che si dovrebbe affrontare per sviluppare un sistema operativo su un multiprocessore NUMA. Si pensi in particolare a dove e come mantenere le strutture dati del kernel.

    (b) Che differenza in scalabilità ci si può aspettare (sempre su questa architettura) fra un Sistema Operativo a microkernel e uno monolitico?

    (c) Si possono avere deadlock "locali" dovuti all'uso di risorse del kernel tutte situate in una memoria locale? E ci possono essere deadlock "globali" che coinvolgono risorse del kernel che si trovano in differenti memorie locali? Quale soluzione per evitarli, in caso, appare più adeguata?

    Sì è possibile avere deadlocks di entrambe i tipi. Visto che le risorse del kernel sono ben note, e non cambiano facilmente nelle varie release del sistema operativo, la soluzione più adeguata (per evitare deadlocks) è l'ordinamento preventivo delle risorse del kernel: ogni processo può allocare tali risorse solo in ordine crescente.