

COGNOME **NOME** **MATRICOLA** **CORSO |A| B|**

ESERCIZIO A-1 (4 punti)

Si considerino due thread linux A e B che scambiano messaggi attraverso un buffer condiviso a n posizioni, numerate da 0 a n-1. Ogni messaggio occupa una posizione del buffer. A tal fine A e B condividono un semaforo di mutua esclusione *mux* (inizializzato a 1), due variabili di condizione *NonPieno* e *NonVuoto* e una variabile *n_elem*, che rappresenta il numero di posizioni occupate nel buffer, e si sincronizzano con funzioni della libreria *pthreads*.

La parte significativa del codice di A e B è la seguente:

Thread A:

```
a.1) while (true) {
    a.2) <produce v>
    a.3) pthread_mutex_lock(&mux);
    a.4) if (n_elem == n-1) pthread_cond_wait(&NonPieno,&mux);
    a.5) <deposita v nel buffer>
    a.6) n_elem++;
    a.7) if (n_elem == 1) pthread_cond_signal(&NonVuoto);
    a.8) pthread_mutex_unlock(&mux);
}
```

Thread B:

```
b.1) while (true) {
    b.2) pthread_mutex_lock(&mux);
    b.3) if (n_elem == 0) pthread_cond_wait(&NonVuoto,&mux);
    b.4) <preleva un valore v dal buffer>
    b.5) n_elem--;
    b.6) if (n_elem == n-2) pthread_cond_signal(&NonPieno);
    b.7) pthread_mutex_unlock(&mux);
    b.8) <consuma v>
}
```

A un certo istante si ha *n_elem = n-1*, *mux= 1*, *NonVuoto= 1*, *NonPieno= 0*, e i due thread avanzano eseguendo la seguente sequenza di operazioni:

- 1) A esegue a.2 a.3;
- 2) B esegue b.2;
- 3) A esegue a.4
- 4) B esegue b.3, b.4, b.5, b.6;
- 5) B esegue b.7, b.8
- 6) A esegue a.5, a.6, a.7, a.8

Si chiede il valore di *mux* e lo stato (non sospeso, sospeso su *mux*, sospeso su *NonPieno*, sospeso su *NonVuoto*) dei thread A e B subito dopo 1), subito dopo 2), subito dopo 3), subito dopo 4), subito dopo 5) e subito dopo 6).

SOLUZIONE

	Valore di <i>mux</i>	Stato del thread A	Stato del thread B
Subito dopo 1)	0	Non sospeso	Non sospeso
Subito dopo 2)	0	Non sospeso	Sospeso su <i>mux</i>
Subito dopo 3)	0	Sospeso su <i>NonPieno</i>	Non sospeso
Subito dopo 4)	1	Non sospeso	Non sospeso
Subito dopo 5)	1	Non sospeso	Non sospeso
Subito dopo 6)	0	Non sospeso	Non sospeso

SISTEMI OPERATIVI, CORSI A e B - SESTO APPELLO - 14/9/2006

COGNOME NOME MATRICOLA CORSO |A| |B|

ESERCIZIO A-2 (4 punti)

Un sistema con processi A, B, C, D e risorse dei tipi R1, R2, R3, R4, rispettivamente di molteplicità [4, 4, 5, 5], ha raggiunto lo stato mostrato nelle tabelle seguenti. A partire da questo stato A richiede una risorsa di R2, B richiede una risorsa di R4, C richiede una risorsa di R4 e D richiede in sequenza due risorse di R3.

		MOLTEPLICITA' →				ESIGENZA ATTUALE			
ASSEGNAZIONE ATTUALE →		R1	R2	R3	R4	R1	R2	R3	R4
A	2	1	1	1		4	4	5	5
B	2	0	1	2					
C	0	1	0	0					
D	0	2	2	2					

DISPONIBILTA' ATTUALE		R1	R2	R3	R4	ESIGENZA ATTUALE			
		0	0	1	0				

Si chiede:

- 1) Dopo queste richieste, il sistema è in stallo?
- 2) Se si sopprime il processo A, qual'è lo stato raggiunto dal sistema immediatamente dopo la soppressione?
- 3) Questo stato è sicuro?

SOLUZIONE

- 1) Il sistema è in stallo.
- 2) Stato raggiunto dopo l'eliminazione del processo A:

		MOLTEPLICITA' →				ESIGENZA ATTUALE			
ASSEGNAZIONE ATTUALE →		R1	R2	R3	R4	R1	R2	R3	R4
A	-	-	-	-		4	4	5	5
B	2	0	1	2					
C	0	1	0	0					
D	0	2	2	2					

DISPONIBILTA' ATTUALE		R1	R2	R3	R4	ESIGENZA ATTUALE			
		2	1	2	1				

- 3) Lo stato raggiunto eliminando il processo A è sicuro. Infatti:
 - Il processo B può terminare
La disponibilità di {R1, R2, R3, R4} diviene { 4, 1 , 3 , 3 }
 - Il processo C può terminare
La disponibilità di {R1, R2, R3, R4} diviene { 4 , 2 , 3 , 3 }
 - Il processo D può terminare
La disponibilità di {R1, R2, R3, R4} diviene { 4 , 4 , 5 , 5 }

SISTEMI OPERATIVI, CORSI A e B - SESTO APPELLO - 14/9/2006

COGNOME NOME MATRICOLA CORSO **A** | **B**

ESERCIZIO A-3 (3 punti)

In un processo P sono definiti il thread MAIN , A e B , realizzati a livello utente con scheduling *round robin* e quanto di tempo di 5 msec. L'ordinamento dei thread ai fini dello scheduling è MAIN → A → B.

I thread A e B che cooperano scambiando messaggi attraverso un buffer condiviso, capace di contenere un solo messaggio. Le parti rilevanti del codice dei thread A e B sono le seguenti:

Thread A:

```
a.1) while (not fine) {  
a.2)     <produce mess>  
a.3)     lock(&BufferVuoto);  
a.4)     <deposita mess nel buffer>  
a.5)     void fun();  
a.6)     unlock(&BufferPieno);  
}
```

Thread B:

```
b.1) while (not fine) {  
b.2)     lock(&BufferPieno);  
b.3)     <preleva mess dal buffer>  
b.4)     unlock(&BufferVuoto);  
b.5)     <consuma mess>  
}
```

dove BufferPieno, BufferVuoto e fine sono variabili binarie condivise.

Al tempo T si ha BufferPieno= 1, BufferVuoto= 0 e fine= 0 ed è in esecuzione il thread A che esegue la riga a.1). Il thread B è pronto a eseguire la riga b.1). Il tempo di esecuzione di tutte le righe di codice di A e B è trascurabile, ad eccezione della riga a..5) che richiede 8 msec.

S chiede il tempo necessario ad A per depositare 2 messaggi, a partire dal tempo T. Si suppone che la variabile *fine* conservi il valore 0 e che il thread MAIN consumi sempre per intero il suo quanto di tempo.

SOLUZIONE

1) Primo messaggio depositato al tempo T+ 15 . Infatti

- Thread A esegue a.1), a.2) e poi a.3) per 5 msec;
- Thread B esegue b.1), b.2), b.3), b.4), b.5), b.1), b.2) per 5 msec
- Thread MAIN utilizza il processore per 5 msec;
- Thread A esegue a.3) e a.4) e deposita mess

2) Secondo messaggio depositato al tempo T+ 15 + 30

- Thread A esegue a.5) per 5 msec);
- Thread B esegue b.2 per 5 msec;
- Thread MAIN utilizza il processore per 5 msec;
- Thread A esegue a.5) per 3 msec e quindi a.6), a.1), a.2) e poi a.3) per 2 msec
- Thread B esegue b.2), b.3), b.4), b.5, b.1), b.2) per 5 msec
- Thread MAIN utilizza il processore per 5 msec;
- Thread A esegue a.3) e a.4) e deposita mess

SISTEMI OPERATIVI, CORSI A e B - SESTO APPELLO - 14/9/2006

COGNOME NOME MATRICOLA CORSO **A** **B**

ESERCIZIO A-4 (2 punti)

Nel sistema UNIX, il processo P genera un figlio eseguendo il seguente frammento di codice:

```
...
printf("uno ");
a = fork();
if (a>0)printf("due ");
else
    if (a==0) {
        printf("tre ");
        execl("/bin/pippo",NULL);
    }
    else printf("quattro ");
printf("cinque ");
```

Che cosa stampano per effetto di questo codice il processo padre e il processo figlio se la *fork* e la *exec* sono eseguite con successo?

SOLUZIONE

Il processo P stampa “uno”, “due”, “cinque”;
Il processo figlio stampa “tre”.

ESERCIZIO A-5 (2 punti)

Si consideri un sistema operativo con thread realizzati a livello kernel, dove l’unità di schedulazione è il thread. Quali dei seguenti dati sono contenuti nel descrittore di processo e quali nel descrittore di thread?

Dato	Descr. Processo	Descr Thread
Immagine del contatore di programma	NO	SI
Puntatore alle aree dati in memoria	SI	NO
Immagine dei registri del processore	NO	SI
Stato (pronto/esecuzione/attesa)	NO	SI
Puntatore alla tabella dei file aperti	SI	NO
Puntatore allo stack	NO	SI

SISTEMI OPERATIVI, CORSI A e B - SESTO APPELLO - 14/9/2006

COGNOME NOME MATRICOLA CORSO **A** | **B**

ESERCIZIO B-1 (4 PUNTI)

In un sistema che gestisce la memoria con partizioni variabili, la memoria ha dimensione 40 e il sistema operativo occupa una partizione con origine 0 e lunghezza 6. Al tempo 0 sono stati generati e caricati in memoria i seguenti processi:

- il processo A, caricato nella partizione con origine 6 e lunghezza 5;
- il processo B, caricato nella partizione con origine 13 e lunghezza 7;
- il processo C, caricato nella partizione con origine 25 e lunghezza 3;
- il processo D, caricato nella partizione con origine 28 e lunghezza 8;

Successivamente:

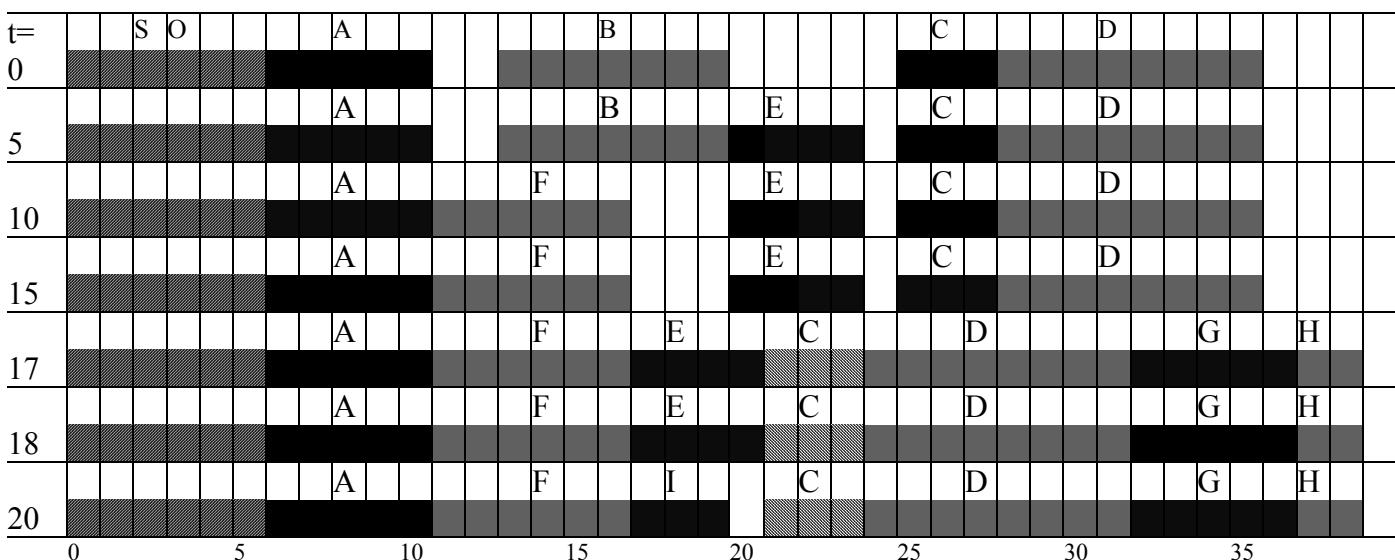
- al tempo 5 viene generato il processo E, che richiede una partizione di lunghezza 4;
- al tempo 7 viene generato il processo F, che richiede una partizione di lunghezza 6;
- al tempo 10 termina il processo B;
- al tempo 15 viene generato il processo G, che richiede una partizione di lunghezza 5;
- al tempo 17 viene generato il processo H, che richiede una partizione di lunghezza 2;
- al tempo 18 viene generato il processo I, che richiede una partizione di lunghezza 3;
- al tempo 20 termina il processo E.

Il codice dei processi è rilocabile. Per l'assegnazione delle partizioni disponibili si adotta la politica *first-fit*.

Il caricamento dei processi avviene con politica FIFO; quelli in attesa di caricamento sono inseriti in una coda. Quando il numero di processi in attesa di caricamento diviene uguale a 2, si esegue il compattamento e si riconsidera la possibilità di eseguire il caricamento.

Utilizzando il grafico sotto riportato, mostrare come evolve l'occupazione della memoria fino al tempo 20. Mostrare anche la composizione della coda dei processi in attesa di caricamento.

SOLUZIONE



Coda dei processi in attesa di caricamento:

- t= 0: Ø
- t= 5: Ø
- t= 7: F
- t= 10: Ø
- t= 15: G
- t= 17: G → H ; dopo il compattamento Ø
- t= 18: I
- t= 20: Ø

SISTEMI OPERATIVI, CORSI A e B - SESTO APPELLO - 14/9/2006

COGNOME NOME MATRICOLA CORSO **A** | **B**

ESERCIZIO B-2 (4 punti)

In un sistema che gestisce la memoria con paginazione, sono presenti i processi A, B e C.

A un certo istante le tabelle delle pagine dei tre processi sono le seguenti (con notazione evidente; in parentesi è indicato il tempo dell'ultimo riferimento). I tempi sono intesi come virtuali: per ogni processo il tempo avanza solo quando il processo è in esecuzione.

Pagina	Blocco
0	-
1	6 (2)
2	-
3	-
4	-
5	8 (9)
6	-
7	10 (4)

Processo A

Pagina	Blocco
0	7 (3)
1	-
2	-
3	-
4	5 (9)
5	-
6	-
7	11 (5)

Processo B

Pagina	Blocco
0	9 (6)
1	1 (7)
2	-
3	12 (0)
4	2 (9)
5	-
6	-
7	-

Processo C

Per la gestione della memoria si utilizza un algoritmo di sostituzione LRU locale. Il working set (inteso come insieme di blocchi a disposizione del processo, anche se momentaneamente non occupati) dei processi è il seguente:

- processo A: {6,8,10}
- processo B: {4, 5, 7,11}
- processo C: {1, 2, 9,12}

I working set si possono espandere e contrarre: si espandono con l'assegnazione di un ulteriore blocco se il processo commette tre errori di pagina consecutivi e si contraggono con la sottrazione di un blocco se la distanza passata della pagina contenuta in quel blocco supera il valore 10. Per l'espansione dei working set si attinge a una coda FIFO di blocchi disponibili, che inizialmente contiene i blocchi 3 → 14 → 15. I blocchi sottratti ai working set vengono inseriti in questa coda.

I processi avanzano con le seguenti modalità:

- Inizialmente avanza il processo A, che a partire dal proprio tempo virtuale 10 riferisce le pagine 0, 4, 6 e 5 rispettivamente ai tempi virtuali 10,11,12 e 13;
- Successivamente avanza il processo B, che a partire dal proprio tempo virtuale 10 riferisce le pagine 4, 2, 0 e 6 rispettivamente ai tempi virtuali 10,11,12 e 13;
- Infine avanza il processo C, che a partire dal proprio tempo virtuale 10 riferisce le pagine 1, 6, 3 e 1 rispettivamente ai tempi virtuali 10,11,12 e 13.

Mostrare come si modificano le tabelle delle pagine dei tre processi ai rispettivi tempi virtuali 10, 11, 12 e 13.

SOLUZIONE

Processo A

Pagina	Blocco
0	6(10)
1	-
2	-
3	-
4	-
5	8 (9)
6	-
7	10 (4)

t= 10

Pagina	Blocco
0	6(10)
1	-
2	-
3	-
4	10(11)
5	8 (9)
6	-
7	-

t= 11

Pagina	Blocco
0	6(10)
1	-
2	-
3	-
4	10(11)
5	8 (9)
6	3(12)
7	-

t= 12

Pagina	Blocco
0	6(10)
1	-
2	-
3	-
4	10(11)
5	8 (13)
6	3(12)
7	-

t= 13

Processo B

Pagina	Blocco
0	7 (3)
1	-
2	-
3	-
4	5 (10)
5	-
6	-

Pagina	Blocco
0	7 (3)
1	-
2	4(11)
3	-
4	5 (10)
5	-
6	-

t= 11

Pagina	Blocco
0	7 (12)
1	-
2	4(11)
3	-
4	5 (10)
5	-
6	-

Pagina	Blocco
0	7 (12)
1	-
2	4(11)
3	-
4	5 (10)
5	-
6	11(13)

SISTEMI OPERATIVI, CORSI A e B - SESTO APPELLO - 14/9/2006

COGNOME **NOME** **MATRICOLA** **CORSO |A| |B|**

7	11 (5)
t= 10	

7	11 (5)
t= 11	

7	11 (5)
t= 12	

7	-
t= 13	

Processo C

Pagina	Blocco
0	9 (6)
1	1 (10)
2	-
3	-
4	2 (9)
5	-
6	-
7	-

t= 10

Pagina	Blocco
0	-
1	1 (10)
2	-
3	-
4	2 (9)
5	-
6	9 (11)
7	-

t= 11

Pagina	Blocco
0	-
1	1 (10)
2	-
3	2 (12)
4	-
5	-
6	9 (11)
7	-

t= 12

Pagina	Blocco
0	-
1	1 (13)
2	-
3	2 (12)
4	-
5	-
6	9 (11)
7	-

t= 13

SISTEMI OPERATIVI, CORSI A e B - SESTO APPELLO - 14/9/2006

COGNOME NOME MATRICOLA CORSO |A| |B|

ESERCIZIO B-3 (3 PUNTI)

Un disco RAID di livello 4 è composto da 5 dischi fisici, numerati da 0 a 4. 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 $b \bmod 4$ del disco fisico di indice $b \bmod 4$. 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.

Il gestore del disco virtuale accetta comandi (di lettura o scrittura) che interessano k blocchi ($k \geq 1$) consecutivi del disco virtuale. Il comando di lettura ha il formato $\text{read}(buffer, PrimoBlocco, NumeroBlocchi)$, dove $buffer$ è la destinazione dei dati. A un certo tempo viene eseguita l'operazione $\text{read}(buffer, 13, 5)$, che viene realizzata leggendo il blocco 3 e il blocco 4 dei 5 dischi fisici.

Per alcuni malfunzionamenti, la lettura del blocco 3 del disco fisico 4 e quella del blocco 4 del disco fisico 0 falliscono. Il fallimento viene riconosciuto e segnalato dal gestore, che può ricostruire i blocchi mancanti.

Il contenuto dei blocchi letti con successo dai dischi fisici è il seguente:

	Blocco 3							
Disco fisico 0	0	1	0	0	1	1	0	1
Disco fisico 1	1	1	1	0	1	1	0	0
Disco fisico 2	0	1	1	0	1	0	0	1
Disco fisico 3	0	1	1	1	1	0	0	1
Disco fisico 4								

	Blocco 4							
	1	1	1	0	0	1	1	0
	1	1	0	1	0	0	1	1
	0	0	1	0	1	1	1	1
	1	0	0	1	1	0	1	1

Si chiede:

1. la posizione sui dischi fisici dei blocchi effettivamente interessati all'operazione di lettura;
2. Il valore ricostruito dei blocchi la cui lettura è fallita.

Se successivamente si esegue l'operazione $\text{write}(buffer, 17, 1)$, dove il contenuto di $buffer$ è 01100010, si chiede inoltre:

3. la posizione sui dischi fisici del blocco da scrivere;
4. quale altro blocco deve essere modificato sui dischi fisici e qual è il suo nuovo contenuto.

SOLUZIONE

1) e 2) Posizione dei blocchi da leggere sui dischi fisici e contenuti ricostruiti

	Blocco 3							
Disco fisico 0	0	1	0	0	1	1	0	1
Disco fisico 1	1	1	1	0	1	1	0	0
Disco fisico 2	0	1	1	0	1	0	0	1
Disco fisico 3	0	1	1	1	1	0	0	1
Disco fisico 4	1	0	1	1	0	0	0	1

	Blocco 4							
	1	0	0	0	0	0	0	1
	1	1	1	0	0	1	1	0
	1	1	0	1	0	0	1	1
	0	0	1	0	1	1	1	1
	1	0	0	1	1	0	1	1

3) e 4) Posizione del blocco da scrivere e contenuti modificati:

	Blocco 3							
Disco fisico 0	0	1	0	0	1	1	0	1
Disco fisico 1	1	1	1	0	1	1	0	0
Disco fisico 2	0	1	1	0	1	0	0	1
Disco fisico 3	0	1	1	1	1	0	0	1
Disco fisico 4	1	0	1	1	0	0	0	1

	Blocco 4							
	1	0	0	0	0	0	0	1
	0	1	1	0	0	0	1	0
	1	1	0	1	0	0	1	1
	0	0	1	0	1	1	1	1
	0	0	0	1	1	1	1	1

SISTEMI OPERATIVI, CORSI A e B - SESTO APPELLO - 14/9/2006

COGNOME NOME MATRICOLA CORSO **A** | **B**

ESERCIZIO B-4 (2 PUNTI)

In un sistema che gestisce la memoria con rilocazione dinamica, spazio logico segmentato (con segmento codice e segmento dati) e caricamento in partizioni variabili, i registri base e limite del segmento codice hanno a un certo istante i valori $B_1=15.000$ e $L_1=35.000$, mentre i registri base e limite del segmento dati hanno i valori $B_2=55.000$ e $L_2=20.000$.

Supponendo che il processo in esecuzione estragga istruzioni con indirizzi logici 7.000, 37.000 e 25.000, e riferisca dati con indirizzi logici 45.000, 10.000 e 60.000, dire se ognuno dei precedenti indirizzi logici è legittimo e, in caso affermativo, calcolare il corrispondente indirizzo fisico.

SOLUZIONE

Indirizzo logico	Legittimo?	Indirizzo fisico
7.000	SI	$15.000 + 7.000 = 22.000$
37.000	NO	
25.000	SI	$15.000 + 25.000 = 40.000$
45.000	NO	
10.000	SI	$55.000 + 10.000 = 65.000$
60.000	NO	

ESERCIZIO B-5 (2 PUNTI)

In un file system UNIX dove ogni i-node occupa 1 blocco, si consideri il file */usr/tizio/esercizi/filesystem5*

Supponendo che ogni directory di questo path occupi 1 blocco e che la directory radice sia caricata in memoria, mentre tutti gli altri i-node e tutte le directory interessate risiedono su disco, calcolare il numero di accessi al disco necessari per leggere i primi due blocchi del file considerato.

SOLUZIONE

1. 1 accesso per leggere lo i-node di *usr*
2. 1 accesso per leggere la directory *usr*
3. 1 accesso per leggere lo i-node di *tizio*
4. 1 accesso per leggere la directory *tizio*
5. 1 accesso per leggere lo i-node di *esercizi*
6. 1 accesso per leggere la directory *esercizi*
7. 1 accesso per leggere lo i-node di *filesystem5*
8. 1 accesso per leggere il primo blocco di *filesystem5*
9. 1 accesso per leggere il secondo blocco di *filesystem5*

In totale : 9 accessi.