

Quesito 1 (punti 8). Cinque processi *batch*, identificati dalle lettere $A - E$ rispettivamente, arrivano all'elaboratore agli istanti 0, 2, 3, 6, 10 rispettivamente. Tali processi hanno un tempo di esecuzione stimato di 3, 6, 4, 5, 1 unità di tempo rispettivamente. Per ognuna delle seguenti politiche di ordinamento: (1) RR: divisione di tempo, senza priorità¹ e con quanto di tempo di ampiezza 2; (2) RR: divisione di tempo, con priorità e prerilascio, e quanto di tempo di ampiezza 2; (3) SJF_{SRTN}: senza considerazione di valori di priorità espliciti² e 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*.

Ove la politica di ordinamento in esame consideri i valori di priorità, tali valori, mantenuti staticamente per l'intera durata dell'esecuzione, sono rispettivamente: 2, 3, 5, 3, 2 (con 5 valore maggiore). Nel caso di arrivi simultanei di processi allo stato di pronto, fatta salva l'eventuale considerazione del rispettivo valore di priorità, si dia la precedenza ai processi usciti dallo stato di esecuzione rispetto a quelli appena arrivati.

Quesito 2 (punti 8).

Dato il sistema descritto dalla seguente rappresentazione insiemistica delle assegnazioni di risorsa, dove \mathcal{P} denota l'insieme dei processi, \mathcal{R} l'insieme delle risorse R_i^j di indice i e molteplicità j , $\mathcal{E}(\mathcal{P})$ l'insieme delle richieste di accesso a risorse in \mathcal{R} emesse da processi in \mathcal{P} e attualmente pendenti, e $\mathcal{E}(\mathcal{R})$ l'insieme degli accessi attualmente soddisfatti di processi in \mathcal{P} a risorse in \mathcal{R} :

$$\mathcal{P} = \{P_1; P_2; P_3; P_4; P_5; P_6\} \quad (1)$$

$$\mathcal{R} = \{R_1^2; R_2^1; R_3^2; R_4^1\} \quad (2)$$

$$\mathcal{E}(\mathcal{P}) = \{P_2 \rightarrow R_2; P_3 \rightarrow R_4; P_4 \rightarrow R_1; P_5 \rightarrow R_4; P_6 \rightarrow R_3\} \quad (3)$$

$$\mathcal{E}(\mathcal{R}) = \{R_1 \rightarrow (P_1, P_2); R_2 \rightarrow P_3; R_3 \rightarrow (P_4, P_5); R_4 \rightarrow P_6\} \quad (4)$$

si analizzi il grafo di allocazione delle risorse, determinando se il sistema i trovi attualmente in situazione di stallo o meno. Successivamente, si studi l'evoluzione dello stato del sistema ove il processo P_1 richiedesse accesso alla risorsa R_2 a partire dalla situazione data.

Quesito 3 (punti 4).

[4.A]: Dato un sistema di *swapping* e una memoria con zone disponibili di ampiezza: 10, 4, 20, 18, 17, 9, 12, 15 KB, in questo ordine, indicare quale area venga prescelta dalla politica *next-fit* a fronte della richiesta di caricamento di un segmento di ampiezza 3 KB dopo aver caricato un segmento ampio 12 KB: (1) 4 KB; (2) 18 KB; (3) 20 KB; (4) 9 KB.

[4.B]: Data una partizione ampia 4 GB, con blocchi di ampiezza 4 KB, e contenente 128 K *file*, l'ampiezza della FAT dipende da: (1) il numero di *file* in essa rappresentati; (2) l'ampiezza dei blocchi; (3) l'ampiezza del disco in blocchi e l'ampiezza degli indici di blocco; (4) l'ampiezza del disco.

[4.C]: Sia data una memoria dotata di 4 *page frame*, inizialmente libere, e 8 pagine di memoria virtuale. Utilizzando la politica LRU per il rimpiazzo delle pagine, indicare quanti *page fault* si verifichino a fronte della stringa di riferimenti: 0172327103: (1) 4; (2) 5; (3) 6; (4) 7.

[4.D]: In un confronto prestazionale tra *hard link* (HL) e *symbolic link* (SL): (1) gli HL sono da preferire perchè velocizzano gli accessi ai *file*; (2) gli SL sono da preferire perchè assicurano la singolarità dell'associazione tra *directory* e *i-node*; (3) i due sono sostanzialmente indistinguibili; (4) gli SL hanno prestazioni superiori perchè impiegano meno spazio nella *directory*.

Quesito 4 (punti 4). Si consideri un sistema dotato di memoria virtuale, con memoria fisica divisa in 8 *page frame*, condivisa da 4 processi contemporaneamente attivi e denominati A, B, C e D rispettivamente. Si supponga che all'istante 100 lo stato della memoria sia quello riportato in tabella 1:

Si supponga che il sistema utilizzi un algoritmo di rimpiazzo delle pagine *second chance* (globale), e che la lista delle pagine accedute all'istante 100 sia: $\{ | C2(15,1) - D8(27,0) - B5(30,1) - A1(37,0) - A0(50,1) - C5(70,0) - C9(92,0) - B3(97,0) | \}$ con C2(15,1) primo elemento della lista; in tale elemento, C2 indica che si tratta della pagina logica 2 del processo C, mentre (15,1) indica che tale pagina è stata caricata in memoria all'istante 15 e che il suo *bit* di riferimento è uguale a 1.

Si considerino i due casi seguenti, eseguiti in alternativa all'istante 101: (1) C riferisce la pagina logica 5; (2) A riferisce la pagina logica 9.

Assumendo che, a parte quelle sopra elencate, non vi siano altre operazioni che modifichino i *bit* di riferimento, si scriva la lista delle pagine accedute aggiornata nei due casi dopo aver eseguito, alternativamente, le operazioni di cui ai punti 1 e 2.

¹Consequentemente con prerilascio dovuto solo a fine quanto.

²Esclusi ovviamente i valori di priorità impliciti determinati dalla durata (residua) dei processi.

processo	pag. logica	pag. fisica	istante di caricamento	bit di riferimento
A	0	7	50	1
A	1	6	37	0
B	5	5	30	1
B	3	4	97	0
C	2	0	15	1
C	5	2	70	0
C	9	1	92	0
D	8	3	27	0

Tabella 1: Stato della memoria all'istante 100.

Quesito 5 (punti 8). Il programma in linguaggio C riportato in figura 1, 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 1: Codice sorgente del programma da analizzare.