Esame 2019/07/15

Blascovich Alessio

Domanda 1

MAX PUNTI: 4
Domanda:

Spiegare in dettaglio che cos'è il trashing, perche' si verifica e le possibili misure per mitigarlo/eliminarlo.

Soluzione:

Il fenomeno del trashing si verifica quando il numero dei page fault aumenta a causa del numero limitato di frame che un processo alloca.
Questo porta ad un circolo vizioso che si manifesta in perdita di prestazioni con conseguente:

1. Diminuzione dell'uso della CPU perchè molti processi sono in attesa di gestire page fault.
2. Vedendo la CPU poco usata l' O.S. carica ulteriori processi aumentando la multiprogrammazione.
3. I nuovi processi rubano frame ai vecchi processi, per cui aumentano i page fault.
4. Ritorno al punto 1

Ad un certo punto il throughput ha un tracollo, per cui bisogna stimare con una certa precisione l'effettivo numero di frame che servono ad un processo.
Una soluzione accettabile è determinare un numero di page fault "ideale", quindi se ho troppi page fault aumento il numero di frame associati ad quel processo, mentre se ne ho troppo pochi vuol dire che può rilasciare dei frame.

Domanda 2

MAX PUNTI: 6
Domanda:

Data la situazione descritta successivamente, con 4 tipi di risorse e 5 processi in competizione per le risorse.
Si supponga che il processo P_1 effettui la richiesta Req=(0,4,2,0).
Usando l'algoritmo del banchiere indicare se la richiesta porta ad uno stato safe, in caso affermativo elencare una sequenza.

	Allocation			
P_0	0	0	1	2
P_1	1	0	0	0
P_2	1	3	5	4
P_3	0	6	3	2
P_4	0	0	1	4

Max			
0	0	1	2
1	6	5	0
2	3	5	6
0	6	4	2
0	6	5	6

Availaible			
1	5	2	0

Soluzione:

Calcolo la prima tabella che mi serve definita come *Need*[i]=*Max*[i]-*Allocation*[i].

	Need			
P_0	0	0	0	0
P_1	0	6	5	0
P_2	1	0	0	2
P_3	0	0	1	0
P_4	0	6	4	2

Adesso verifico che (con i si intende l'indfice del prcesso che fa la richiesta) Req[]>*Need*[i][] e che Req[]>*Availaible*[], il test è passato, quindi posso verificare se è safe o meno la richiesta del processo, simulandola con l'ausilio di 3 nuove tabelle:

1. *Ava*[]=*Available*[]-Req[]

2. *Alloc*[i][]=*Allocation*[i][]+Req[]
3. *Needed*[i][]=*Need*[i][]-Req[]

Ava

1	1	0	0

Alloc

P_0	0	0	1	2
P_1	1	4	2	0
P_2	1	3	5	4
P_3	0	6	3	2
P_4	0	0	1	4

Needed

P_0	0	0	0	0
P_1	0	2	3	0
P_2	1	0	0	2
P_3	0	0	1	0
P_4	0	6	4	2

Ora verifico di essere in uno stato safe con i seguenti passagi.
Inanzitutto creo *work*[]=*Ava*[] e *Finish*[]={False,...,False}.

1. Trovo un indice i tale che:
   - *Finish*[i]==False
   - *Needed*[i][] $\leq$ *Work*[]

   Se non esiste una tale i andare al punto 3.
2. *Work*[]=*Work*[]+*Alloc*[i][]
   Finish[i][]=True
   Torna al punto 1.
3. Se *Finish*[i]==True $\forall$ i, allora il sistema è in stato safe.

Ora lancio l'algoritmo con *Work*=(1,1,0,0):

1. Iterazione:
   - i=0 *Finsh*[0]=False and (0,0,0,0) $\leq$ (1,1,0,0) $\Rightarrow$ *Finsh*[0]=True e *Work*[]=*Work*[]+*Alloc*[0][]=(1,1,1,2)
2. Iterazione:
   - i=0 *Finish*[0]==True $\Rightarrow$ ++i
   - i=1 *Finish*[1]==False ma (0,2,3,0) > (1,1,1,2) $\Rightarrow$ ++i
   - i=2 *Finish*[2]==False and (1,0,0,2) $\leq$ (1,1,1,2) $\Rightarrow$ *Finsh*[2]=True e *Work*[]=*Work*[]+*Alloc*[2][]=(2,4,6,6)
3. Iterazione:
   - i=0 *Finish*[0]==True $\Rightarrow$ ++i
   - i=1 *Finish*[0]==False and (0,2,3,0) $\leq$ (2,4,6,6) $\Rightarrow$ *Finsh*[1]=True e *Work*[]=*Work*[]+*Alloc*[1][]=(3,8,8,6)
4. Iterazione:
   - i=0,...,2 *Finish*[i]==True $\Rightarrow$ ++i
   - i=3 *Finish*[3]==False and (0,0,1,0) $\leq$ (3,8,8,6) $\Rightarrow$ *Finsh*[3]=True e *Work*[]=*Work*[]+*Alloc*[3][]=(3,14,11,8)
5. Iterazione:
   - i=0,...,3 *Finish*[i]==True $\Rightarrow$ ++i
   - i=4 *Finish*[4]==False and (0,6,4,2) $\leq$ (3,14,11,8) $\Rightarrow$ *Finsh*[4]=True e *Work*[]=*Work*[]+*Alloc*[4][]=(3,14,12,12)
6. Iterazione:
   - $\forall$ i *Finish*[i]==True $\Rightarrow$ ++i

Esco dalla funzione e restituisco False, quindi nonn è presente uno stato unsafe.

Domanda 3

MAX PUNTI: 6
Domanda:

Data una memoria divisa in 5 partizioni da 100K, 500K, 200k, 300k e 600k (nell'ordine) ed i processi P_i di dimensioni 212K, 417K, 112K e 426K.
Applico le politiche del First-fit, Best-fit e Worst-fit, quale performa meglio?
Soluzione:

Devo disegnare delle tabelle in cui le applico tutte.

- *First-fit*:

100k	
500K	P_0
200k	P_2
300k	
600k	P_1

Con questo metodo metto il processo nel primo slot abbstanza grande da poterlo contenere.
Così facendo però, il P_3 non viene caricato in memoria ed ho ancora 400k disponibili(+ quella interna ad ogni partizione che non viene usata).

- *Best-fit*

100k	
500K	P_1
200k	P_2
300k	P_0
600k	P_3

Con questo metodo inserisco il processo nel buco più piccoloc he lo può contenere, devo però scannerizzare ogni volta la coda che contiene le partizioni di memoria.
Seppur più pesante questo metodo mi permette di allocare tutti i processi lasciando liberi solo 100k di memoria(+ quella interna ad ogni partizione che non viene usata).

- *Worst-fit*

100k	
500K	P_1
200k	
300k	P_2
600k	P_0

Questo metodo come il *Best-fit* necessita di scannerizzare ogni volta la lista delle partizioni, perchè alloca un processo nello slot più grande che è disponibile in quel momento.
Così facendo si hanno 300k di memoria del tutto liberi(+ quelli interni ad ogni partizione che non vengono usati).

Ne risulta che che seppur più pesante del *First-fit* il *Best-fit* è quello che permette di allocare più processi con un minore spreco di memoria.

Domanda 4

MAX PUNTI: 4
Domanda:

Si consideri un calcolatore con memoria virtuale in cui i frame sono di 2K, per descrivere l'indirizzo di un frame servono 32 bit.
Quanti KB di memoria virtuale posso rappresentare se uso una paginazione a 2 livello?.
Soluzione:

So che per una pagina ho 2KB=2048B=2^{11}B $\Rightarrow$ 11 bit di offset, ora devo sotrarre dal totale dio bit(32) il numero di bit necessari per l'offset, ottengo 32-11=21 che sono i bit che rappresentano un frame.
So che ogni frame è grande 2KB e con 11 bit posso rappresentare 2^{11}=2097152 indirizzi, quindi posso indirizzare un totale di 2KB*2^{11}=4194304KB=4GB.

Domanda 5

MAX PUNTI: 5
Domanda:

Descrivere nel dettaglio come avviene la comunicazione tra 2 processi in UNIX tramite memoria condivisa.

Soluzione:

Nel mondo UNIX si posso usare delle funzioni di C a basso livello per craere aree di memoria condivisa.
Prima si crea l'area di memoria identificata da un id, sucessivamente tutti i processi che si vogliono attaccare devo creare un puntatore a quell'area tramite l'id.
Per scrivere/leggere sul segmento in comune di usare la funzione di stampa/lettura indicando come stream il puntatore prima otenuto.
Una volta terminato il processo può rimuovere dal suo spazio di indirizzi il puntatoreall'area condivisa, al termine di tutto il programma si dovrà eseguire una funzione che libera del tutto l'area di memoria che prima era condivisa tra i processi.

Domanda 6

MAX PUNTI: 7

Domanda:

Scrivere in pseudocidice un algoritmo basato sui semafori che permetta di:

1. Far arrivare il processo P alla risorsa A e fargliela usare.
2. Far arrivare i processi P_1 e P_2 alla risorsa A, non importa l'ordine di utilizzo della risorsa A.
3. Liberare la risorsa e ritornare al punto 1.

Bisognerebbe avere una cosa del tipo:

$$\mathrm{P} \to P_i \to P_i \to \mathrm{P} \to P_i \to P_i \to \mathrm{P} \to P_i \to P_i \to \dots$$

Si supponga pure che P, P_1 e P_2 operino secondo uno schema di esecuzione perpetua `while(true){...}`.

Soluzione:

In questo caso si posso usare dei semaforti, quindi si possono usare le primitive `V(s)` per incrementare il semaforo e `P(s)` tentare di decrementare di 1 il valore del semaforo.
Possibile implementazione di P, P_1 e P_2:

```
Semaforo turno=2;
Semaforo R1=false;
Semaforo R2=false;

    P(){
        while(true){
            P(Turno)
            P(Turno)
            A
            V(R1)
            V(R2)
        }
    }
```

```
P1(){
    while(true){
        P(R1)
        A
        V(P1)
    }
}
```

```
P2(){
    while(true){
        P(R2)
        A
        V(P2)
    }
}
```