SISTEMI OPERATIVI E LABORATORIO, CORSI A e B Prova del 9/6/2010 CORSO |A| |B|

ESERCIZIO A-1 (4 punti)

Un negozio di un barbiere è dotato di un'unica una poltrona per il servizio dei clienti e di un numero illimitato di sedie per l'attesa. La poltrona è una risorsa, che può essere occupata da un cliente per il tempo necessario all'esecuzione del taglio dei capelli, oppure dal barbiere per attendere (dormendo) l'arrivo di clienti da servire. Le sedie sono risorse utilizzate dai clienti per attendere il proprio turno, dopo essere entrati nel negozio. Il barbiere e i clienti sono processi ad ambiente locale, che interagiscono con lo scambio di messaggi.

All'apertura del negozio il barbiere si addormenta sulla poltrona e sarà svegliato del primo cliente che entrerà nel negozio.
Il generico cliente ha il seguente comportamento:

- entra nel negozio e, con un messaggio, chiede al barbiere il taglio dei capelli; quindi occupa una sedia in attesa della risposta del barbiere, che lo invita ad occupare la poltrona;
- si siede sulla poltrona e si addormenta in attesa che venga eseguito il taglio dei capelli;
- al termine viene risvegliato da un messaggio del barbiere che comunica la tariffa. Quindi esegue il pagamento e ne dà conferma al barbiere con un messaggio;
- infine esce dal negozio.

Il barbiere ha il seguente comportamento ciclico:

- occupa la poltrona e attende la richiesta da parte di un cliente. Se non la riceve immediatamente, si addormenta;
- quando riceve la richiesta, che eventualmente lo risveglia, risponde al cliente invitandolo ad occupare la poltrona; quindi esegue il taglio dei capelli mentre il cliente si addormenta;
- al termine risveglia il cliente comunicandogli la tariffa da pagare e attende dal cliente la conferma del pagamento.

Il problema viene risolto con le seguenti primitive di comunicazione:

- *send(destinatario, &messaggio).* Primitiva asincrona per la comunicazione diretta, che invia al processo di indice *destinatario* il messaggio contenuto nella variabile *messaggio;*
- *send(PortaDestinatario, &messaggio).* Primitiva asincrona per la comunicazione indiretta, che scrive il messaggio contenuto nella variabile *messaggio* nella porta del processo destinatario (mailbox), denominata *PortaDestinatario;*
- *receive(mittente, &messaggio),* Primitiva bloccante per la comunicazione diretta, che restituisce nella variabile *messaggio* un messaggio inviato dal processo di indice *mittente;*
- *receive(&mittente, &messaggio),* Primitiva bloccante per la comunicazione indiretta, che legge un messaggio dalla porta del processo che la esegue e restituisce l'indice del processo mittente nella variabile *mittente* e il messaggio nella variabile *messaggio.*

Si chiede di completare il programma del barbiere e il frammento di codice che controlla i clienti che entrano nel negozio, inserendo le opportune primitive di comunicazione.

SOLUZIONE

```
Barbiere
{
receive(&cliente, &RichiestaTaglio);
// attende il primo cliente; se non vi sono clienti in attesa si addormenta; //
// il contenuto del messaggio è implicito e non viene verificato //
 while(true)        {
            messaggio="si accomodi sulla poltrona"
            send(cliente, &messaggio);    / il cliente  prende posto sulla poltrona //
            <esegue il taglio dei capelli>  // nel frattempo il cliente si addormenta //
            tariffa= TariffaTaglio;     // sveglia il cliente, comunicandogli la tariffa //
            send(cliente, &tariffa);
            // attende la conferma del pagamento, che riceverà nella variabile ConfermaPagamento. Si suppone che la
            // conferma arrivi in un tempo finito e che non sia necessario verificare il contenuto del messaggio //
            receive(cliente, &ConfermaPagamento); // attende il cliente successivo; eventualmente si addormenta
            receive(&cliente, &RichiestaTaglio);
            }
}

Cliente
//Frammento di codice//
< entra nel negozio >
mess= "RichiestaTaglio"; send(PorteBarbiere, &mess);
// attende dal barbiere l'invito a occupare la poltrona; eventualmente si addormenta //
receive(barbiere, &chiamata);    // il contenuto del messaggio è implicito e non viene verificato //
<raggiunge la poltrona e la occupa>
// il barbiere inizia il taglio dei capelli; il cliente attende (dormendo) che il barbiere termini e comunichi la tariffa //
receive(barbiere, &tariffa);
paga(tariffa);
ConfermaPagamento ="OK";
send(barbiere, &ConfermaPagamento);
<esce dal negozio>
// fine del frammento di codice //
```

SISTEMI OPERATIVI E LABORATORIO, CORSI A e B Prova del 9/6/2010 CORSO |A| |B|

ESERCIZIO A-2 (4 punti)

Un sistema gestisce il processore con una politica a code multiple ispirata a WINDOWS, con 5 Code Pronti associate alle classi di priorità di valori compresi tra 0 e 4. I thread pronti sono inseriti nella coda della classe corrispondente alla propria priorità; il processore viene assegnato al thread che occupa la prima posizione nella coda non vuota della classe di massima priorità; ai thread pronti della stessa classe di priorità si applica la politica Round Robin con Quanto di Tempo (QdT) di *10 msec*. Quando un thread va in esecuzione gli viene assegnato un intero quanto di tempo, indipendentemente dal tempo consumato nel precedente turno di esecuzione.

La politica prevede la revoca del processore, che avviene *immediatamente* al verificarsi di uno dei seguenti eventi:

- il thread in esecuzione esaurisce il quanto di tempo;
- viene riattivato un thread con priorità maggiore di quello in esecuzione.

Come regola generale, la priorità dei thread è statica, ma la priorità di un generico thread P_i viene transitoriamente elevata al valore 4 se, allo scadere di un quanto di tempo assegnato ad un altro thread P_k, si verifica che il thread P_i è rimasto in stato di pronto per 3 interi quanti di tempo (non si considerano i quanti di tempo utilizzati parzialmente da thread che escono dallo stato di esecuzione prima di aver esaurito il quanto di tempo loro assegnato). L'incremento permane per 2 quanti di tempo (utilizzati totalmente o parzialmente), dopo di che la priorità del thread torna al valore statico.

Al tempo *t* sono presenti i seguenti thread:

- Thread A, con priorità 3, in stato di esecuzione;
- Thread B, con priorità 1, in stato di pronto;
- Thread C, con priorità 2, in stato di attesa sul semaforo Sem1, la cui coda non contiene altri thread;
- Thread D, con priorità 2, in stato di attesa sul semaforo Sem2, la cui coda non contiene altri thread;

Sono definiti i semafori Sem1, Sem2 e Sem3, con i seguenti valori e composizioni delle code:

- Sem1: Valore 0, CodaSem1 → C
- Sem2: Valore 0, CodaSem2 → D
- Sem3: Valore 0, CodaSem3 → ∅

Al tempo *t* si ha la seguente situazione:

- Thread A: il contatore di programma punta alla chiamata della primitiva *wait(Sem3)* che ha tempo di esecuzione 0; successivamente il programma invoca la funzione *FunzioneA;*
- Thread B: il contatore di programma punta alla chiamata della primitiva *signal(Sem1)* che ha tempo di esecuzione 0; successivamente il programma invoca la funzione *FunzioneB,* la cui esecuzione richiede *15 msec,* e **infine** invoca la primitiva *signal(Sem3)* che ha tempo di esecuzione 0;
- Thread C: il contatore di programma punta all'invocazione della funzione *FunzioneC1* la cui esecuzione richiede *5 msec,* successivamente invoca la primitiva *signal(Sem2)* che ha tempo di esecuzione 0; quindi invoca la funzione *FunzioneC2,* la cui esecuzione richiede *50 msec;*
- Thread D: il contatore di programma punta all'invocazione della funzione *FunzioneD* la cui esecuzione richiede *60 msec.*

Utilizzando la tabella seguente, si chiede di analizzare l'evoluzione dello stato dei thread A, B, C, D in base alla politica di scheduling, fino all'invocazione della funzione *FunzioneA* da parte del thread A.

SOLUZIONE

Tempo	Evento	Thread A	Thread B	Thread C	Thread D
t+0	Eseguita *wait(Sem3)*	Stato: attesa su Sem3	Stato: esecuzione	Stato: attesa su Sem1	Stato: attesa su Sem2
		Priorità: 3	Priorità: 1	Priorità: 2	Priorità: 2
		QdT in stato di pronto: 0	QdT in stato di pronto:0	QdT in stato di pronto:0	QdT in stato di pronto:0
t+0	Eseguita *signal(Sem1)*	Stato: attesa su Sem3	Stato: pronto	Stato: esecuzione	Stato: attesa su Sem2
		Priorità: 3	Priorità: 1	Priorità: 2	Priorità: 2
		QdT in stato di pronto: 0	QdT in stato di pronto:0	QdT in stato di pronto:0	QdT in stato di pronto:0
t+5	Eseguita *signal(Sem2)*	Stato: attesa su Sem3	Stato: pronto	Stato: esecuzione	Stato: pronto
		Priorità: 3	Priorità: 1	Priorità: 2	Priorità: 2
		QdT in stato di pronto: 0	QdT in stato di pronto:0	QdT in stato di pronto:0	QdT in stato di pronto:0
t+10	Esaurito QdT	Stato: attesa su Sem3	Stato: pronto	Stato: pronto	Stato: esecuzione
		Priorità: 3	Priorità: 1	Priorità: 2	Priorità: 2
		QdT in stato di pronto: 0	QdT in stato di pronto:1	QdT in stato di pronto:0	QdT in stato di pronto:0
t+20	Esaurito QdT	Stato: attesa su Sem3	Stato: pronto	Stato: esecuzione	Stato: pronto
		Priorità: 3	Priorità: 1	Priorità: 2	Priorità: 2
		QdT in stato di pronto: 0	QdT in stato di pronto:2	QdT in stato di pronto:0	QdT in stato di pronto:0
t+30	Esaurito QdT	Stato: attesa su Sem3	Stato: esecuzione	Stato: pronto	Stato: pronto
		Priorità: 3	Priorità: 4	Priorità: 2	Priorità: 2
		QdT in stato di pronto: 0	QdT in stato di pronto:0	QdT in stato di pronto:0	QdT in stato di pronto:1
t+40	Esaurito QdT	Stato: attesa su Sem3	Stato: esecuzione	Stato: pronto	Stato: pronto
		Priorità: 3	Priorità: 4	Priorità: 2	Priorità: 2
		QdT in stato di pronto: 0	QdT in stato di pronto:0	QdT in stato di pronto:0	QdT in stato di pronto:0
t+45	Eseguitoa *signal(Sem3)* Thread A invoca *FunzioneA*	Stato: esecuzione	Stato: terminato	Stato: pronto	Stato: pronto
		Priorità: 3	Priorità: -	Priorità: 2	Priorità: 2
		QdT in stato di pronto: 0	QdT in stato di pronto: -	QdT in stato di pronto:1	QdT in stato di pronto:2

IL thread A invoca *FunzioneA* al tempo t+45

ESERCIZIO A-3 (3 punti)
Un sistema con risorse dei tipi R1, R2, R3, R4, rispettivamente di molteplicità [5, 6, 3, 6] non adotta alcuna politica di prevenzione dello stallo, ma lo riconosce e lo elimina dopo che si sia eventualmente verificato. Al tempo *t* sono presenti e attivi i thread A, B, C e D , con i valori di assegnazione mostrati in Tabella2. Allo stesso tempo, la disponibilità di risorse è mostrato in Tabella 2 e si è raggiunto lo stato mostrato nelle tabelle.

Assegnazione (tempo t)				
	R1	R2	R3	R4
A	2		1	2
B	1	1	0	1
C			2	1
D	1	3		2

Tabella 1

Disponibilità (tempo t)			
R1	R2	R3	R4
1	2	0	0

Tabella 2

Esigenza residua (tempo t)				
	R1	R2	R3	R4
A	2	0	0	1
B	4	2	2	0
C	0	0	1	2
D	1	0	0	1

Tabella 3

Al tempo $t+k$ si sono verificati i seguenti eventi:

- A ha richiesto 1 istanza di R4 e si è sospeso con richiesta pendente di R4;
- B ha richiesto 1 istanza di R3 e si è sospeso con richiesta pendente di R3;
- C ha richiesto 1 istanza di R4 e si è sospeso con richiesta pendente di R4;
- D ha richiesto 1 istanza di R4 e si è sospeso con richiesta pendente di R4.

Ciascuno dei processi sospesi sarà riattivato se e quando la sua richiesta pendente potrà essere soddisfatta.
Il sistema rileva che tutti i processi sono sospesi e che si è raggiunto uno stallo. Allo scopo di eliminarlo, sopprime il processo B, recuperando le risorse ad esso assegnate.
Si chiede se la soppressione del processo B è sufficiente ad eliminare lo stallo.
Si tenga presente che lo stallo è eliminato se la soppressione permette di riattivare immediatamente almeno un processo e se esiste almeno una sequenza di avanzamento che permette a tutti i processi di ottenere le risorse di cui hanno esigenza e, conseguentemente, di giungere alla terminazione. Sebbene la natura del problema non richiede che i processi rendano nota la propria esigenza, si suppone che le esigenze residue dei processi al tempo *t* siano quelle mostrate in Tabella 3.

SOLUZIONE

Stato raggiunto dopo la forzata terminazione del processo B (si suppone nota l'esigenza residua):

Assegnazione				
	R1	R2	R3	R4
A	2		1	2
B	-	-	-	-
C			2	1
D	1	3		2

Disponibilità			
R1	R2	R3	R4
2	3	0	1

Esigenza residua				
	R1	R2	R3	R4
A	2	0	0	1
B	-	-	-	-
C	0	0	1	2
D	1	0	0	1

Dopo il rilascio:

1) I processi A, C e D sono riattivati in quanto la richiesta pendente di ciascun processo può essere soddisfatta .
   Quindi, i processi attivi sono A, C, D
   Se avanza A, può ottenere tutte le risorse di cui ha esigenza e terminare;
   dopo la terminazione di A la disponibilità diviene [4, 3, 1, 3]
2) Nessun processo da riattivare. Sono attivi i processi C e D.
   Se avanza C, può ottenere tutte le risorse di cui ha esigenza e terminare
   dopo la terminazione di C la disponibilità diviene [4, 3, 3, 4]
3) Nessun processo da riattivare. E` attivo il processo D.
   D può avanzare, ottenere tutte le risorse di cui ha esigenza e quindi terminare
   Alla fine la disponibilità diviene [5, 6, 3, 6]. Tutti i processi sono terminati e tute le risorse sono state restituite.

Di conseguenza:

- lo stallo è eliminato.

ESERCIZIO A-4 (2 punti) molto simile ad A2, riconsiderare.
Si consideri un sistema nel quale è definito il semaforo *sem* e i processi P1, P2 e P3.
Al tempo *t* il semaforo *sem* ha la seguente configurazione:
Sem: valore: 0, coda: P3
Allo stesso tempo, la CodaPronti contiene il processo P2 e il processo P1 è in esecuzione.
Lo scheduler dei processi non prevede il prerilascio del processore.
Dire come si modificano il semaforo *sem* e la CodaPronti e quale processo è in esecuzione se si verificano (in alternativa) le due seguenti sequenze di eventi:

a) P1 esegue *wait (Sem)* e successivamente il processo in esecuzione esegue *wait(Sem);*
b) P1 esegue *signal (Sem)* e successivamente il processo in esecuzione esegue *wait(Sem);*
c) P1 esegue *signal (Sem)* e successivamente il processo in esecuzione esegue *signal(Sem);*

SOLUZIONE

	Sequenze di eventi	In Esecuzione	Coda Pronti	Valore Sem, coda
a-1	P1 esegue *wait (Sem)*	P2	-	0, P3->P1
a-2	Il processo in esecuzione esegue *wait (Sem)*	-	-	0, P3->P1->P2
b-1	P1 esegue *signal (Sem)*	P1	P2->P3	0, -
b-2	Il processo in esecuzione esegue *wait (Sem)*	P2	P3	0, P1
c-1	P1 esegue *signal (Sem)*	P1	P2->P3	0,-
c-2	Il processo in esecuzione esegue *signal (Sem)*	P1	P2->P3	1,-

ESERCIZIO A-5 (2 punti)
In un sistema UNIX, un processo P esegue il seguente frammento di codice:

```
…
printf("uno");
a = fork(); //generazione del processo figlio A
b = fork();
if (a>0 && b>0) {
       printf("tre");
       }
else if (a==0 || b==0){
              printf("quattro");
              execl("/bin/xx",NULL);
       }
     else printf("cinque");
…
```

Dire cosa stampa il processo P e il processo figlio A eseguendo questo frammento di codice se il file /bin/xx è eseguibile?

SOLUZIONE
Inizialmente il processo P stampa "`uno`". Successivamente stampa "`tre`" solo se entrambe le fork hanno successo, altrimenti, se almeno una fork fallisce, stampa "`cinque`".
Se la prima fork ha successo il processo A viene creato e stampa "`quattro`".

ESERCIZIO B-1 (4 punti)

Un sistema simile a Windows gestisce la memoria con paginazione a domanda mediante un *Working Set Manager*. A un certo tempo lo stato di occupazione della memoria (senza considerare i blocchi riservati al sistema operativo) è quello descritto nella seguente Core Map, i cui elementi hanno valore nullo se il blocco è libero, o altrimenti identificano il processo e la pagina a cui il blocco è assegnato.

	A,2	C,1	A,4	A,8		B,1		B,2	C,2	C,5	B,5	B,6		C,6	C,7	B,8	A,9	C,9	
10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	29	29

Nel sistema sono presenti i processi A, B e C, le cui tabelle delle pagine sono le seguenti (il campo *TempoRif* registra il ***tempo virtuale del processo*** al quale è avvenuto l'ultimo riferimento alla pagina):

Pagina	Blocco	TempoRif
0		
1		
2	11	8
3		
4	13	5
5		
6		
7		
8	14	9
9	27	4
Processo A		

Pagina	Blocco	TempoRif
0		
1	16	9
2	18	7
3		
4		
5	21	4
6	22	5
7		
8	26	1
9		
Processo B		

Pagina	Blocco	TempoRif
0		
1	12	2
2	19	11
3		
4		
5	20	9
6	24	3
7	25	6
8		
9	29	5
Processo C		

A ogni processo è assegnato un *WorkingSet Ammissibile* di 4 pagine. Per ogni pagina riferita da un processo che avanza, si procede nel modo seguente:

- se la pagina è presente in memoria, si scrive il valore attuale del tempo virtuale del processo nel campo *TempoRif* della tabella delle pagine;
- se si verifica un *PageFault*, la pagina riferita viene caricata in un blocco disponibile (anche se il numero di pagine caricate, o *insieme residente*, supera la dimensione del *Working Set Ammissibile*), registrando il valore attuale del tempo virtuale del processo nel campo *TermpoRif*. I blocchi disponibili vengono assegnati in ordine crescente di indice.
- Il *Working Set Manager* viene attivato periodicamente o quando il numero di blocchi disponibili si riduce a 3, e si comporta nel modo seguente;
  - considera i soli processi la cui dimensione dell'*insieme residente* (numero di pagine caricate in memoria) superi quella del *WorkingSet Ammissibile*, e per questi processi ordina globalmente le pagine per valori decrescenti del parametro *TempoVirtuale- TempoRif,* dove *TempoVirtuale* è il valore attuale del tempo virtuale del processo;
  - scarica dalla memoria le pagine secondo questo ordinamento, fino a quando il numero di blocchi disponibili diventa uguale a 8. In caso di parità tra due o più pagine si considera l'ordine alfabetico dei processi.

A partire dal tempo considerato il sistema evolve nel modo seguente:

1. avanza il processo C, che ai tempi virtuali 12, 13 e 14 riferisce rispettivamente le pagine 1, 6 e 9
2. avanza il processo B, che ai tempi virtuali 10,11, 12, 13 e 14 riferisce rispettivamente le pagine 4, 5, 6, 7 e 2.
3. quindi avanza il processo C, che ai tempi virtuali 15, 16, 17 riferisce rispettivamente le pagine 7, 5, 8.
4. quindi si attiva il *Working Set Manager.* I tempi virtuali dei processi A, B e C sono rispettivamente 9, 14 e 17.

Si chiede la configurazione della *CoreMap* e delle tabelle delle pagine dei 3 processi al termine dei punti 1, 2, 3 e 4. Nei casi nei quali viene eseguito il *Working Set Manager* indicare anche il tempo attuale dei processi al momento dell'esecuzione del *Working Set Manager*, e quante e quali pagine vengono rimosse.

SISTEMI OPERATIVI E LABORATORIO, CORSI A e B Prova del 9/6/2010 CORSO |A| |B|

SOLUZIONE

1) Configurazione della *CoreMap* e delle tabelle delle pagine al termine del punto 1:

	A,2	C,1	A,4	A,8		B,1		B,2	C,2	C,5	B,5	B,6		C,6	C,7	B,8	A,9	C,9	
10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	29	29

Pagina	Blocco	TempoRif
0		
1		
2	11	8
3		
4	13	5
5		
6		
7		
8	14	9
9	27	4
Processo A		

Pagina	Blocco	TempoRif
0		
1	16	9
2	18	7
3		
4		
5	21	4
6	22	5
7		
8	26	1
9		
Processo B		

Pagina	Blocco	TempoRif
0		
1	12	12
2	19	11
3		
4		
5	20	9
6	24	13
7	25	6
8		
9	29	14
Processo C		

Non è stato eseguito il *Working Set Manager*

2) Configurazione della *CoreMap* e delle tabelle delle pagine al termine del punto 2:

B,4	A,2	C,1	A,4	A,8	B,7	B,1		B,2	C,2	C,5	B,5	B,6		C,6	C,7	B,8	A,9	C,9	
10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	29	29

Pagina	Blocco	TempoRif
0		
1		
2	11	8
3		
4	13	5
5		
6		
7		
8	14	9
9	27	4
Processo A		

Pagina	Blocco	TempoRif
0		
1	16	9
2	18	14
3		
4	10	10
5	21	11
6	22	12
7	15	13
8	26	1
9		
Processo B		

Pagina	Blocco	TempoRif
0		
1	12	12
2	19	11
3		
4		
5	20	9
6	24	13
7	25	6
8		
9	29	14
Processo C		

Viene quindi eseguito il *Working Set Manager* che rimuove 5 pagine dei soli processi B e C.
Il tempo attuale del processo B è: 14
Il tempo attuale del processo C è: 14

Vengono quindi rimosse le pagine: B8, C7, C5, B1, B4

-	A,2	C,1	A,4	A,8	B,7	-		B,2	C,2	-	B,5	B,6		C,6	-	-	A,9	C,9	
10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	29	29

Pagina	Blocco	TempoRif
0		
1		
2	11	8
3		
4	13	5
5		
6		
7		
8	14	9
9	27	4
Processo A		

Pagina	Blocco	TempoRif
0		
1	-	-
2	18	14
3		
4	-	-
5	21	11
6	22	12
7	15	13
8	-	-
9		
Processo B		

Pagina	Blocco	TempoRif
0		
1	12	12
2	19	11
3		
4		
5	-	-
6	24	13
7	-	-
8		
9	29	14
Processo C		

SISTEMI OPERATIVI E LABORATORIO, CORSI A e B Prova del 9/6/2010 CORSO |A| |B|

3) Configurazione della *CoreMap* e delle tabelle delle pagine al termine del punto 3:

C,7	A,2	C,1	A,4	A,8	B,7	C,5	C,8	B,2	C,2		B,5	B,6		C,6			A,9	C,9	
10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	29	29

Pagina	Blocco	TempoRif
0		
1		
2	11	8
3		
4	13	5
5		
6		
7		
8	14	9
9	27	4
Processo A		

Pagina	Blocco	TempoRif
0		
1		
2	18	14
3		
4		
5	21	11
6	22	12
7	15	13
8		
9		
Processo B		

Pagina	Blocco	TempoRif
0		
1	12	12
2	19	11
3		
4		
5	16	16
6	24	13
7	10	15
8	17	17
9	29	14
Processo C		

4) Configurazione della *CoreMap* e delle tabelle delle pagine al termine del punto 4:

Viene eseguito il *Working Set Manager* che rimuove 3 pagine del solo processo C.
Il tempo attuale del processo C è: 17

Vengono quindi rimosse le pagine: C2, C1, C6

C,7	A,2	-	A,4	A,8	B,7	C,5	C,8	B,2	-		B,5	B,6		-			A,9	C,9	
10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	29	29

Pagina	Blocco	TempoRif
0		
1		
2	11	8
3		
4	13	5
5		
6		
7		
8	14	9
9	27	4
Processo A		

Pagina	Blocco	TempoRif
0		
1		
2	18	14
3		
4		
5	21	11
6	22	12
7	15	13
8		
9		
Processo B		

Pagina	Blocco	TempoRif
0		
1	-	-
2	-	-
3		
4		
5	16	16
6	-	-
7	10	15
8	17	17
9	29	14
Processo C		

ESERCIZIO B-2 (4 punti)
Si consideri un disco organizzato con *NCilindri= 100* (numerati da 0 a 99), *NFacce= 4* (numerate da 0 a 3) e *NSettori= 100* (numerati da 0 a 99), dove ogni settore (corrispondente a un blocco) contiene 1024 byte. Gli *indici* dei blocchi del disco sono interi compresi nell'intervallo *[0, CapacitàDisco),* mentre gli indirizzi fisici sono *terne* del tipo *(cil, faccia, sett)*. Si ricorda che l'indice *b* corrisponde alla terna *(cil, faccia, sett)* con:

cil= b **div** *(NFacce*NSettori);*
faccia= (b **mod** *(NFacce*NSettori))* **div** *NSettori;*
sett= (b **mod** *(NFacce*NSettori))* **mod** *NSettori* .

Il periodo di rotazione è di *10 msec*, per cui ogni settore viene percorso in *0,1 msec.* Il tempo di seek è somma di una componente fissa di *10 msec* e di una componente variabile pari a *1 ms* per ogni cilindro attraversato. Al termine di ogni operazione di *seek,* le teste di lettura/scrittura sono posizionate sul settore *0* del cilindro di destinazione. La capacità di buffering dell'unità di controllo del disco permette di eseguire senza ritardo di comandi relativi a settori consecutivi di uno stesso cilindro.
Su questo disco è definito un File System simile a NTFS, dove ogni file è allocato in *run* di *blocchi contigui*. Al file *pippo* sono allocati i seguenti 2 run:

- *run(1) ≡ (6080, 30),*
- *run(2) ≡ (15190, 20),*

dove il generico run è definito da una coppia composta, nell'ordine, dall'indice del primo blocco del run e dalla lunghezza del run espressa in blocchi.
Si chiede il tempo necessario a leggere per intero il file *pipo,* a partire dal suo primo blocco, supponendo che le teste di lettura/scrittura siano inizialmente posizionate sul cilindro *0.*

SOLUZIONE

Il run *run(1)* è suddiviso tra:

- una traccia che comprende i blocchi di indici compresi (estremi inclusi) tra:
  - *blocco(1,1)* con indirizzo fisico (*Cil(1,1), Faccia(1,1), Sett(1,1)) ≡ (15, 0, 80)*
  - *blocco(1,2),* con indirizzo fisico (*Cil(1,2), Faccia(1,2), Sett(1,2)) ≡ (15, 0, 99)*
- una traccia che comprende i blocchi di indici compresi (estremi inclusi) tra:
  - *blocco(1,3)= blocco(1,2)+1* con indirizzo fisico (*Cil(1,3), Faccia(1,3), Sett1,3)) ≡ (15, 1, 0),*
  - *blocco(1,4)*, con indirizzo fisico (*Cil(1,4), Faccia(1,4), Sett(1,4)) ≡ (15, 1, 9)*

Il run *run(2),* è suddiviso tra:

- una traccia che comprende i blocchi di indici compresi (estremi inclusi) tra:
  - *blocco(2,1)* con indirizzo fisico (*Cil(2,1), Faccia(2,1), Sett(2,1)) ≡ (37, 3, 90)*
  - *blocco(2,2),* con indirizzo fisico (*Cil(2,2), Faccia(2,2), Sett(2,2)) ≡ (37, 3, 99);*
- una traccia che comprende i blocchi di indici compresi (estremi inclusi) tra:
  - *blocco(2,3)=blocco(2,2)+1* con indirizzo fisico (*Cil(2,3), Faccia(2,3), Sett(2,3)) ≡ (38, 0, 0),*
  - *blocco(2,4)*, con indirizzo fisico (*Cil(2,4), Faccia(2,4), Sett(2,4)) ≡ (38,0, 9)*

Pertanto:
Per leggere *run(1):*

Tempo necessario per raggiungere *Cil(1,1): 10+(15-0)*1= 25;*
Ulteriore tempo necessario per raggiungere *Sett(1,1): 80* 0,1= 8;*
Ulteriore tempo necessario per percorrere i settori da *Sett(1,1)* a *Sett(1,2): 20* 0,1= 2;*
Tempo necessario per raggiungere *Cil(1,3): 0*
Ulteriore tempo necessario per raggiungere *Sett(1,3): 0;*
Ulteriore tempo necessario per percorrere i settori da *Sett(1,3)* a *Sett(1,4): 10* 0,1= 1;*

Per leggere *run(2):*

Ulteriore empo necessario per raggiungere *Cil(2,1): 10+(37-15)*1= 32;*
Ulteriore tempo necessario per raggiungere *Sett(2,1): 90* 0,1= 9;*
Ulteriore tempo necessario per percorrere i settori da *Sett(2,1)* a *Sett(2,2): 10* 0,1= 1;*
Tempo necessario per raggiungere *Cil(2,3): 10+ (38-37)*1= 11;*
Ulteriore tempo necessario per raggiungere *Sett(2,3):0;*
Ulteriore tempo necessario per percorrere i settori da *Sett(2,3)* a *Sett(2,4): 10* 0,1= 1;*

Tempo totale per leggere il file *pippo: 25+8+2+0+0+1+32+9+1+11+0+1=90 msec*

SISTEMI OPERATIVI E LABORATORIO, CORSI A e B Prova del 9/6/2010 CORSO |A| |B|

ESERCIZIO B-3 (3 punti)

In un file system UNIX si consideri il file */home/lidia/foto/panorama.jpg,* appartenente all'utente *lidia* e la directory */home/cinzia/docs* appartenente all'utente *cinzia* che è un hard link alla directory */home/lidia/foto*.
I diritti associati alle directory *home, lidia, foto, cinzia, docs* e al file *panorama.jpg* sono i seguenti:

	owner (r w x)	*group (r w x)*	*others (r w x)*
home	*1 1 1*	*1 1 1*	*1 1 1*
lidia	*1 0 1*	*0 0 0*	*0 0 0*
foto	*1 1 1*	*1 1 1*	*0 0 0*
panorama.jpg	*1 1 0*	*1 1 1*	*1 1 1*
cinzia	*1 1 1*	*0 0 0*	*1 0 0*
docs	*1 1 1*	*1 1 1*	*0 0 0*

Si supponga inoltre che il contatore degli hard link della directory */home/lidia/foto* sia pari a 2.

1. Dire quali tra le operazioni di lettura, scrittura e cancellazione sono permesse all'utente *cinzia* supponendo che *cinzia* e *lidia* appartengano a gruppi diversi.
2. Dire quali tra le operazioni di lettura, scrittura e cancellazione sono permesse all'utente *cinzia* supponendo che *cinzia* e *lidia* appartengano allo stesso gruppo.

SOLUZIONE

IPOTESI	LETTURA del file */home cinzia/docs/panorama.jpg* [SI/NO]	SCRITTURA sul file */home cinzia/docs/panorama.jpg* [SI/NO]	CANCELLAZIONE della directory */home/cinzia/docs* [SI/NO]
cinzia e *lidia* appartengono a gruppi diversi.	NO	NO	NO
cinzia e *lidia* appartengono allo stesso gruppo.	SI	SI	SI

ESERCIZIO B.4 (2 punti)

In un sistema UNIX il processo P esegue la chiamata di sistema `int pipe (int fd[2])`, dove `fd[0]` è il descrittore del lato di lettura e `fd[1]` quello del lato di scrittura, e successivamente esegue la chiamata di sistema `fork` generando il processo figlio F1. Successivamente P esegue la chiamata di sistema `close(fd[0])` e F1 esegue la chiamata di sistema `close(fd[1])`.
Date le seguenti istruzioni eseguite in sequenza dopo il tempo t:

1. P esegue `int a=write(fd[1],"prova di",8)`
2. P esegue `int a=read(fd[0],"&buf,9)`
3. P esegue `int a=write(fd[1]," esame",6)`
4. F1 esegue `int a=read(fd[0],"&buf,5)`

Dire quali di queste sono eseguibili, e quale è il valore della variabile `a` al loro termine e cosa viene letto nelle istruzioni di lettura.

SOLUZIONE

OPERAZIONE	ESEGUIBILE?	Valore di a:
1.	SI	8
2.	NO	-
3.	SI	6
4.	SI	5

L'istruzione nella riga 2 non legge niente
L'istruzione nella riga 4 legge "`prova`"

ESERCIZIO B.5 (2 punti)

Un disco RAID di livello 4 è composto da 5 dischi fisici, numerati da 0 a 4. Le strip corrispondono a blocchi.
I blocchi del disco virtuale V sono mappati nei dischi 0, 1, 2, 3: precisamente il blocco *b* del disco V è mappato nel blocco *b* ***div*** *4* del disco fisico di indice *b* ***mod*** *4*. Il disco 4 è ridondante e il suo blocco di indice *i* contiene la parità dei blocchi di indice *i* dei dischi 0, 1, 2, 3.
I blocchi di indice compreso tra 8 e 15 del disco virtuale contengono rispettivamente:

8: 1 1 0 0 …	12: 1 1 0 0 …
9: 1 0 0 1 …	13: 1 0 0 1 …
10: 1 1 1 1 …	14: 1 1 1 1 …
11: 1 0 1 0 …	15: 1 0 1 0 …

Si chiede:

1) Quali blocchi fisici dei dischi è necessario leggere per modificare il contenuto del blocco virtuale di indice 10 in 1 1 1 0 …
2) Quali blocchi fisici dei dischi è necessario scrivere per modificare il contenuto del blocco virtuale di indice 12 in 1 1 1 1 …
3) Come si modifica il contenuto del disco ridondante in seguito alle modifiche dei blocchi virtuali 10 e 12

SOLUZIONE

1) Il blocco virtuale 10 è allocato sul blocco fisico 2 del disco 2.
La sua modifica comporta la lettura del blocco fisico 2 dei dischi 2 e 4 e la conseguente scrittura del blocco fisico 2 dei dischi 2 e 4.

2) Il blocco virtuale 12 è allocato sul blocco fisico 3 del disco 0.
La sua modifica comporta la lettura del blocco fisico 3 dei dischi 0 e 4 e la conseguente scrittura del blocco fisico 3 dei dischi 0 e 4.

3) dopo la modifica i blocchi fisici 2 e 3 del disco ridondante 4 diventano:
blocco fisico 2: 0 0 0 1 …
blocco fisico 3: 0 0 1 1 …