ESERCIZIO A-1 (4 punti)

In un sistema LINUX che realizza i thread a livello kernel, si vogliono introdurre due funzioni, denominate *MessWrite* e *MessRead*, per realizzare la comunicazione tra thread attraverso un unico buffer condiviso denominato *PseudoPipe.* La comunicazione avviene con le seguenti modalità:

- l'accesso a *PseudoPipe*,in lettura e in scrittura, è consentito a tutti i thread:
- i messaggi sono sequenze di *L* caratteri, con $0 \leq L \leq 256$;
- *PseudoPipe* è un buffer circolare di $N >> 256$ caratteri, al quale sono associati, con significato evidente, i puntatori *UltimoCarattereScritto* e *PrimoCarattereDaLeggere* e la variabile binaria *BufferVuoto.* Il numero attuale di posizioni libere nel buffer è uguale a (*PrimoCarattereDaLeggere- (UltimoCarattereScritto+ 1)) **mod** N*. Inizialmente il buffer è vuoto, con *PrimoCarattereDaLeggere = (UltimoCarattereScritto+ 1) **mod** N* e *BufferVuoto= 1*. Si noti che la variabile *BufferVuoto* è necessaria per distinguere tra il caso di Buffer completamente pieno e quello di Buffer completamente vuoto, entrambe caratterizzate da *PrimoCarattereDaLeggere= (UltimoCarattereScritto+ 1) **mod** N.*
- Per ogni thread*,* il buffer *OutMess* ha una lunghezza di 256 caratteri e il messaggio da scrivere, di lunghezza *L*, occupa i primi *L* caratteri ($0 \leq L \leq 256$) di questo buffer. Analogamente, il buffer *InMess* ha una lunghezza di 256 caratteri e il messaggio letto, di lunghezza *L*, va ad occupare i primi *L* caratteri ($0 \leq L \leq 256$) di questo buffer.

La specifica delle funzioni *MessWrite* e *MessRead* è la seguente:

- ***void function** MessWrite (&PseudoPipe, &OutMess, L)*: se il numero delle posizioni libere in *PseudoPipe* è maggiore di *L,* scrive L in *PseudoPipe[UltimoCarattereScritto+ 1 **mod** N]* e quindi copia il messaggio nelle *L* successive posizioni del buffer. Alla fine il valore di *UltimoCarattereScritto* risulta incrementato ***(mod** N)* di *L+1*. Se il numero delle posizioni libere in *PseudoPipe* è minore di *L+1,* il thread viene sospeso e verrà riattivato, per ripetere il tentative di scrittura, dalla funzione *MessRead.*
- ***void function** read(&PseudoPipe, &InMess, &L):* se se il buffer *PseudoPipe* non è vuoto, legge la lunghezza del messaggio dalla posizione *PseudoPipe[PrimoCarattereDaLeggere]* e la restituisce con la variabile *L*; successivamente legge il corpo del messaggio dalle successive *L* posizioni di *PseudoPipe* e lo scrive nel buffer *InMess*. Alla fine il valore di *PrimoCarattereDaLeggere* risulta incrementato ***(mod** N)* di *L+1*. Se il buffer *PseudoPipe* è vuoto, il tread viene sospeso e verrà riattivato, per ripetere il tentative di lettura, dalla funzione *MessWrite.*

Per la sincronizzazione, i thread utilizzano le variabili *SospesiInScrittura* e *SospesiInLettura* (di tipo *condition)*, e *mux* (di tipo *mutex*), e i meccanismi ***pthread_condition_wait, pthread_condition_signal, pthread_mutex_lock** e **pthread_mutex_unlock*** dello standard POSIX.
Si chiede di completare lo pseudo codice delle funzioni *read* e *write*, parzialmente definito nello schema di soluzione.

SOLUZIONE

```
void function MessWrite (&PseudoPipe, &OutMess, L)
{
pthread_mutex_lock(&mux)
while (BufferVuoto==0 and (PrimoCarattereDaLeggere-(UltimoCarattereScritto+ 1)) mod N <L+1) )
        //* (PrimoCarattereDaLeggere-(UltimoCarattereScritto+ 1)) mod N  è il numero di posizioni disponibili //*
        pthread_condition_wait(&SospesiInScrittura,&mux);
UltimoCarattereScritto= (UltimoCarattereScritto  +1) mod N;
PseudoPipe[UltimoCarattereScritto]= L;
i= 1;
while i<== L {
        UltimoCarattereScritto= (UltimoCarattereScritto +1) mod N;
        PseudoPipe[UltimoCarattereScritto] =  OutMess[i]; i ++;
        }
BufferVuoto= 0
pthread_condition_signal(&SospesiInLettura);
pthread_mutex_unlock(&mux);
}

void function MessRead(&PseudoPipe, &InMess, &L)
{
pthread_mutex_lock(&mux)
while (BufferVuoto== 1) pthread_condition_wait(&SospesiInLettura,&mux);
L = PseudoPipe[PrimoCarattereDaLeggere];
i= 1;
while i<= L  {
        PrimoCarattereDaLeggere = (PrimoCarattereDaLeggere +1) mod N;
        InMess[i]= PseudoPipe[PrimoCarattereDaLeggere];  i++ ;
        }
if (PrimoCarattereDaLeggere== UltimoCarattereScritto)   BufferVuoto= 1;
PrimoCarattereDaLeggere = (PrimoCarattereDaLeggere +1) mod N
pthread_condition_signal(&SospesiInScrittura);
pthread_mutex_unlock(&mux);
}
```

ESERCIZIO A-2 (4 punti)

Un sistema con 5 processi (A, B, C, D, E) e risorse dei tipi R1, R2, R3, R4, rispettivamente di molteplicità [6, 6, 7, 3], utilizza l'algoritmo del banchiere per evitare lo stallo. Il sistema ha raggiunto lo stato (sicuro) mostrato nelle tabelle seguenti.

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

Esigenza Iniziale				
	R1	R2	R3	R4
A	1	1		1
B	3	4	2	3
C	2	2		2
D	1		3	2
E	2		3	3

Esigenza residua (dopo l'assegnazione attuale)				
	R1	R2	R3	R4
A	1	0	0	0
B	2	2	0	3
C	0	1	0	2
D	1	0	2	2
E	0	0	0	1

Molteplicità			
R1	R2	R3	R4
6	6	7	3

Disponibilità			
1	2	1	0

Si considerino ora i seguenti casi (in alternativa):

a. il processo A richiede un'istanza della risorsa R1
b. Il processo C richiede un'istanza della risorsa R2

In ognuno dei due casi, dire se la risorsa viene assegnata dal sistema operativo.

SOLUZIONE

a) Stato raggiunto dopo l'ipotetica assegnazione di un'istanza di R1 al processo A:

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

Esigenza Iniziale				
	R1	R2	R3	R4
A	1	1		1
B	3	4	2	3
C	2	2		2
D	1		3	2
E	2		3	3

Esigenza residua (dopo l'assegnazione attuale)				
	R1	R2	R3	R4
A	0	0	0	0
B	2	2	0	3
C	0	1	0	2
D	1	0	2	2
E	0	0	0	1

Molteplicità			
R1	R2	R3	R4
6	6	7	3

Disponibilità			
0	2	1	0

a1) Il processo A può terminare; la disponibilità di {R1, R2, R3, R4} diviene { 1,3,1,1}
a2) Il processo E può terminare ; la disponibilità di {R1, R2, R3, R4} diviene { 3,3,4,3}
a3) Il processo B può terminare; la disponibilità di {R1, R2, R3, R4} diviene { 4,5,6,3}
a4) Il processo C può terminare ; la disponibilità di {R1, R2, R3, R4} diviene { 6,6,6,3}
a5) Il processo D può terminare ; la disponibilità di {R1, R2, R3, R4} diviene {6,6,7,3}

Di conseguenza: risorsa assegnata? SI

b) Stato raggiunto dopo l'ipotetica assegnazione di un'istanza di R2 al processo C:

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

Esigenza Iniziale				
	R1	R2	R3	R4
A	1	1		1
B	3	4	2	3
C	2	2		2
D	1		3	2
E	2		3	3

Esigenza residua (dopo l'assegnazione attuale)				
	R1	R2	R3	R4
A	1	0	0	0
B	2	2	0	3
C	0	0	0	2
D	1	0	2	2
E	0	0	0	1

Molteplicità			
R1	R2	R3	R4
6	6	7	3

Disponibilità			
1	1	1	0

b1) Il processo A può terminare; la disponibilità di {R1, R2, R3, R4} diviene { 1,2,1,1}
b2) Il processo E può terminare; la disponibilità di {R1, R2, R3, R4} diviene { 3,2,4,3}
b3) Il processo B può terminare ; la disponibilità di {R1, R2, R3, R4} diviene { 4,4,6,3}
b4) Il processo C può terminare ; la disponibilità di {R1, R2, R3, R4} diviene { 6,6,6,3}
b5) Il processo D può terminare; la disponibilità di {R1, R2, R3, R4} diviene {6,6,7,3}

Di conseguenza: risorsa assegnata? SI

ESERCIZIO A-3 (3 punti)

In un sistema con thread *realizzati a livello del nucleo*, l'unità schedulabile dal kernel è il thread e lo schedulatore adotta la politica Round Robin e non prevede il preriascio. Inoltre il thread in esecuzione può rilasciare volontariamente il processore, invocando la chiamata *ThreadYield,* che lo fa passare in stato di pronto. I thread si sincronizzano con i meccanismi *pthread_condition_wait, pthread_condition_signal, pthread_mutex_lock e pthread_mutex_unlock* dello standard POSIX . Sono definite la variabile *Mux,* di tipo *mutex,* e la variabile *Cond1*, di tipo *condition*.
Al tempo *t* è in esecuzione il thread T11, la coda pronti contiene T21->T12 (T21 è in testa alla coda), e il thread T22 è sospeso sulla condizione *Cond1*. La variabile *Mux.,* ha valore 0 (mutex impegnato).
A partire dal tempo *t* si verifica la seguente sequenza di eventi:

t1: il thread in esecuzione esegue *pthread_mutex_lock*(*Mux*)
t2: scade il quanto di tempo
t3: il thread in esecuzione esegue *pthread_condition_signal(Cond1))*
t4: il thread in esecuzione esegue *pthread_condition_signal(Cond1)*
t5: il thread in esecuzione esegue *ThreadYield*
t6: il thread in esecuzione esegue *pthread_mutex_unlock* (*Mux*)

Si chiede di indicare il thread in esecuzione, il contenuto della coda pronti, il valore di *Mux* e i contenuti delle code di *Mux* e di *Cond1* subito dopo ogni evento.

SOLUZIONE

Tempo	Evento che si verifica	Situazione dopo l'evento				
		Thread in esecuzione	Coda pronti	Valore di *Mux*	Coda di *Mux*	Coda di *Cond1*
t1	il thread in esecuzione esegue *pthread_mutex_lock*(*Mux*)	T21	T12	0	T11	T22
t2	Scade il quanto di tempo	T12	T21	0	T11	T22
t3	Il thread in esecuzione esegue *pthread_condition_signal(Cond1)*	T12	T21 ->T22	0	T11	-
t4	Il thread in esecuzione esegue *pthread_condition_signal(Cond1)*	T12	T21 ->T22	0	T11	-
t5	Il thread in esecuzione esegue *ThreadYield*	T21	T22->T12	0	T11	-
t6	il thread in esecuzione esegue *pthread_mutex_unlock* (*Mux*)	T21	T22->T12-> T11	0	-	-

ESERCIZIO A-4 (2 punti)

Si consideri un sistema nel quale è definito il semaforo *sem1* e i processi P1, P2 e P3. Lo scheduler dei processi non prevede il prerilascio del processore.
Al tempo *T* è in esecuzione il processo P1 e il processo P2 è in stato di pronto. Allo stesso tempo, il semaforo *sem1* ha valore 0 e nella sua coda è inserito il processo P3.
Successivamente si verificano in successione i seguenti eventi:

a) P1 esegue *signal (Sem1)*
b) il processo in esecuzione esegue *wait(Sem1);*
c) termina il processo in esecuzione
d) il processo in esecuzione esegue *signal(Sem1);*

Si chiede quale processo è in esecuzione, il contenuto della coda pronti, il valore di *sem1* e il contenuto della sua coda immediatamente dopo ogni evento.

SOLUZIONE

Sequenza di eventi		In Esecuzione	Coda Pronti	Valore di Sem1	Coda di Sem1
a	P1 esegue *signal (Sem1)*	P1	P2 → P3	0	∅
b	Il processo in esecuzione esegue *wait(Sem1)*	P2	P3	0	P1
c	Termina il processo in esecuzione	P3	∅	0	P1
d	Il processo in esecuzione esegue *signal(Sem1)*	P3	P1	0	∅

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 P1;
c) il processo P1 esegue con successo una *exec;*
d) il processo P esegue l'operazione *close (fd[0]);*
e) il processo P esegue con successo una *fork,* generando il processo P2;
f) il processo P1 esegue con successo una *close (fd[1]);*

Si chiede se, dopo l'evento f):

- il processo P può eseguire l'operazione *read(fd[0] &buffer, NCaratteri* e/o *write(fd[1], &buffer, NCaratteri)?*
- il processo P1 può eseguire l'operazione *read(fd[0] &buffer, NCaratteri* e/o *write(fd[1], &buffer, NCaratteri)*?
- il processo P2 può eseguire l'operazione *read(fd[0] &buffer, NCaratteri* e/o *write(fd[1], &buffer, NCaratteri)?*

SOLUZIONE

PROCESSO	*read(fd[0] &buffer, NCaratteri)*	*write(fd[1], &buffer, NCaratteri)*
P	Eseguibile? NO	Eseguibile? SI
P1	Eseguibile? SI	Eseguibile? NO
P2	Eseguibile? NO	Eseguibile? SI

ESERCIZIO B-1 (4 punti)

Un sistema operativo gestisce la memoria con paginazione dinamica con pagine di 1 Kbyte e utilizza un file system di tipo FAT-16 con blocchi di 1 Kbyte e indici di blocco codificati con 16 bit.
Il disco che ospita il File System è organizzato nel modo seguente:

- il blocco di indice 0 contiene il *boot block;*
- il blocco di indice 1 contiene il *Super Block;*
- il blocco di indice 2 e i blocchi successivi, fino al blocco di indice *2+ LunghFAT-1* (dove *LunghFAT* è la lunghezza della FAT, espressa in numero di blocchi) contengono l'immagine stabile della FAT. Di conseguenza, il blocco *i* della FAT, con $0 \leq i < LunghFAT$, risiede nel blocco di indice $i+2$ del disco;
- i blocchi successivi, a partire dal blocco di indice *LunghFAT+2* , contengono i blocchi dati del File System.

I blocchi della FAT vengono caricati in memoria a domanda. L'accesso a un elemento della FAT contenuto in un blocco non caricato in memoria provoca un *PageFault.*
Si chiede:

1. Lo spazio occupato dalla FAT nel disco, in numero di byte e di blocchi;
2. Gli indici del primo e dell'ultimo blocco del disco ch esono occupati dalla FAT
3. Gli indici del primo e dell'ultimo blocco del disco che contengono blocchi dati del File System, supponendo che il disco abbia una capacità di 64Mbyte;

Supponendo che in questo File System sia definito il file `pippo`, che occupa 6 blocchi dati, e che i blocchi di questo file (con indici logici numerati da 0 a 5) siano allocati come segue:

Indice del blocco logico (nel file):	0	1	2	3	4	5
Indice *IndFis* del blocco (nel disco):	5637	9820	9823	315	716	10187

si chiede inoltre:

4. in quali blocchi della FAT sono contenuti gli indici dei blocchi dati del file `pippo`;
5. qual è il massimo numero di *PageFault* che si possono verificare per leggere in sequenza tutto il file.

SOLUZIONE

1. Spazio occupato dalla FAT nel disco: 2^{16} elementi → 2^{17} byte → $2^{17} / 2^{10} = 2^7 = 128$ blocchi
2. Indice del primo blocco del disco occupato dalla FAT: 2
   indice dell'ultimo blocco: 2+ 128 -1 = 129
3. Indice del primo blocco dati del File System: 130
   indice dell'ultimo blocco dati: $2^{16} - 1$
4. l'elemento della FAT associato al blocco dati di indice *IndFis* risiede su disco nel blocco *IndFis* /512 +2 e punta al blocco dati successivo. Pertanto:

Elemento della FAT di indice *IndFis* =	contenuto nel blocco della FAT	punta al blocco di indice:
5637	5637/512+ 2= 13	9820
9820	9820/512+ 2= 21	9823
9823	9820/512+ 2= 21	315
315	315/512+ 2= 2	716
716	315/512+ 2= 3	10187
10187	10187/512+ 2= 21	∅

Gli errori di pagina sono dovuti al riferimento di elementi della FAT contenuti in blocchi non caricati in memoria. Supponendo che, una volta caricati in memoria, i blocchi vi rimangano per un tempo abbastanza lungo, il massimo numero di *PageFault* che si possono verificare per leggere in sequenza tutto il file è 4: infatti per leggere l'intero file si devono caricare in memoria, nel caso peggiore, i blocchi 13, 21, 2 e 3della FAT.

ESERCIZIO B.2 (4 punti)

Un disco con 2 facce, 100 settori per traccia e 100 cilindri ha un tempo di seek proporzionale al numero di cilindri attraversati e pari a 1 ms per ogni cilindro. Il periodo di rotazione è di 10 msec: conseguentemente il tempo impiegato per percorrere un settore è di 0,1 msec. La politica di scheduling è la *shortest seek first.*
A un certo tempo (convenzionalmente indicato come t= 0) termina l'esecuzione dei comandi sul cilindro 44 e sono pendenti i seguenti comandi:

tempo	cilindro	settore	faccia
0	91	12	0
0	87	1	0
0	12	9	1

Successivamente pervengono, ai tempi indicati, le seguenti richieste di lettura o scrittura:

tempo	cilindro	settore	faccia
126	86	21	1
130	44	25	0
137	91	90	0

Calcolare il tempo necessario per eseguire tutte queste operazioni, tenendo presente che il tempo di esecuzione di ogni operazione è uguale alla somma dell'eventuale tempo di *seek,* del ritardo rotazionale (tempo necessario per raggiungere il settore indirizzato) e del tempo di percorrenza del settore indirizzato. Per il ritardo rotazionale dopo un'operazione di *seek* si assume sempre il valore di caso peggiore, pari a un periodo di rotazione (10 msec). Nel caso in cui ci siano due richieste da servire sullo stesso cilindro queste vengono servite in ordine FIFO.
Il controllore è dotato di sufficiente capacità di buffering ed è sempre in grado di accettare senza ritardo i dati letti dal disco o quelli da scrivere sul disco.

SOLUZIONE

op. su cilindro:	12	settore:	9						
inizio:	0	seek:	32	rotazione:	10	percorrenza:	0,1	fine:	42,1
op. su cilindro:	87	settore:	1						
inizio:	42,1	seek:	75	rotazione:	10	percorrenza:	0,1	fine:	127,2
op. su cilindro:	86	settore:	21						
inizio:	127,2	seek:	1	rotazione:	10	percorrenza:	0,1	fine:	138,3
op. su cilindro:	91	settore:	12						
inizio:	138,3	seek:	5	rotazione:	10	percorrenza:	0,1	fine:	153,4
op. su cilindro:	91	settore:	90						
inizio:	153,4	seek:	0	rotazione:	7,7	percorrenza:	0,1	fine:	161,2
op. su cilindro:	44	settore:	25						
inizio:	161,2	seek:	47	rotazione:	10	percorrenza:	0,1	fine:	218,3

ESERCIZIO B-3 (3 punti)

Un sistema operativo simile a UNIX gestisce la memoria con paginazione a domanda mediante il processo *PageDaemon* (con parametri *Lotsfree* e *MinFree*), che applica l'algoritmo di sostituzione *Second Chance.* Ad ogni intervento, il processo *PageDaemon* si comporta nel modo seguente:

- se *NumeroDiBlocchiLiberi* ≥ *MinFree* e *NumeroDiBlocchiLiberi* < *LotsFree*, applica ripetutamente l'algoritmo *Second Chance* e ogni volta scarica la pagina da questo selezionata come vittima, finchè *NumeroDiBlocchiLiberi* = *Lotsfree.* L'algoritmo *SecondChance* ignora i blocchi liberi e quelli assegnati al sistema operativo.
- se *NumeroDiBlocchiLiberi* < *MinFree* esegue lo swapout di uno o più processi, finchè diviene *NumeroDiBlocchiLiberi* ≥ *Lotsfree*. Per lo swapout dei processi si seleziona di volta in volta quello che ha il maggior numero di pagine caricate in memoria principale, e in caso di parità si segue l'ordine alfabetico.

Gli elementi della *CoreMap* hanno i campi *Proc* (processo a cui è assegnato il blocco; il campo è vuoto se il blocco è libero); *Pag* (pagina del processo caricata nel blocco), *Rif* (bit di pagina riferita, utilizzato da *Second Chance*). Al tempo *t* (quando sono presenti i processi A, B, C, D , la *Core Map* ha la configurazione mostrata in figura (non sono mostrati i blocchi assegnati al sistema operativo) e il puntatore dell'algoritmo di *SecondChance* è posizionato sul blocco 11) viene attivato il processo il *PageDaemon*.

Proc	D		A	A	A	B	C	A	B	A	B	B	C	D	D		D	B	A	B		C	D	C
Pag	0		5	10	12	7	0	1	0	2	9	6	3	1	2		6	2	7	3		7	11	2
Rif	1		0	0	1	1	0	1	0	0	1	1	1	1	1		1	1	1	0		1	1	0
Blocco	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Core Map al tempo *t*

Si chiede quali sono le pagine scaricate dal processo *PageDaemon* e la configurazione della *CoreMap* nelle seguenti ipotesi:
a) *PageDaemon* ha parametri *Lotsfree= 7* e *MinFre= 3;*
b) *PageDaemon* ha parametri *Lotsfree= 7* e *MinFre= 4.*

SOLUZIONE

Ipotesi a)
Si ha *NumeroDiBlocchiLiberi = MinFree* e *NumeroDiBlocchiLiberi* < *LotsFree* e pertanto *PageDaemon* applica ripetutamente l'algoritmo *Second Chance* per scaricare 4 pagine.

- Vengono scaricate le pagine B-3; C-2; A-5; A-10
- La configurazione finale della *CoreMap* è la seguente:

Proc	D				A	B	C	A	B	A	B	B	C	D	D		D	B	A			C	D	
Pag	0				12	7	0	1	0	2	9	6	3	1	2		6	2	7			7	11	
Rif	0				1	1	0	1	0	0	1	0	0	0	0		0	0	0			0	0	
Blocco	0	1	2	0	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Ipotesi b)
Si ha *NumeroDiBlocchiLiberi* < *MinFree* e pertanto *PageDaemon* esegue lo swapout del processo A.*cond Chance* per scaricare 4 pagine.

- Vengono scaricate le pagine A-5, A-10, A-12, A-1, A-2, A-7
- La configurazione finale della *CoreMap* è la seguente:

Proc	D					B	C		B		B	B	C	D	D		D	B		B		C	D	C
Pag	0					7	0		0		9	6	3	1	2		6	2		3		7	11	2
Rif	1					1	0		0		1	1	1	1	1		1	1		0		1	1	0
Blocco	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

ESERCIZIO B-4 (2 punti)

Un disco è organizzato con *NCilindri= 100* (numerati da 0 a 99), *NFacce=4* (numerate da 0 a 3) e *NSettori= 100* (numerati da 0 a 99). Ogni settore contiene 2^{10}= 1024 byte e corrisponde a un blocco.
A livello logico i blocchi sono individuati con *indici di blocco,* interi compresi nell'intervallo *[0, capacità_in_blocchi)* .
Gli indici di blocco sono definiti secondo la sequenza *cilindro, faccia, settore.*
Si chiede:

1) il numero di blocchi contenuti in ogni cilindro
2) la capacità del disco, in numero di blocchi e di byte
3) quanti cilindri vengono percorsi con l'operazione di *seek* necessaria per leggere il blocco di indice *10.000* dopo aver letto il blocco di indic*e 2000*

SOLUZIONE

1) Ogni cilindro contiene 4* 100 = 2^9 = 400 blocchi;
2) la capacità del disco è di 400*100 = 40.000 blocchi, pari a 40.000*2^{10} bite → 40.000/2^{10} ~ 39 Mbyte
3) il blocco di indice *2.000* è contenuto nel cilindro *2.000* ***div*** *400= 5;*

   il blocco di indice *10.000* è contenuto nel cilindro *10.000* ***div*** *400=25;*

   quindi con l'operazione di *seek* si percorrono 20 cilindri.

ESERCIZIO B-5 (2 punti)

In un file system UNIX si consideri il file *appunti,* creato dall'utente *Giovanni* nella directory */usr/Giovanni /SistemiOperativi.* L'utente *Cristina* ha creato un collegamento a questo file (con *hard link*) nella sua directory */usr/Cristina/Lezioni /SO,* con il nome *AppuntiDiGiovanni.* I diritti associati alle directory e ai file rilevanti sono i seguenti:

	owner (r w x)	*group (r w x)*	*others (r w x)*
SistemiOperativi	*1 1 1*	*1 0 1*	*1 0 0*
SistemiOperativi/appunti	*1 1 0*	*1 1 0*	*0 0 0*
Lezioni/SO	*1 1 1*	*1 0 1*	*1 0 0*
Lezioni/SO /AppuntiDiGiovanni	*1 1 0*	*1 1 0*	*0 0 0*

Gli utenti *Giovanni* e *Cristina* appartengono a medesimo gruppo.
Al tempo T *Giovanni* elimina il file *SistemiOperativi/appunti.*

1. L'utente *Giovanni* ha il diritto di eliminare il file *appunti*?
2. Dopo il tempo *T, Cristina* può leggere e scrivere il file *appunti* attraverso il link *AppuntiDiGiovanni?*
3. Dopo il tempo *T, Giovanni* può leggere e scrivere il file *appunti* attraverso il link *AppuntiDiGiovanni?*
4. Dopo il tempo *T, Giovanni* può eliminare il link *AppuntiDiGiovanni?*

SOLUZIONE

1. Al tempo *T*, *Giovanni* ha il diritto di eliminare il file *appunti*? SI

Dopo il tempo *T:*

2. *Cristina* può leggere e scrivere il file *appunti* attraverso il link *AppuntiDiGiovanni?* SI
3. *Giovanni* può leggere e scrivere il file *appunti* attraverso il link *AppuntiDiGiovanni?* SI
4. *Giovanni* può eliminare il link *AppuntiDiGiovanni?* NO