Quesito 1 (punti 5). Considerando i processi P1, P2, P3, P4, P5 e P6, in esecuzione su un elaboratore monoprocessore multiprogrammato, con l'ordine di arrivo e di esecuzione mostrato in figura 1, si determini quale/i tra le seguenti politiche di ordinamento senza uso di priorità esplicite possa(no) essere stata/e utilizzata/e:

1. First-In First-Out: indicare solo Sì o No
2. Shortest Job First (nella variante senza prerilascio): indicare solo Sì o No
3. Round Robin: in caso di risposta affermativa, indicare un'ampiezza di quanto temporale concordante con l'ordinamento mostrato in figura 1.

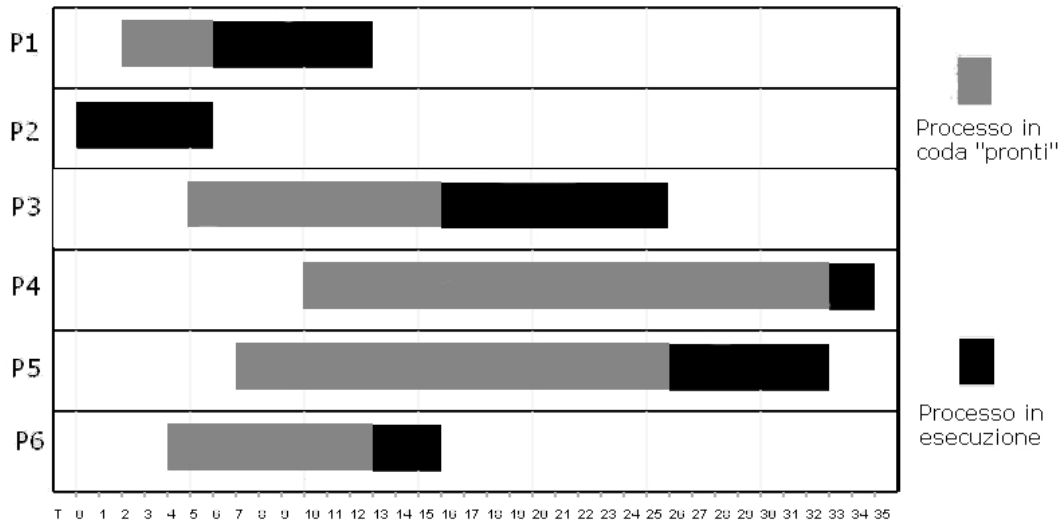

Figura 1: Ordine di arrivo e di esecuzione di sei processi su un elaboratore monoprocessore multiprogrammato.

Quesito 2 (punti 6). Cinque processi *batch*, identificati dalle lettere $A - E$ rispettivamente, arrivano all'elaboratore agli istanti $0, 2, 3, 7, 9$ rispettivamente. Tali processi hanno un tempo di esecuzione stimato di $2, 7, 3, 4, 1$ unità di tempo rispettivamente. Per ognuna delle seguenti politiche di ordinamento:

1. Round Robin (divisione di tempo, senza priorità e con quanto di tempo di ampiezza 3)
2. Shortest Job First (nella variante "*Shortest Remaining Time Next*", ovvero con prerilascio)

determinare, trascurando i ritardi dovuti allo scambio di contesto: (i) il tempo medio di risposta; (ii) il tempo medio di attesa; (iii) il tempo medio di *turn-around*. I risultati finali possono essere lasciati espressi come frazioni. Nel caso di arrivi simultanei di processi allo stato di pronto, si dia la precedenza ai processi usciti dallo stato di esecuzione rispetto a quelli appena arrivati. Nel caso della politica Round Robin si assuma inoltre che i processi appena arrivati al sistema vengano inseriti in fondo alla coda dei pronti.

Quesito 3 (punti 5). Dato un sistema di *swapping* e una memoria con zone disponibili di ampiezza: 10, 6, 16, 3, 7, 9, 11, 12 KB, in questo ordine, indicare quale area venga prescelta a fronte della richiesta di caricamento di un segmento di ampiezza 5 KB dopo aver caricato un segmento ampio 12 KB, considerando le seguenti politiche di rimpiazzo:

1. First Fit
2. Next Fit
3. Best Fit
4. Worst Fit

Quesito 4 (punti 8). Il programma in linguaggio C riportato in figura 2, utilizzabile in ambiente GNU/Linux, mostra come un processo utente possa creare sia un processo figlio (tramite la chiamata `fork()` alla linea 9) che un *thread* (tramite la chiamata `pthread_create(...)` alla linea 14). I flussi di controllo risultanti dall'esecuzione del programma condividono la variabile `value` inizializzata a 0 alla linea 3. Lo studente indichi quale valore tale variabile avrà quando sarà stampato alle linee 8, 13, 17, 21, illustrando i meccanismi di livello di sistema operativo che intervengono per produrre tale effetto.

```
 1: #include <pthread.h>
 2: #include <stdio.h>
 3: int value = 0;
 4: void *troublemaker(void *param);
 5: int main(int argc, char *argv[]) {
 6:     int pid; // id (intero) del processo creato da fork()
 7:     pthread_t tid; // id (strutturato) del thread creato da pthread_create(...)
 8:     printf("PARENT before: value = %d\n", value);
 9:     pid = fork();
10:      if (pid == 0) { // ramo del processo figlio
11:         /* il processo figlio crea un thread nel suo proprio ambiente
12:             e gli fa eseguire la procedura "troublemaker" */
13:          printf("CHILD before: value = %d\n", value);
14:          pthread_create(&tid,NULL,troublemaker,NULL);
15:          // il processo figlio attende il completamento del thread
16:          pthread_join(tid,NULL);
17:          printf("CHILD after: value = %d\n", value);}
18:      else if (pid > 0) { // ramo del processo padre
19:          // il processo padre aspetta la fine del processo figlio
20:          waitpid(pid,0,0);
21:          printf("PARENT after: value = %d\n", value);}
22: }
23: void *troublemaker(void *param) {
24:     value = 42;}
```

Figura 2: Codice sorgente del programma da analizzare.

Quesito 5 (punti 8). Un utente operante in ambiente GNU/Linux crea nel proprio spazio di lavoro due *file*, denominandoli rispettivamente "`./PARENT/pippo.txt`" e "`./PARENT/CHILD/pippo.txt`". Dopo aver scritto diverse quantità di testo nei due *file* usando il proprio editore preferito, su consiglio dell'amministratore del sistema, l'utente usa il comando `stat` per ottenere informazioni sui due *file* e sul *file system* che li ospita e le riporta diligentemente in tabella 1. Lo studente discuta il significato di tutte le voci ivi indicate e la correlazione tra loro e poi spieghi, con precisione e concisione, le ragioni per le quali l'editore di testo utilizzato non confonda i due *file* nonostante essi abbiano entrambi nome "`pippo.txt`".

Sul *file system*	
Namelength	255
Type	. . . ext3
Block size	4096
. . .	. . .
Blocks Total	13547642
Blocks Free	11640686
Blocks Available	10952502
Inodes Total	6884192
Inodes Free	6772305

Sui *file*		
	`./PARENT/pippo.txt`	`./PARENT/CHILD/pippo.txt`
Size	27	19
Blocks	16	8
Inode	2584677	2584676
Links	1	1
Access rights	. . .	. . .
Access time	2007-12-11 10:15:10. . . .	2007-12-11 10:15:25. . . .
Modify time	2007-12-11 10:15:07. . . .	2007-12-11 10:15:25. . . .
Change time	2007-12-11 10:15:10. . . .	2007-12-11 10:15:25. . . .

Tabella 1: Selezione di informazioni sui *file* e sul *file system* fornite dal comando `stat` nel caso in esame.

Soluzione 1 (punti 5).

1. FIFO: Sì
2. SJF: No
3. RR: Sì, con qualsiasi quanto temporale di ampiezza maggiore o uguale al massimo tempo di esecuzione fra i processi considerati, ovvero 10 u.t. (per P3).

Soluzione 2 (punti 6).

- RR (con quanto di tempo di ampiezza 3)

```
processo A  AA                    LEGENDA DEI SIMBOLI
processo B  --BBBbbbBBBbbbbB      - non ancora arrivato
processo C  ---ccCCC              x (minuscolo) attesa
processo D  -------ddddDDDddD     X (maiuscolo) esecuzione
processo E  ---------eeeeeE       . coda vuota

CPU         AABBBCCCBBBDDDEBD
coda        ...ccbbbdddeeebd.
            .......d.eebbbd..
```

processo	tempo di risposta	attesa	*turn-around*
A	0	0	0+2= 2
B	0	3+4= 7	7+7= 14
C	2	2	2+3= 5
D	4	4+2= 6	6+4= 10
E	5	5	5+1= 6
medie	2,20	4,00	7,40

- SJF_{SRTN} (ovvero con prerilascio)

```
processo A  AA                     LEGENDA DEI SIMBOLI
processo B  --BbbbBbbbbbBBBBB      - non ancora arrivato
processo C  ---CCC                 x (minuscolo) attesa
processo D  -------DDdDD           X (maiuscolo) esecuzione
processo E  ---------E             . coda vuota
02379
CPU         AABCCCBDEDDDBBBBB
coda        ...bbb.bdbbb.....
            ........b........
```

processo	tempo di risposta	attesa	*turn-around*
A	0	0	0+2= 2
B	0	3+5= 8	8+7= 15
C	0	0	0+3= 3
D	0	1	1+4= 5
E	0	0	0+1= 1
medie	0,00	1,80	5,20

Politica	Area scelta
First Fit	10 KB
Next Fit	7 KB
Best Fit	6 KB
Worst Fit	12 KB

Tabella 2: Riepilogo delle aree scelte dalle politiche indicate nel quesito.

Soluzione 3 (punti 5).

Soluzione 4 (punti 8). L'esecuzione del programma mostrato in figura 2 richiede l'intervento del sistema operativo a vari livelli. Esaminiamoli seguendo il numero d'ordine della linea di programma che li chiama in gioco, assumendo una memoria virtuale paginata.

linee 5 : questo è il punto di inizio del processo P risultante dall'invocazione dell'eseguibile prodotto dal comando di compilazione del programma (`gcc -g thread.c -o thread -lpthread`, dove "`thread.c`" è il nome arbitrario del *file* contenente il sorgente); in questo momento il S/O ha creato un singolo processo, assegnandogli la sua memoria virtuale in modalità "*paging-on-demand*"; naturalmente, trattandosi di un programma estremamente ridotto, tutti i segmenti del suo eseguibile sono agevolmente contenuti, separatamente, in una singola pagina; poiché siamo in ambiente GNU/Linux, al processo P è stato implicitamente assegnato un *thread* T_P che ne rappresenta il flusso di controllo

linee 6-8 : in questa fase è in esecuzione il solo *thread* T_P per conto del processo P; alla linea 8 T_P stampa su `stout` (il terminale di invocazione del programma) la stringa `PARENT before: value = 0`

linea 9 : l'invocazione di `fork()` duplica il processo P creando un clone identico C, ma non comprensivo dei *thread* interni, e in modalità *copy-on-write*; come per P viene creato implicitamente un *thread* T_{C_1} il cui PC (*program counter* viene posizionato alla stessa linea 9 del programma, al punto in cui la variabile `pid` viene assegnata; in virtù della condivisione *copy-on-write*, T_{c_1} legge una copia diversa della variabile `pid`, che vale 0 (un valore fittizio) per T_{c_1} e un valore positivo (vero) per T_P

linea 13 : qui è in esecuzione T_{C_1}, che legge la variabile `value` dalla copia di T_P e dunque stampa sullo stesso `stout` di T_P la stringa `CHILD before: value = 0`

linee 14-16 : T_{C_1} prima crea un nuovo *thread* T_{c_2}, con il quale condivide tutte le risorse fisiche e logiche, e lo incarica di eseguire la procedura `troublemaker` e poi si pone in attesa del suo completamento

linea 24 : questa è la sola istruzione eseguita da T_{C_2}, che opera sulla stessa variabile `value` di T_{C_1}, cui dunque viene assegnato il valore 42; in virtù della modalità di lavoro *copy-on-write* questa modifica non ha effetto sulla copia in possesso del processo P; T_{C_2} termina subito dopo l'assegnamento a `value`

linea 17 : quando T_{C_1} riprende l'esecuzione T_{C_2} è terminato e dunque `value` vale 42, ragion per cui T_{C_1} stampa su `stout` la stringa `CHILD after: value = 42` e poi termina

linee 20-21 : la terminazione di T_{C_1} comporta anche la terminazione del processo C che non ha altri *thread* in esecuzione; a questo punto T_P può riprendere l'esecuzione, non trovando alcuna modifica nella propria copia di `value`, stampando dunque su `stout` la stringa `PARENT after: value = 0`

Soluzione 5 (punti 8). Esaminiamo per prima la parte di tabella 1 che concerne il *file system*. Si tratta di un *file system* di tipo ext3, molto diffuso in ambiente GNU/Linux e, in quanto derivato del mondo UNIX, a struttura di allocazione a lista indicizzata basata su *i-node*. La dimensione del blocco è quella classica, ossia 4 KB. Conoscendo il numero totali di blocchi (per la cui indirizzazione servono 24 dei 32 *bit* della parola di architettura) possiamo facilmente derivare la dimensione della partizione, che è ampia 51,6 GB, occupata solo per il 14,1%. Risulta inoltre che 688.184 blocchi tra quelli liberi sono riservati dal Sistema Operativo per uso interno (per esempio per contenere puntatori a blocchi per gli *inode* con indici di prima, seconda e terza indirezione) e non sono dunque disponibili per i dati di utente. Il numero di *inode* rappresentabili nella partizione (6.884.192, esprimibile su 24 *bit*) deriviamo anche il massimo numero di *file* rappresentabili nella partizione (un *inode* per *file*). Dal numero di indici di *inode* liberi deriviamo anche che la partizione attualmente contiene 111.887 *file*. Passiamo ora alle considerazioni concernenti i due *file* creati dall'utente evocato nel quesito. Si tratta di due *file* piuttosto piccoli quanto a contenuto: il primo consta di soli 27 B e il secondo 19 B. È importante però notare che, in ragione dell'algoritmo di assegnazione noto come *Buddy algorithm*, di blocchi contigui aggregati in gruppi chiamati regioni, al primo sono

stati assegnati $16 = 2^4$ blocchi e al secondo $8 = 2^3$, in quantità notevolmente superiore al bisogno immediato, ma dimensionati sulle aspettivative presumibili di tipici *file* di utente. Ci viene fornito anche il numero d'ordine dell'*inode* corrispondente ai due *file*, che notiamo essere consecutivo. Questo può ragionevolmente suggerire che i due *file* siano stati creati l'uno appresso all'altro. Il campo informativo Links ci dice che entrambi i *file* hanno un solo collegamento diretto tra *directory* e *inode* corrispondente (ossia non sono attualmente riferiti da alcun *link* di tipo *hard*). Infine notiamo la differenza concettuale tra l'informazione relativa alla modifica, che concerne i dati utente del *file*, e al cambiamento, che invece concerne i metadati (che includono gli attributi e che ext3 salva prima nel *journal* e solo successivamente riflette nel *file* vero e proprio).
Per concludere, osserviamo che l'editore non può in alcun modo confondere i due *file* perché essi, essendo rappresentati da *inode* diversi, risultano al sistema come assolutamente diversi. Servirà infine osservare che la collocazione dei due *file* in *directory* diversi non ha alcuna influenza sulla garanzia di diversa identità tra essi, visto il *file system* in questione permette di creare *link* tra *file* ponendoli in posizioni qualunque nel *file system*.