ESERCIZIO A-1 (4 punti)
In un sistema che gestisce il processore con la tecnica delle code multiple (una coda FIFO per ogni valore di priorità; i processi pronti di uguale priorità sono inseriti in una stessa coda; il processore viene assegnato al processo che occupa la prima posizione nella coda non vuota di massima priorità; ai processi pronti di uguale priorità si applica la politica Round Robin), il quanto di tempo è di *10 msec*. Quando un processo va in esecuzione gli viene assegnato un intero quanto di tempo, indipendentemente dal tempo consumato nel precedente turno di esecuzione.
La politica prevede il prerilascio, che avviene immediatamente al verificarsi dell'evento che lo determina, senza attendere l'esaurimento del quanto di tempo corrente.
Al tempo t, nel sistema sono presenti i seguenti processi:

1. Processo A, con priorità 3, che al tempo t è sospeso sul semaforo *Attesa*;
2. Processo B, con priorità 2, che al tempo t ottiene il processore entrando nello stato di esecuzione;
3. Processo C, con priorità 1, che al tempo t è pronto;

La composizione delle tre code al tempo t è la seguente:

- coda 1: C
- coda 2: Ø
- coda 3: Ø

Si chiede quale è il processo in esecuzione e la composizione delle 3 code **nell'intervallo di tempo da *t* a *t+50***, nel corso del quale si verificano i seguenti eventi:

1. al tempo *t*+5 il processo in esecuzione esegue *Signal(Attesa)*;
2. al tempo *t*+8 il processo in esecuzione esegue *Signal(Attesa)*;
3. al tempo t+9 viene generato il processo D con priorità 3;
4. al tempo t+16 il processo in esecuzione esegue *Wait(Attesa)*;
5. al tempo t+ 21 termina il processo in esecuzione;
6. al tempo t+30 viene generato il processo E con priorità 1;
7. al tempo t+31 termina il processo in esecuzione;
8. al tempo t+33 termina il processo in esecuzione;
9. al tempo t+40 il processo in esecuzione esegue *Wait(Attesa)*;

SOLUZIONE

Tempo	In Esecuzione	Coda 1	Coda 2	Coda 3	Valore del semaforo *Attesa*	Coda del semaforo *Attesa*
t	B(2)	C(1)	Ø	Ø	0	A(3)
t+5	A(3)	C	B	Ø	0	Ø
t+8	A(3)	C	B	Ø	1	Ø
t+9	A(3)	C	B	D(3)	1	Ø
t+15	D(3)	C	B	A(3)	1	Ø
t+16	D(3)	C	B	A(3)	0	Ø
t+21	A(3)	C	B	Ø	0	Ø
t+30	A(3)	C->E	B	Ø	0	Ø
t+31	B(3)	C->E	Ø	Ø	0	Ø
t+33	C(1)	E	Ø	Ø	0	Ø
t+40	E(1)	Ø	Ø	Ø	0	C(1)
t+50	E(1)	Ø	Ø	Ø	0	C(1)

ESERCIZIO A-2 (4 punti)

Nel problema dei lettori-scrittori, *n* processi lettori e *m* processi scrittori competono per l'accesso ad una struttura dati condivisa. In particolare i processi lettori accedono alla struttura dati esclusivamente per leggere, senza modificare i dati, mentre i processi scrittori accedono alla struttura dati senza restrizioni, con la possibilità di modificare i dati.

Per evitare interferenze, i processi scrittori accedono alla base dati in mutua esclusione (rispetto agli altri scrittori e ai lettori), mentre i processi lettori accedono alla base dati in mutua esclusione rispetto agli scrittori, ma senza vincolo di mutua esclusione rispetto agli altri lettori (in altre parole, più lettori possono accedere concorrentemente alla struttura dati).

Per la soluzione del problema, si utilizzano i seguenti dati condivisi da tutti i processi:

- *LettoriAmmessi*: intero non negativo; valore iniziale 0
- *LettoriInAttesa*: intero non negativo; valore iniziale 0
- *ScrittoreAmmesso*: intero non negativo; valore iniziale 0
- *ScrittoriInAttesa*: intero non negativo; valore iniziale 0

Se i lettori e gli scrittori sono thread di uno stesso processo realizzati a livello kernel, è nota la seguente soluzione che utilizza i seguenti semafori:

- *mutex* (valore iniziale 1): semaforo utilizzato per la mutua esclusione sulle sezioni critiche con le quali i processi verificano la condizione di accesso;
- *AttesaLettori (*valore iniziale 0)*:* semaforo utilizzato per la sospensione dei lettori;
- *AttesaScrittori (*valore iniziale 0)*:* semaforo utilizzato per la sospensione degli scrittori; valore iniziale 0.

```
Lettore_i
{
   ……..
   // prologo dell'accesso in lettura//
   OK = 0 ;
   wait(mutex);
   if ScrittoriInAttesa == 0 and ScrittoreAmmesso == 0
   {
        LettoriAmmessi ++; OK= 1;
        }
   else LettoriInAttesa ++;
   signal(mutex);
   if OK == 0  wait(AttesaLettori);
  < esegue accesso in lettura >
   // epilogo dell'accesso in lettura//
   wait(mutex);
   LettoriAmmessi -- ;
   if LettoriAmmessi== 0  and  ScrittoriInAttesa > 0    {
        ScrittoreAmmesso = 1; signal(AttesaScrittori);
        }
   signal(mutex)
   ……..
}
```

```
Scrittore_j
    {
   ……..
   // prologo dell'accesso in scrittura//
   OK = 0;
   wait(mutex);
   if LettoriAmmessi== 0 and ScrittoreAmmesso == 0{
         ScrittoreAmmesso =1; OK= 1;
        }
   signal(mutex);
  if OK == 0 wait(AttesaScrittori);
  < esegue accesso in scrittura >
  // epilogo dell'accesso in scrittura//
   wait(mutex);
   ScrittoreAmmesso = 0;
   if LettoriInAttesa > 0            {
        while LettoriInAttesa> 0  {
        LettoriInAttesa --; LettoriAmmessi++;
        signal(LettoriInAttesa);
        }
   else if ScrittoriInAttesa > 0     {
                ScrittoriInAttesa --; ScrittoreAmmesso =1;
                signal(AttesaScrittori);
                }
        }
   signal(mutex)
end;
}
```

Si chiede di trasformare la precedente soluzione utilizzando, invece dei semafori, i meccanismi POSIX *pthread_mutex_lock, pthread_mutex_unlock, pthread_cond_wait e pthread_cond_signal.*

Utilizzare la variabile *P_mutex*,di tipo *mutex* e, per la sospensione dei lettori e degli scrittori, le variabili *AttesaLettori* e *AttesaScrittori*, di tipo *condition*. Si suggerisce di riflettere se in questa soluzione è necessario conservare la variabile locale *OK*.

SOLUZIONE

Lettore_i

```
{
   ……..
   // prologo dell'accesso in lettura//
   pthread_mutex_lock(&P_mutex);
   LettoriInAttesa++;
   while (ScrittoriInAttesa > 0 or ScrittoreAmmesso > 0)        pthread_cond_wait(&AttesaLettori, &P_mutex);
   LettoriInAttesa--; LettoriAmmessi ++;
   pthread_mutex_unlock(&P_mutex);

   < esegue accesso in lettura >
   // epilogo dell'accesso in lettura//

   pthread_mutex_lock(&P_mutex);
   LettoriAmmessi --;
   if (LettoriAmmessi== 0  and  ScrittoriInAttesa > 0) pthread_cond_signal(&AttesaScrittori);
   pthread_mutex_unlock(&P_mutex);
   ……..
}
```

Scrittore_j

```
    {
   …….
   // prologo dell'accesso in scrittura//
   pthread_mutex_lock(&P_mutex);
   ScrittoriInAttesa++;
   while (LettoriAmmessi > 0 or ScrittoreAmmesso > 0)         pthread_cond_wait(&AttesaScrittori, &P_mutex);
   ScrittoriInAttesa--; Scrittore Ammesso=1;
   pthread_mutex_unlock(&P_mutex);

  < esegue accesso in scrittura >
  // epilogo dell'accesso in scrittura//

   pthread_mutex_lock(&P_mutex);
   ScrittoreAmmesso = 0;
   if (LettoriInAttesa>0)
        while (LettoriInAttesa> 0) pthread_cond_ signal(&AttesaLettori);
   else if ScrittoriInAttesa > 0 pthread_cond_ signal(&AttesaScrittori);
   pthread_mutex_unlock(&P_mutex);
   ……..
}
```

ESERCIZIO A3 (3 punti)
In un sistema dove i processi operano in ambiente locale e dispongono di una libreria per la realizzazione dei thread a livello utente, sono presenti il processo P1 con priorità 10, e il processo P2 con priorità 5. Lo scheduling dei processi avviene con una politica a priorità e prerilascio (va in esecuzione il processo pronto con il massimo valore di priorità).
Per ogni processo, i thread alternano tra lo stato di esecuzione e quello di pronto. Le transizione avvengono quando il thread esegue la funzione *ThreadYield* oppure quando il processo cui appartiene il thread esce dallo stato di esecuzione. Il thread che passa dallo stato di esecuzione a quello di pronto viene inserito nell'ultima posizione della coda dei thread pronti. Lo scheduling dei thread avviene con politica FIFO.
Al tempo *T* lo stato dei processi e dei thread è il seguente:

P1: Priorità 10; Stato di P1: esecuzione;
Thread di P1: T11, T12; Thread in esecuzione: T11; CodaPronti dei thread → T12
P2: Priorità 5; Stato di P2: pronto;
Thread di P2: T21, T22, T23; Thread in esecuzione: ∅; CodaPronti dei thread → T21→ T22→ T23
A partire dal tempo *t* si verifica la seguente sequenza di eventi :

1. il thread in esecuzione esegue la primitiva (non bloccante) *send(&mess1, P2)*
2. il thread in esecuzione esegue *ThreadYield;*
3. il thread in esecuzione esegue la primitiva (bloccante) *receive(&buff1, P2)*
4. il thread in esecuzione esegue la primitiva (non bloccante) *send(&mess2, P1)*
5. il thread in esecuzione esegue *ThreadYield;*
6. il thread in esecuzione esegue la primitiva (bloccante) *receive(&buff1, P2)*

Supponendo che prima del tempo *T* non siano state eseguite primitive *send* o *receive*, si chiede quali sono gli eventi che determinano la ricezione di messaggi e quali sono i thread che li ricevono. (compilare la tabella)

SOLUZIONE

Lo stato in tabella è quello **immediatamente dopo** l'evento

Evento	Processo P1 (priorità 10)			Processo P2 (priorità 5)			Messaggio inviato o ricevuto
	Stato	Thread in esecuzione	CodaPronti dei thread	Stato	Thread in esecuzione	CodaPronti dei thread	
1	Esecuzione	T11	→ T12	Pronto	∅	→T21→T22→T23	Inviato *mess1* Ricevuto
2	Esecuzione	T12	→ T11	Pronto	∅	→T21→T22→T23	Inviato Ricevuto
3	Bloccato	∅	T11→T12	Esecuzione	T21	→T22→T23	Inviato Ricevuto
4	Esecuzione	T11	→ T12	Pronto	∅	→T22→T23→T21	Inviato *mess2* Ricevuto *mess2*
5	Esecuzione	T12	→ T11	Pronto	∅	→T22→T23→T21	Inviato Ricevuto
6	Bloccato	∅	→T11→T12	Esecuzione	T22	→T23→T21	Inviato Ricevuto

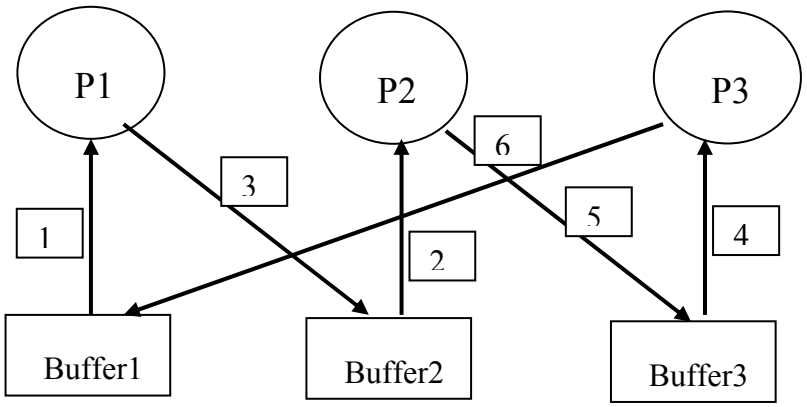
ESERCIZIO A-4 (2 punti)

Si considerino i processi P1, P2 e P3 che condividono i buffer Buffer1, Buffer2 e Buffer3. Ciascun buffer ha la capacità di un carattere e ogni operazione eseguita dai processi legge o scrive un singolo carattere. Ai buffer sono associati i semafori SemBuff1, SemBuff2 e SemBuff3.
Inizialmente ciascuno dei buffer contiene 1 carattere; quindi i tre processi eseguono le seguenti sequenze di operazioni:

Processo 1:	Processo 2:	Processo 3:
wait (SemBuff1);	wait (SemBuff2);	wait (SemBuff3);
legge un carattere da Buffer1;	legge un carattere da Buffer2;	legge un carattere da Buffer3;
wait (SemBuff2);	wait (SemBuff3);	wait (SemBuff1);
scrive un carattere su Buffer2;	scrive un carattere su Buffer3;	scrive un carattere da Buffer1;
signal (SemBuff2);	signal (SemBuff3);	signal (SemBuff1);
signal (SemBuff1)	signal (SemBuff2)	signal (SemBuff3)

Si chiede:
1) quale deve essere il valore iniziale dei tre semafori?
2) le operazioni eseguite dai tre processi possono determinare stallo?
3) in caso affermativo, evidenziare sul grafo (inserendo e numerando le frecce) una sequenza che determina stallo

SOLUZIONE
1) valore iniziale dei tre semafori: valore 1 per ciascun semaforo
2) le operazioni eseguite dai tre processi possono determinare stallo? SI
3) una sequenza che determina stallo



ESERCIZIO A-5 (2 punti)

In un sistema UNIX, vengono eseguite in sequenza le seguenti operazioni:

a) il processo P esegue con sucesso una chiamata *pipe,* che restituisce i descrittori *fd[0]* e *fd[1];*
b) il processo P esegue con successo una *fork,* generando il processo F1 che immediatamente esegue con successo una *exec;*
c) il processo P esegue *close(fd[1];*
d) il processo P esegue con successo una *fork,* generando il processo F2 che immediatamente esegue con successo una *exec*
e) il processo F2 esegue con sucesso una chiamata *pipe,* che restituisce i descrttori *fd'[0]* e *fd'[1];*

Si chiede quali tra le seguenti operazioni sono eseguibili dopo l'evento e):

1) il processo F1 esegue *write(fd[1], &OutBuffer, 10)*
2) il processo F2 esegue *read(fd[0], &InBuffer, 5).*
3) il processo F2 esegue *write(fd[1], &OutBuffer, 5)*
4) il processo P esegue *read(fd'[0], &InBuffer, 2)*

SOLUZIONE

1) il processo F1 esegue *write(fd[1], &OutBuffer, 10):* Eseguibile
2) il processo F2 esegue *read(fd[0], &InBuffer, 5)):* Eseguibile
3) il processo F2 esegue *write(fd[1], &OutBuffer, 5)):* Non eseguibile
4) il processo P esegue *read(fd'[0], &InBuffer, 2)):* Non eseguibile

ESERCIZIO B-1 (4 punti)

Si consideri un sistema che gestisce la memoria con paginazione a domanda, applicando un algoritmo di sostituzione LRU locale una politica di controllo dinamico del Working Set. Per ogni processo sono definiti i seguenti dati:

- l'intero *MaxBlocchi:* massimo numero di blocchi disponibili per il caricamento del *Working Set,* che viene ridefinito periodicamente dal demone *WorkingSetManager;*
- l'intero *PagineResidenti,* uguale al numero di pagine attualmente caricate in memoria e variabile nel tempo;
- la tabella delle pagine di ogni processo, con campi *Pagina, Blocco, R (bit di pagina riferita) e DP (Distanza Passata).*

Quando un processo avanza, per ogni pagina riferita:

- se il riferimento determina *Page Fault*, la pagina riferita viene caricata nel blocco libero individuato dal primo elemento della lista *BlocchiDisponibili* (che su suppone sempre non vuota), viene incrementata la variabile *PagineResidenti* e viene definito *DP= 0;*
- in ogni caso, si assegna *R= 1.*

Il processo *WorkingSetManager,* che interviene periodicamente, esegue le seguenti operazioni per ogni processo:
Fase 1) per ogni pagina residente in memoria aggiorna la distanza passata con il seguente algoritmo:

- se *R= 0* assegna *DP= DP+ 1;*
- altrimenti assegna *R= 0* e *DP= 0*

Fase 2) se *PagineResidenti> MaxBlocchi:*

- scarica dalla memoria principale *PagineResidenti- MaxBlocchi*, selezionandole in ordine decrescente del valore di *DP;*
- se per l'ultima pagina scaricata si ha *DP< 7,* assegna *MaxBlocchi= MaxBlocchi + 1;*
  se *PagineResidenti< MaxBlocchi,* assegna *MaxBlocchi= MaxBlocchi – 1.*

Al tempo T1, subito dopo un intervento di *WorkingSetManager*, per il processo *P* si ha ***MaxBlocchi* = 7** e la tabella delle pagine ha la seguente configurazione:

Pagina	Blocco	R	DP
0	-	-	-
1	6	0	2
2	-	-	-
3	-	-	-
4	-	-	-
5	8	0	5
6	-	-	-
7	10	0	4
8	15	0	8
9	-	-	-
10	20	0	9
11	-	-	-
12	30	0	7

Dopo *T1* e prima del tempo *T2* il processo *P* avanza e riferisce nell'ordine le seguenti pagine: ***5, 3, 0, 5, 0, 9, 3, 8, 9, 11***
Al tempo *T2* entra in esecuzione *WorkingSetManager*, che applica la politica sopra definita.
Al tempo *T4* termina l'intervento di *WorkingSetManager*
Supponendo che il contenuto della lista *BlocchiDisponibili* al tempo T1 sia → 50 → 51 → 52 → 53 → 54 → …., si chiede:

a) il valore di *PagineResidenti* e la configurazione della Tabella delle Pagine del processo P al tempo *T2*;
b) la configurazione della Tabella delle Pagine del processo *P* al tempo *T3*, quando termina la Fase 1 di *WorkingSetManager*;
c) il valore di *PagineResidenti* e di *MaxBlocchi* e la configurazione della Tabella delle Pagine del processo *P* al tempo *T4,* quando termina la Fase 2 di *WorkingSetManager*.

SOLUZIONE

Tempo T2: Tabella delle Pagine

Pagina	Blocco	R	DP
0	51	1	-
1	6	0	2
2	-	-	-
3	50	1	-
4	-	-	-
5	8	1	5
6	-	-	-
7	10	0	4
8	15	1	8
9	52	1	-
10	20	0	9
11	53	1	-
12	30	0	7

Pagine Residenti= 10

Tempo T3: Tabella delle Pagine

Pagina	Blocco	R	DP
0	51	0	0
1	6	0	3
2	-	-	-
3	50	0	0
4	-	-	-
5	8	0	0
6	-	-	-
7	10	0	5
8	15	0	0
9	52	0	0
10	20	0	10
11	53	0	0
12	30	0	8

Tempo T4: Tabella delle Pagine

Pagina	Blocco	R	DP
0	51	0	0
1	6	0	3
2	-	-	-
3	50	0	0
4	-	-	-
5	8	0	0
6	-	-	-
7	-	-	-
8	15	0	0
9	52	0	0
10	-	-	-
11	53	0	0
12	-	-	-

Pagine Residenti= 7; MaxBlocchi= 8

ESERCIZIO B-2 (4 punti)
In un sistema che gestisce la memoria con paginazione, sono presenti i processi A, B, C e D. Lo stato di occupazione della memoria è descritto dalla *Core Map*, dove per ogni blocco si specifica nell'ordine: il processo a cui appartiene la pagina caricata, l'indice della pagina caricata e il bit di pagina riferita.
Al tempo *t* la configurazione della *CoreMap* è quella riportata in tabella, dove, per esempio, nel blocco 11 è caricata la pagina 6 del processo C con bit di pagina riferita uguale a 1.

A,2 1	D,4 0	B,2 0	D,3 1	C,7 1	A,5 1	A,6 0	B,3 1	B,9 1	C,1 1	D,6 0	C,6 1	D,8 1	A,3 1	D,5 0	A,9 1	C,2 1	C,3 0	C,9 0	A,1 1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Il sistema usa l'algoritmo di sostituzione *Second Chance* (globale) e prima di ogni invocazione il puntatore è posizionato sul blocco successivo alla *vittima* individuata nell'invocazione precedente. Al tempo *t* il puntatore è posizionato sull'elemento 7 della core map (evidenziato in grigio).

Mostrare come si modifica la *CoreMap* (e la posizione del puntatore) se dal tempo *t* si verificano (in alternativa) le seguenti serie di eventi:

1. Il processo A riferisce le pagine 5, 10, 6; successivamente il processo D riferisce le pagine 7, 6, 5
2. Il processo B riferisce le pagine 4, 1, 6, 7; successivamente il processo C riferisce le pagine 3, 9, 4

SOLUZIONE

1) Il processo A riferisce le pagine 5, 10, 6; successivamente il processo D riferisce le pagine 7, 6, 5
Configurazione dopo che il processo A ha riferito le pagine 5, 10 e 6:

A,2 1	D,4 0	B,2 0	D,3 1	C,7 1	A,5 1	A,6 1	B,3 0	B,9 0	C,1 0	A,10 1	C,6 1	D,8 1	A,3 1	D,5 0	A,9 1	C,2 1	C,3 0	C,9 0	A,1 1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Configurazione dopo che il processo D ha riferito le pagine 7, 6, e 5:

A,2 1	D,4 0	B,2 0	D,3 1	C,7 1	A,5 1	A,6 1	B,3 0	B,9 0	C,1 0	A,10 1	C,6 0	D,8 0	A,3 0	D,7 1	A,9 0	C,2 0	D,6 1	D,5 1	A,1 1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

2) Il processo B riferisce le pagine 4, 1, 6, 7; successivamente il processo C riferisce le pagine 3, 9, 4
Configurazione dopo che il processo B ha riferito le pagine 4, 1, 6 e 7:

A,2 1	D,4 0	B,2 0	D,3 1	C,7 1	A,5 1	A,6 0	B,3 0	B,9 0	C,1 0	B,4 1	C,6 0	D,8 0	A,3 0	B,1 1	A,9 0	C,2 0	B,6 1	B,7 1	A,1 1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Configurazione dopo che il processo C ha riferito le pagine 3, 9 e 4:

A,2 0	C,3 1	C,9 1	D,3 0	C,7 0	A,5 0	C,4 1	B,3 0	B,9 0	C,1 0	B,4 1	C,6 0	D,8 0	A,3 0	B,1 1	A,9 0	C,2 0	B,6 1	B,7 1	A,1 0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

ESERCIZIO B3 (3 punti)

Si consideri un File System che alloca i file in sequenze contigue *(run)* individuate mediante coppie del tipo *(inizio, lunghezza)* dove *inizio* è l'indice del blocco fisico iniziale della sequenza e *lunghezza* esprime il numero di blocchi che la compongono. Ogni file è descritto da un *Master Record* che contiene, oltre ad altri attributi, una o più coppie *(inizio, lunghezza)*. Il File System è ospitato da un disco con *NCilindri= 50*, *NFacce= 4* e *NSettori= 20*. Il tempo necessario per percorrere un settore è di *0,1 msec* e il tempo medio di esecuzione di un'operazione di *seek* (comprensivo del ritardo rotazionale per raggiungere il primo settore indirizzato) è di *5 msec.*
A un certo tempo viene eseguita l'operazione *read(filename, &buffer, Ncaratteri),* per effetto della quale si leggono 10 blocchi a *partire dal nono blocco del file*. Nel *Master Record* del file *filename* sono definite, nell'ordine, le seguenti sequenze contigue:

1. *(1525, 15)*
2. *(3170, 12)*

dove l'indice di blocco *1525* corrisponde alla terna *(cilindro= 19, faccia= 0, settore= 5)* e l'indice di blocco *3170* corrisponde alla terna *(cilindro= 39, faccia= 2, settore= 10).*
Si calcoli il tempo necessario per eseguire la lettura, supponendo che le teste di lettura scrittura siano inizialmente posizionate sul cilindro *12* e che tempo di esecuzione delle eventuali operazioni di *seek* sia sempre uguale a quello medio.

SOLUZIONE

- Indice del primo blocco estratto dalla sequenza 1: blocco 1533
- Numero di blocchi estratti dalla sequenza 1: 7 blocchi
- Numero di cilindri su cui è distribuita la sequenza 1: 1
- Tempo necessario per estrarre i blocchi della sequenza 1: 7 * 0,1= 0,7 msec
- Indice del primo blocco estratto dalla sequenza 2: blocco 3170
- Numero di blocchi estratti dalla sequenza 2: 3 blocchi
- Numero di cilindri su cui è distribuita la sequenza 2: 1
- Tempo necessario per estrarre i blocchi della sequenza 2: 3* 0,1= 0,3 msec
- Numero di operazioni di *seek* eseguite: 2
- Tempo totale impiegato: 2*5 + 1,7+ 0,3= 11 msec.

Esercizio B4 (2 punti)
Un disco RAID di livello 1 è composto da 8 dischi fisici, numerati da 0 a 7. Le strip corrispondono a blocchi, in particolare la strip numero *Si* del RAID è allocata nel blocco (*Si* div 4) nel disco (*Si* mod 4), ed è replicata nel blocco (*Si* div 4) del disco (*Si* mod 4 + 4).
I blocchi di indice 2 e 3 degli 8 dischi contengono rispettivamente:

Disco n.	0	1	2	3	4	5	6	7
Blocco n.2	1010……	1100……	1000……	0000……	1010……	1100……	1000……	0000……
Blocco n.3	1111……	0011……	0101……	0001……	1111……	0011……	0101……	0001……

Dire quali operazioni vengono effettuate, quali blocchi e dischi fisici vengono acceduti e quanto tempo è necessario (in numero di accessi) per eseguire le seguenti operazioni:

1) lettura delle strip numero 10,11,12 e 13.
2) scrittura di 1001…… sulla strip 15

Nel caso dell'operazione 2 dire anche come cambia il contenuto del RAID.

SOLUZIONE

1) lettura delle strip numero 10,11,12 e 13:

- strip n. 10: Disco n.: 2 Blocco n.: 2 Tipo operazione: Lettura
- strip n. 11: Disco n.: 3 Blocco n.: 2 Tipo operazione: Lettura
- strip n. 12: Disco n.: 0 Blocco n.: 3 Tipo operazione: Lettura
- strip n. 13: Disco n.: 1 Blocco n.: 3 Tipo operazione: Lettura

Tempo necessario: 1 accesso

2) scrittura di 1001…… sulla strip 15

- strip n. 15: Disco n.: 3 Blocco n.: 3 Tipo operazione: Scrittura
- strip n. 15: Disco n.: 7 Blocco n.: 3 Tipo operazione: Scrittura

Tempo necessario: 1 accesso

Contenuto del RAID dopo l'operazione:

Disco n.	0	1	2	3	4	5	6	7
Blocco n.2	1010……	1100……	1000……	0000……	1010……	1100……	1000……	0000……

Blocco n.3	1111……	0011……	0101……	1001……	1111……	0011……	0101……	1001……

ESERCIZIO B5 (2 PUNTI)

Un sistema UNIX gestisce la memoria con segmentazione combinata con paginazione. La memoria virtuale di ogni processo comprende i segmenti *codice*, *dati* e *pila.* Per la gestione si utilizzano, oltre alla *Core Map,* anche tre tabelle delle pagine per ogni processo, rispettivamente per il segmento codice, per il segmento dati e per il segmento pila. Le tre tabelle sono individuate da altrettanti puntatori contenuti nella *Process Structure* del processo.

Il gestore della memoria adotta la politica della *paginazione pura:* cioè non esegue alcun caricamento iniziale di pagine alla generazione dei processi, e demanda il caricamento delle pagine al solo meccanismo dei *Page Faults.*

A un certo tempo, il processo *ProcK* esegue la chiamata di sistema *fork* che genera il processo *ProcJ*. Al momento della chiamata la tabelle delle pagine dei tre segmenti del processo *ProcK* hanno i seguenti contenuti:

Pagina	Blocco
0	91
1	-
2	70
3	-
4	-
5	79
6	-
ProcK: segmento codice	

Pagina	Blocco
0	-
1	-
2	92
3	99
4	-
5	65
6	-
ProcK: segmento dati	

Pagina	Blocco
0	-
1	-
2	87
3	88
4	-
5	-
6	-
ProcK: segmento pila	

La primitiva che esegue la chiamata di sistema crea sul disco due file di swap per il processo *ProcJ* e vi copia, rispettivamente, i dati e la pila del padre, quindi assegna e inizializza le tabelle delle pagine di *ProcJ*,

Si chiede:

1. Quante tabelle devono essere assegnate al processo *ProcJ*?
2. I puntatori alle tabelle contenuti nella *Process Structure* di *ProcJ* sono uguali o diversi da quelli del padre?
3. Qual è il contenuto delle tabelle delle pagine del processo *ProcJ* subito dopo la *fork ?*

SOLUZIONE

1. tabelle delle assegnate al processo *ProcJ*: 2 (per i dati e la pila)
2. puntatori alle tabelle contenuti nella *Process Structure* di *ProcJ;*
   puntatore alla tabella del segmento codice: uguale
   puntatore alla tabella del segmento dati: differente
   puntatore alla tabella del segmento pila: differente
3. contenuto delle tabelle delle pagine del processo *ProcJ* subito dopo la *fork ?*

Pagina	Blocco
0	-
1	-
2	-
3	-
4	-
5	-
6	-
ProcJ: segmento dati	

Pagina	Blocco
0	-
1	-
2	-
3	-
4	-
5	-
6	-
ProcJ: segmento pila	