Politecnico di Milano
Dipartimento di Elettronica e Informazione

prof.ssa Anna Antola
prof. Luca Breveglieri
prof. Giuseppe Pelagatti
prof.ssa Donatella Sciuto
prof.ssa Cristina Silvano

AXO – Architettura dei Calcolatori e Sistemi Operativi

Prova di martedì 26 novembre 2013

Cognome _____ **Nome** _____

Matricola _____ **Firma** _____

Istruzioni

- Si scriva solo negli spazi previsti sui fogli distribuiti e per la minuta si utilizzino le pagine bianche in fondo.
- Non si separino i fogli tranne eventualmente l'ultimo per la minuta, che va consegnato anche se staccato intestandolo con nome e cognome.
- È vietato portare con sé libri, eserciziari e appunti, nonché cellulari e altri dispositivi mobili di calcolo o comunicazione. Chiunque fosse trovato in possesso di documentazione relativa al corso – anche se non strettamente attinente alle domande proposte – vedrà annullata la propria prova.
- Non è possibile lasciare l'aula conservando il tema della prova in corso.
- Tempo a disposizione: 2 h : 30 m

Valore indicativo di domande ed esercizi, voti parziali e voto finale:

esercizio 1 (4 punti) _____

esercizio 2 (4 punti) _____

esercizio 3 (4 punti) _____

esercizio 4 (2 punti) _____

esercizio 5 (1 punti) _____

esercizio 6 (1 punti) _____

voto finale: (16 punti) _____

I NUMERI INDICANO I PUNTEGGI APPROSSIMATIVI.

CON SOLUZIONI (in corsivo)

esercizio n. 1 – modello thread e parallelismo

Si consideri il programma C seguente (gli "#include" sono omessi):

```
pthread_mutex_t time, space;
sem_t open, close;
int global = 0;
```

```
void * one (void * arg) {
   pthread_mutex_lock (&time);
   sem_post (&open);                                   /* statement A */
   pthread_mutex_lock (&space);
   global = 1;
   pthread_mutex_unlock (&time);
   pthread_mutex_unlock (&space);
   return (void *) 1;
} /* end one */
```

```
void * two (void * arg) {
   pthread_mutex_lock (&time);
   global = 2;
   sem_wait (&open);                                   /* statement B */
   pthread_mutex_unlock (&time);
   sem_wait (&close);
   return NULL;
} /* end two */
```

```
void * three (void * arg) {
   pthread_mutex_lock (&space);
   sem_wait (&close);
   global = 3;
   pthread_mutex_unlock (&space);
   sem_post (&close);                                  /* statement C */
   return NULL;
} /* end two */
```

```
void main ( ) {
   pthread_t th_1, th_2, th_3;
   sem_init (&open, 0, 0);
   sem_init (&close, 0, 1);
   pthread_create (&th_2, NULL, two, NULL);
   pthread_create (&th_1, NULL, one, NULL);
   pthread_create (&th_3, NULL, three, NULL);
   pthread_join (th_3, NULL);
   pthread_join (th_1, &global);                       /* statement D */
   pthread_join (th_2, NULL);
return;
} /* end main */
```

Si completi la tabella qui sotto **indicando lo stato di esistenza del thread** nell'istante di tempo specificato da ciascuna condizione, così: se il thread **esiste**, si scriva ESISTE; se **non esiste**, si scriva NON ESISTE; e se può essere **esistente o inesistente**, si scriva PUÒ ESISTERE. Ogni casella della tabella va riempita in uno dei tre modi (non va lasciata vuota).

Si badi bene alla colonna "condizione": con "subito dopo statement X" si chiede lo stato che il thread assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**thread**		
	th_1	**th_2**	**th_3**
subito dopo stat. **A**	*ESISTE*	*ESISTE*	*PUÒ ESISTERE*
subito dopo stat. **B**	*PUÒ ESISTERE*	*ESISTE*	*PUÒ ESISTERE*
subito dopo stat. **D**	*NON ESISTE*	*PUÒ ESISTERE*	*NON ESISTE*

Si completi la tabella qui sotto, **indicando i valori delle variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha un valore definito; oppure X quando è indefinita
- se la variabile può avere due o più valori, li si riporti tutti quanti
- il semaforo può avere valore positivo o nullo (non valore negativo)

Si badi bene alla colonna "condizione": con "subito dopo statement X" si chiede il valore (o i valori) che la variabile ha tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**variabili globali**	
	open	*global*
subito dopo stat. **A**	*1*	*0 / 3*
subito dopo stat. **B**	*0*	*1 / 2 / 3*
subito dopo stat. **C**	*0 / 1*	*1 / 2 / 3*
subito dopo stat. **D**	*0 / 1*	*1 / 2*

Il sistema può andare in stallo (deadlock), con uno o più thread che si bloccano, in due casi diversi (con deadlock si intende anche un blocco dovuto a un solo processo che non potrà mai proseguire). Si indichino gli statement dove avvengono i blocchi, con il valore (o i valori) della variabile `global`:

caso	**th_1**	**th_2**	**th_3**	*global*
1	`lock time`	`wait open`	-	*2 / 3*
2	-	-	`wait close`	*1 / 2*

esercizio n. 2 – nucleo e commutazione tra processi

Si considerino i seguenti frammenti di programma:

```
/* programma CODICE_X.c */
main ( ) {
   ...

   pid1 = fork ( );
   if (pid1 == 0) {
      /* codice eseguito dal figlio Q                          */
      receive (coda1, temp, 25);
      write (stdout, OK_msg, 20);
      fd = open ("/file1", O_RDWR); /* 2 blocchi, lettura */
      exit (2);
   } else {
      /* codice eseguito dal padre P                           */
      send (coda1, vett, 25);
      pid1 = wait (&status);
      exit (0);
   } /* end if */
} /* CODICE_X */

/* programma CODICE_Y.c */
main ( ) {
   /* codice eseguito da R                                     */
     ...
     sleep (1);
     receive (coda2, temp, 10);
     exit (1);
   } /* end if */
} /* CODICE_Y */
```

Si considerino i seguenti processi nel sistema:

- è dato un processo **P** che esegue il programma **CODICE_X**; il processo **P** crea un processo figlio **Q**
- un processo **R,** che non è figlio né di **P** né di **Q**, viene creato in un certo momento ed esegue il programma **CODICE_Y**

Non ci sono altri processi nel sistema.

Ulteriori specifiche del sistema:

1. i **processi utente** hanno associata una **priorità**, il processo "idle" ha priorità minima e non sono da considerare altri processi nel sistema
2. le priorità relative a **tutti i processi di interesse attivi nel sistema** sono indicate – quando necessario – nella tabella di commutazione dei processi da completare
3. il **buffer del driver** di standard output ha dimensione di **50 caratteri**
4. per **tutte le operazioni** su file che implicano trasferimento di blocchi, il numero di blocchi da trasferire e la direzione del traferimento sono specificati come commento nel codice
5. le chiamate di sistema ***wait*** e ***waitpid*** invocano la funzione "Sleep_on" su un evento opportuno
6. per completare la tabella delle commutazioni, si faccia riferimento alla notazione vista a lezione
7. per le chiamate di sistema ***send*** e ***receive*** presenti nel codice, si considerino le seguenti ulteriori specifiche di funzionamento

   a) ***send*** *(coda, var, n)* – spedisce caratteri a una coda. La chiamata copia *n* caratteri – prelevati dalla variabile locale *var* del processo che la invoca – nella variabile condivisa *coda*. Se c'è un processo in attesa di ricevere *n* caratteri da quella coda, il processo viene risvegliato con una "Wake_up". Si noti che la ***send*** non richiede mai di porre il processo che la invoca nello stato di attesa.

   b) ***receive*** *(coda, var, n)* – riceve caratteri da una coda. Se nella variabile condivisa *coda* sono presenti almeno *n* caratteri, esattamente *n* caratteri vengono copiati nella variabile locale *var* del processo che invoca la ***receive***. Se questa condizione non è verificata, il processo viene posto in attesa tramite "Sleep_on" su un evento opportuno.

DOMANDA

Si completino le parti mancanti della tabella di commutazione dei processi riportata alle due pagine seguenti.

AVVERTENZE

Nella tabella di commutazione proposta sono previste righe da completare, dove:

a) è **specificato l'evento** (con informazioni aggiuntive); in questo caso sono da completare le parti relative allo stato dei processi e – se richiesto – ai moduli del SO e al contesto

b) è **specificato lo stato dei processi raggiunto dopo il verificarsi dell'evento**; in questo caso sono da completare i campi relativi all' evento (con eventuali informazioni aggiuntive) e – se richiesto – ai moduli del SO e al contesto

NOTA BENE:

- l'**evento** in questione è sempre **determinabile univocamente** dallo stato raggiunto, dall'evoluzione precedente dei processi, dal codice dei programmi e dalle ulteriori specifiche di sistema
- se l'**evento** è un **interrupt** è obbligatorio usare la notazione "*n* interrupt" **indicando esattamente il numero di interruzioni che si sono verificate**

evento (è preceduto dal processo nel cui contesto l'evento si verifica)	**informazioni aggiuntive**	**moduli del Sistema Operativo eseguiti per gestire l'evento**	**processo/i nel cui contesto è eseguito ogni modulo**	**stato dei processi dopo la gestione dell'evento**		
				P	**Q**	**R**
				esec	**non esiste**	**non esiste**
P: pid1 = fork	il quanto di tempo di P è scaduto	*G_SVC_1* *fork* *G_SVC_2* *Preempt_1* *Change* *(fork)* *G_SVC_23*	*P* *P* *P* *P* *P – Q* *Q* *Q*	*pronto*	*esec*	*non esiste*
Q: receive		*G_SVC_1* *receive* *Sleep_on_1 (E1)* *Change* *Preempt_2* *G_SVC_3*	*Q* *Q* *Q* *Q – P* *P* *P*	*esec*	*attesa (E1)*	*non esiste*
P: interrupt da real time clock	R in stato di pronto dopo fork opportuna il quanto di tempo di P è scaduto **P < R**	*R_int(CK)_1/2* *Preempt_1* *Change* *G_SVC_3*	*P* *P* *P – R* *R*	*pronto*	*attesa (E1)*	*esec*
R: sleep (1)		*G_SVC_1* *sleep* *Sleep_on_1 (E2)* *Change* *Preempt_2* *R_int_3 (CK)*	*R* *R* *R* *R – P* *P* *P*	*esec*	*attesa (E1)*	*attesa (E2)*
P: send	**Q < P**	*G_SVC_1* *send* *Wake-up (E1)* *send* *G_SVC_23*	*P* *P* *P* *P*	*esec*	*pronto*	*attesa (E2)*
P: wait		*G_SVC_1* *wait* *Sleep_on_1 (E3)* *Change* *Sleep_on_2 (E1)* *receive* *G_SVC_23*	*P* *P* *P* *P – Q* *Q* *Q* *Q*	attesa (E3)	esec	attesa (E2)

TABELLA DI COMMUTAZIONE DEI PROCESSI *(continua)*

evento (è preceduto dal processo nel cui contesto l'evento si verifica)	**informazioni aggiuntive**	**moduli del Sistema Operativo eseguiti per gestire l'evento**	**processo/i nel cui contesto è eseguito ogni modulo**	**stato dei processi dopo la gestione dell'evento**		
				P	**Q**	**R**
Q: interrupt da real time clock	è passato 1 secondo della sleep di R **Q < R**	*R_int (CK)_1* *Wake_up (E2)* *R_int (CK)_2* *Preempt_1* *Change* *Sleep_on (E2)* *sleep* *G_SVC_23*	*Q* *Q* *Q* *Q – R* *R* *R* *R*	*attesa (E3)*	*pronto*	*esec*
R: receive		*G_SVC_1* *receive* *Sleep_on_1 (E4)* *Change* *Preempt_2* *R_int_3 (CK)*	*R* *R* *R* *R – Q* *Q* *Q*	attesa (E3)	esec	attesa (E4)
Q: open		*G_SVC_1* *open* *Sleep_on_1 (E5)* *Change*	*Q* *Q* *Q* *Q – idle*	*attesa (E3)*	*attesa (E5)*	*attesa (E4)*
idle: 2 interrupt da DMA_in	l'ultimo è relativo all'ultimo blocco da trasferire	*R_int (DMA_in)_1* *Wake_up (E5)* *R_int (DMA_in)_2* *Preempt_1* *Change* *Sleep_on (E5)_2* *open* *G_SVC_23*	*idle* *idle* *idle* *idle – Q* *Q* *Q* *Q*	*attesa (E3)*	*esec*	*attesa (E4)*
Q: exit		*G_SVC_1* *Exit_1* *Wake_up* *Exit_2* *Sleep_on (E3)* *wait* *G_SVC_23*	*Q* *Q* *Q* *Q – P* *P* *P* *P*	*esec*	*non esiste*	*attesa (E4)*

TABELLA DI COMMUTAZIONE DEI PROCESSI *(fine)*

In rosso sono colorate le parti di tabella assegnate. Come d'uso, quando la Preempt invocata tra G_SVC_2 e G_SVC_3 non effettua commutazione di contesto, non viene indicata e si scrive semplicemente G_SVC_23.

esercizio n. 3 – memoria virtuale

Un sistema dotato di memoria virtuale con paginazione e segmentazione di tipo UNIX, è caratterizzato dai parametri seguenti: la memoria centrale fisica ha capacità di **32 K byte**, quella logica di **32 K byte** e la pagina è di **4096 byte**. Si chiede di svolgere i punti seguenti:

a. Nel sistema vengono creati tre processi, indicati nel seguito con **P**, **Q** e **R**. I programmi eseguiti da tali processi sono due: **X** e **Y**. La dimensione iniziale dei segmenti dei programmi è la seguente:

CX: **8 K**	DX: **4 K**	PX: **4 K**	COND: **1 K**
CY: **8 K**	DY: **8 K**	PY: **4 K**	COND: **1 K**

La pagina condivisa è allocata lasciando libera 1 pagina dopo il segmento dati.

Nella tabella qui accanto **si inserisca** la struttura in pagine della memoria virtuale (mediante la notazione CX0 CX1 DX0 PX0 … CY0 …).

indir. virtuale	prog. X	prog. Y
0	*CX0*	*CY0*
1	*CX1*	*CY1*
2	*DX0*	*DY0*
3	*DX1*	*DY1*
4	*COND*	
5		*COND*
6		
7	*PX0*	*PY0*

b. In un certo istante di tempo t_0 sono terminati, nell'ordine, gli eventi seguenti:

1. creazione del processo **P** e lancio del programma **X** ("fork" di P ed "exec" di X) *alloca CP0 e PP0*
2. **P** accede alla variabile all'indirizzo virtuale assoluto **2100 hex** *alloca DP0*
3. **P** alloca la pagina condivisa *alloca COND*
4. **P** crea il processo figlio **Q** (P esegue "fork") *condivide CQ0 con CP0, DQ0 con DP0, COND e alloca PQ0*
5. P esegue una *exit* *dealloca CP0, DP0, PP0 dalla memoria*
6. **Q** esegue una *fork* e crea il processo figlio **R** *condivide CR0 con CQ0, DR0 con DQ0, COND e alloca PR0*

Valgono le convenzioni seguenti:

- un programma viene lanciato caricando **soltanto** la pagina di codice con l'istruzione di partenza e una pagina di pila, in quest'ordine
- il caricamento di pagine ulteriori avviene in **demand paging** (ossia su richiesta)
- l'indirizzo dell'istruzione di partenza di **X** è **0104 hex** (virtuale assoluto)
- l'indirizzo dell'istruzione di partenza di **Y** è **1010 hex** (virtuale assoluto)
- il numero R di pagine residenti vale **4**
- viene utilizzato l'algoritmo **LRU**
- l'allocazione delle pagine virtuali in pagine fisiche, avviene sempre in **sequenza**
- **ATTENZIONE: in una fork l'allocazione delle pagine del figlio avviene in sequenza di pagina virtuale, codice, eventuali dati e segmento dati condiviso, e pila.**

Si completi la **tab. 1A** (parte sinistra) con l'allocazione fisica delle pagine dei tre processi all'istante t_0, e **si completi** la **tab. 1B** con il contenuto della MMU, ipotizzando che le righe della tabella siano state allocate ordinatamente man mano che venivano allocate le pagine di memoria virtuale.

c. In un certo istante di tempo $t_1 > t_0$ sono terminati, nell'ordine, gli eventi seguenti:

7. **R** chiama la *exec* e passa ad eseguire il programma **Y** *dealloca CR0, DR0, COND e PR0 e alloca CR1, PR0*
8. **Q** chiama la funzione all'indirizzo virtuale assoluto **0200 hex** *accede alla pagina di pila PQ0 e di codice CQ0*
9. **R** esegue un accesso all'indirizzo virtuale assoluto **5016 hex** *alloca COND in MMU*
10. **Q** accede alla pagina dati in memoria *accede a CQ0 e DQ0*
11. **Q** esegue una *sbrk* con indirizzo massimo **3400 hex** *alloca DQ1 e dealloca COND solo dalla MMU*

Si completi la **tab. 2A** supponendo che, se occorre una pagina fisica libera, sia sempre usata la pagina fisica libera con indirizzo minore, e **si completi** la **tab. 2B** con il contenuto della MMU all'istante t_1.

memoria fisica	
indir. fisico	pagine allocate a t_0
0	*(CP0) = CQ0 = CR0*
1	*(PP0) PR0*
2	*(DP0) = DQ0 = DR0*
3	*COND*
4	*PQ0*
5	
6	
7	

Tabella 1A

MMU			
proc.	NPV	NPF	valid bit
(P) R	*(CP0 / 0) CR0 / 0*	*0 / 0*	*101*
(P) R	*(PP0 / 7) DR0 / 2*	*1 / 2*	*101*
(P) R	*(DP0 / 2) COND / 4*	*2 / 3*	*101*
(P) R	*(COND / 4) PR0 / 7*	*3 / 1*	*101*
Q	*CQ0 / 0*	*0*	*1*
Q	*DQ0 / 2*	*2*	*1*
Q	*COND / 4*	*3*	*1*
Q	*PQ0 / 7*	*4*	*1*

Tabella 1B

memoria fisica	
indir. fisico	pagine allocate a t_1
0	*CQ0 = (CR0)*
1	*(PR0) CR1*
2	*DQ0 = (DR0)*
3	*COND*
4	*PQ0*
5	*PR0*
6	*DQ1*
7	

Tabella 2A

MMU			
proc.	NPV	NPF	valid bit
R	*(CR0 / 0) CR1 / 1*	*0 / 0 / 1*	*10101*
R	*(DR0 / 2) PR0 / 7*	*1 / 2 / 5*	*10101*
(R) R	*(COND / 4) COND / 5*	*2 / 3 / 3*	*10101*
(R) Q	*(PR0 / 7) DQ1 / 3*	*3 / 1 / 6*	*10101*
Q	*CQ0 / 0*	*0*	*1*
Q	*DQ0 / 2*	*2*	*1*
(Q)	*(COND / 4)*	*3*	*10*
Q	*PQ0 / 7*	*4*	*1*

Tabella 2B

esercizio n. 4 – file system

prima parte

Su un calcolatore dotato di sistema operativo Linux viene eseguito il programma seguente, che parte come processo **P** e crea il processo **Q**.

NOTA: si ricorda che in ***lseek*** *(fd, offset, riferimento)*, riferimento = 0 indica inizio file, riferimento = 1 indica posizione corrente e riferimento = 2 indica fine file.

`int main ( ) { /* processo P */`	**sequenza**
`...`	
`fd1 = open ("/acso/esame/file1", O_RDWR);`	**1**
`read (fd1, buf1, 10);`	**2**
`write (fd1, buf2, 600);`	**3**
`pid1 = fork ()`	**4**
`if (pid1 == 0) { /* processo Q */`	
`fd2 = open ("/acso/esame/file2", O_RDONLY);`	**5**
`lseek (fd2, 1024, 1);`	**6**
`read (fd1, buf1, 100);`	**9**
`close (fd1)`	**10**
`exit (0);`	**11**
`} else { /* processo P */`	
`read (fd1, buf1, 10);`	**7**
`close (fd1);`	**8**
`} /* if */`	
`exit (0);`	**12**
`} /* main */`	

Durante l'esecuzione è stata osservata la **sequenza di chiamate di sistema indicata nella colonna di destra**.

Si completi il contenuto delle tabelle seguenti, indicando la sequenza dei valori assunti fino alla conclusione della chiamata di sistema numero **12** (si scriva **L** oppure **NE** per indicare che una cella si è liberata o non esiste più). Per la determinazione degli I-node, si veda il contenuto del volume a pagina seguente:

tab. file aperti del proc. **P**	
file des.	rif. riga
0	X
1	X
2	X
3	*2, L, NE*
4	
5	
6	

tab. file aperti del proc. **Q**	
file des.	rif. riga
0	X
1	X
2	X
3	*2, L, NE*
4	*3, NE*
5	
6	

tabella globale dei file aperti			
riga	posizione corrente	n di processi	i-nodo
0	X	X	X
1	X	X	X
2	*0, 10, 610, 620, 720, L*	*1, 2, 1, L*	*98*
3	*0, 1024, L*	*1, L*	*110*
4			
5			
6			

Al momento dell'esecuzione, il contenuto del volume è il seguente:

I-Lista: < 0, dir, 4 > < 6, dir, 20 > < 7, dir, 31 > < 98, norm, { 800, 801, 802, 803, 804, … } >
< 110, norm, { 1718, 1719, … } > < 232, norm, { 2733, 2734, … } >
blocco 4: … < 6, acso > …
blocco 20: … < 7, esame > …
blocco 31: … < 98, file1 > < 110, file2 > < 232, file3 > …

seconda parte

Si completi la tabella seguente, riportando, per ciascuna delle chiamate al File System indicate, la posizione corrente raggiunta dopo l'esecuzione del servizio, la sequenza di blocchi utilizzati e il tipo di accesso al disco eventualmente richiesto, con la notazione seguente:

- L (NB): legge il blocco numero NB dal disco
- S (NB): scrive il blocco numero NB sul disco

Si tenga conto delle ipotesi indicate in testa alla colonna relativamente al **numero di buffer disponibili** per contenere i blocchi del file.

Considerando le regole seguenti:

- le chiamate al File System sono eseguite nella sequenza indicata
- la dimensione di un blocco trasferito in DMA da o su file è di **512 byte**
- un blocco viene letto dal disco **solamente quando è necessario**
- per poter scrivere su un blocco è necessario averlo caricato in memoria, e un blocco viene scritto su disco **solamente quando è necessario** (per esempio quando il buffer che lo contiene va liberato)
- tutti gli I-node utilizzati **sono già presenti in memoria**

NOTA: i blocchi possono trovarsi in memoria (se già acceduti) o necessitare di trasferimento DMA da disco.

chiamata di sistema	**pos. corr.** nel file **dopo** la chiamata di sistema	**1 buffer**		**3 buffer**	
		blocchi di **file1 letti** da disco	blocchi di **file1 scritti** su disco	blocchi di **file1 letti** da disco	blocchi di **file1 scritti** su disco
fd1 = **open** ("/acso/esame/file1", O_RDWR)	0	L(4) *L(20)* *L(31)*	–	L(4) *L(20)* *L(31)*	–
lseek (fd1, 1020, 0)	*1020*	–	–	–	–
write (fd1, buf, 1)	*1021*	*L(801)*	–	*L(801)*	–
read (fd1, buf1, 9)	*1030*	*L(802)*	*S(801)*	*L(802)*	–
lseek (fd1, –700, 1)	*330*	–	–	–	–
write (fd1, buf2, 800)	*1130*	*L(800)* *L(801)* *L(802)*	*S(800)* *S(801)*	*L(800)*	–
close (fd1)	–	–	*S(802)*	–	*S(800)* *S(801)* *S(802)*

esercizio n. 5 – processi e stati fuori memoria

Un sistema ha memoria virtuale con paginazione e segmentazione di tipo UNIX, con numero **R** di pagine residenti pari a **4**. Sono in esecuzione due processi **P** e **Q**. In un certo istante di tempo t_1 la situazione della memoria fisica, il contenuto della MMU e gli stati dei processi sono rappresentati nelle tabelle relative mostrate sotto.

Inoltre si consideri quanto segue:

- Il sistema operativo gestisce anche gli stati dei processi di ***pronto fuori memoria*** e ***attesa fuori memoria***.
- Se è necessario liberare memoria, per esempio a seguito della creazione di un nuovo processo, il sistema operativo porta fuori memoria per primi i processi in attesa, e libera solo il numero di pagine strettamente necessarie. **Pertanto un processo fuori memoria può ancora avere alcune pagine effettivamente residenti nella memoria fisica.**
- Per tenere traccia delle pagine fuori memoria di ogni processo, la struttura della MMU è modificata introducendo il bit di controllo *swap_out*. Quando un processo subisce la transizione fuori memoria, il bit *swap_out* delle sole pagine portate fuori memoria viene posto a 1, e viene azzerato quando si verifica lo *swap_in*. Le pagine vengono portate fuori memoria sempre in quest'ordine: dati, pila e codice, considerando per prime – all'interno di ciascun segmento – **quelle caricate meno di recente**.

In un certo istante di tempo $t_2 > t_1$ si verifica l'evento seguente:

- il processo **P** esegue una fork e crea il processo **R**
  Per il processo R, che va in stato di pronto, vanno allocate 4 pagine: CR0 = CP0, CR1 = CP1, DR0 = DP0 e PR0. Per le prime 3 pagine il SO trova spazio in memoria (sono condivise), mentre per l'ultima pagina è necessario liberare della memoria. Come da specifica, il SO porta fuori memoria il processo Q, che è in attesa. In particolare il SO esegue lo swap out di una sola pagina di Q. La pagina da scaricare è la DQ0.

Si completino le tabelle (a pagina seguente) con la situazione al tempo t_2.

situazione al tempo t_1

memoria fisica	
indir. fisico	**pagine allocate**
0	CP0
1	DP0
2	PP0
3	CQ1
4	DQ0
5	PQ0
6	DQ1
7	CP1

MMU				
proc.	**NPV**	**NPF**	**swap_out bit**	**valid bit**
P	CP0 / 0	0	0	1
P	DP0 / 2	1	0	1
P	PP0 / F	2	0	1
Q	CQ1 / 1	3	0	1
Q	DQ0 / 3	4	0	1
Q	PQ0 / F	5	0'	1
Q	DQ1 / 4	6	0	1
P	CP1 / 1	7	0	1

proc.	stati del processo
P	esecuzione
Q	attesa

situazione al tempo t_2

memoria fisica	
indir. fisico	**pagine allocate**
0	CP0 = CR0
1	DP0 = DR0
2	PP0
3	CQ1
4	(DQ0) PR0
5	PQ0
6	DQ1
7	CP1 = CR1

MMU				
proc.	**NPV**	**NPF**	**swap_out bit**	**valid bit**
P	CP0 / 0	0	0	1
P	DP0 / 2	1	0	1
P	PP0 / F	2	0	1
Q	CQ1 / 1	3	0	1
Q	DQ0 / 3	4	1	1
Q	PQ0 / F	5	0	1
Q	DQ1 / 4	6	0	1
P	CP1 / 1	7	0	1
R	CR0 / 0	0	0	1
R	CR1 / 1	7	0	1
R	DR0 / 2	1	0	1
R	PR0	4	0	1

proc.	stato del processo
P	esecuzione
Q	attesa fuori memoria
R	pronto

esercizio n. 6 – thread e programmazione concorrente

Si consideri la bozza di sistema concorrente seguente (in linguaggio C), con due thread per i quali sono già previsti un mutex (`boh`), e due semafori (`mah_X` e `mah_Y`) inizializzati a zero.

I due thread vanno completati (secondo le modalità spiegate qui di seguito e tenendo conto della tabella mostrata a pagina seguente) dove ci sono righe lasciate appositamente vuote. In una riga vuota si può aggiungere **una sola** chiamata di sistema: o relativa al mutex (`lock` oppure `unlock`), o relativa a uno dei due semafori (`post` oppure `wait`); la riga (o eventualmente un intero gruppo di righe consecutive) può anche restare inutilizzata (o inutilizzato). Il numero di righe vuote **non è significativo**.

```
pthread_mutex_t boh;
sem_t mah_X, mah_Y;
int globale = 0;

void * uno (void * arg) {

   sem_wait (&mah_X) _____________________

   pthread_mutex_lock (&boh) _____________

   globale = 1;                                          /* statement A */

   sem_post (&mah_Y) _____________________

    ______________________________________

   pthread_mutex_unlock (&boh)
   return NULL;
} /* end uno */

void * due (void * arg) {

   pthread_mutex_lock (&boh)                             /* statement B */

   sem_post (&mah_X) _____________________

    ______________________________________

   globale = 2;                                          /* statement C */

   pthread_mutex_unlock (&boh) ___________

   sem_wait (&mah_Y) _____________________
   return NULL;
} /* end due */

void main ( ) {
   pthread_t th_1, th_2;
   sem_init (&mah_X, 0, 0);
   sem_init (&mah_Y, 0, 0);
   pthread_create (&th_1, NULL, uno, NULL);
   pthread_create (&th_2, NULL, due, NULL);
   pthread_join (th_1, NULL);
   pthread_join (th_2, NULL);
return;
} /* end main */
```

Si consideri la tabella seguente, la quale specifica parzialmente lo **stato** che il sistema concorrente dovrebbe assumere in tre **condizioni** differenti (A, B e C, indicate nel codice del sistema).

Si completi il codice dato a pagina precedente, secondo le modalità spiegate all'inizio, così da ottenere un sistema concorrente che **si comporti** precisamente **come specificato dalla tabella**, e **non abbia stalli**.

Si badi bene alla colonna "condizione": con "subito dopo statement X" si chiede lo stato che il sistema assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**stato del sistema concorrente**	
	th_2	*globale*
subito dopo stat. **A**	ESISTE	
subito dopo stat. **B**		0
subito dopo stat. **C**		2

Un modo ordinato per risolvere questo esercizio consiste nell'esaminare le condizioni A, B e C, individuare i problemi di concorrenza che esse sottendono, e per ciascun problema individuato scegliere la soluzione "standard" o comunque ovvia che la programmazione concorrente mette a disposizione, compatibilmente con quanto è già scritto nel codice (dove figurano alcune chiamate di sistema). Conviene dare uno sguardo d'insieme a tutte e tre le condizioni prima di attaccarne una specifica. Si ricordi che l'ordine in cui esse figurano nella tabella dall'alto verso il basso, non implica affatto che esse siano ordinate temporalmente nello stesso modo; anzi il loro ordine temporale potrebbe non essere deterministico.

Esaminando rapidamente le tre condizioni A, B e C, si nota subito che la C implica che il thread 2 operi senza interferenza da parte del thread 1, ossia (ragionevolmente) in mutua esclusione; poiché nel codice sono già presenti chiamate di sistema per il mutex, ma lo schema mutex è incompleto, pare sensato cercare di soddisfare per prima la C (qualunque sia il suo ordine temporale rispetto ad A e B, se pure un ordine temporale deterministico esiste) completando in qualche modo lo schema abbozzato; e poi soddisfare le altre due condizioni, cominciando dalla B, la quale palesemente si riferisce a un valore iniziale, e pertanto concludendo con la A, la quale oltretutto è formulata in modo un po' diverso ossia come stato di esistenza. Ora si può passare a un esame approfondito delle tre condizioni, nell'ordine C, B e A.

Per la condizione C, si vuole tenere fissa a valore 2 la variabile globale subito dopo che il thread 2 la ha assegnata (ossia nell'intervallo di tempo tra l'assegnamento a 2 e lo statement successivo). Pertanto, dato che entrambi i thread modificano globale, gli assegnamenti a globale devono essere mutuamente esclusivi. In altri termini, ciascuno dei due thread deve avere una sequenza critica che contenga il rispettivo assegnamento a globale (o almeno questa è la soluzione più semplice e immediata, rispetto a usare i semafori, e comunque è già parzialmente data nel codice). Ciò implicherà di posizionare la lock del thread 1 a monte dell'assegnamento a globale, e la unlock del thread 2 a valle dell'assegnamento a globale. In linea di massima ciò definisce il posizionamento approssimativo della lock e della unlock evidentemente mancanti. Come esattamente posizionare tale lock e tale unlock relativamente alle post e wait che andranno (presumibilmente) collocate per soddisfare le specifiche rimanenti, è determinato dalle altre due condizioni.

Per la condizione B (la più facile), si vuole tenere fissa a valore iniziale 0 la variabile globale subito dopo che il thread 2 ha superata la lock (e ovviamente prima di avere assegnato valore 2 a globale), ossia prima che il thread 1 possa assegnare valore 1 a globale. Insomma il thread 2 deve eseguire la sequenza critica di assegnamento a globale prima del thread 1. È un problema di ordinamento e si risolve tramite un semaforo. Pertanto il thread 1 deve effettuare una wait su un semaforo, p. es. `mah_X`*, a monte della lock da collocare. Di conseguenza alla wait su* `mah_X`*, il thread 2 deve effettuare una post su* `mah_X`*, a valle della lock già data nel codice (come posizionare tale post rispetto all'assegnamento a globale è un dettaglio minore, senza conseguenze, e viene indirettamente deciso dalla condizione A – vedi sotto).*

Per la condizione A, si vuole garantire che il thread 2 esista (ancora) mentre il thread 1 è in sequenza critica. È di nuovo un problema di ordinamento e si risolve tramite un semaforo. Tuttavia non si può riusare il semaforo `mah_X`*: per due thread che si bloccano-sbloccano ordinatamente a vicenda, occorrono due semafori indipendenti. Pertanto il thread 2 deve effettuare una wait sull'altro semaforo,* `mah_Y`*, a valle della unlock da collocare (il sistema non deve avere stalli – se la wait fosse a monte della unlock ci sarebbe stallo). Di conseguenza alla wait su* `mah_Y`*, il thread 1 deve effettuare una post su* `mah_Y`*.*

Dettagli: la post su `mah_Y` *nel thread 1 potrebbe stare a monte o a valle dell'assegnamento a globale, ma per i vincoli di spazio (numero di righe vuote) deve stare a valle, e potrebbe stare a monte o a valle della unlock già data nel codice, ma per i vincoli di spazio deve stare a monte; e indirettamente, sempre per i vincoli di spazio, la post su* `mah_X` *nel thread 2 va collocata prima dell'assegnamento a globale (ciò sistema il dettaglio lasciato irrisolto esaminando la condizione B).*

Con le considerazioni svolte sopra, la struttura del sistema concorrente risulta (ragionevolmente) determinata univocamente. Ovviamente i due semafori sono interscambiabili, ma questo è un aspetto banale. Si osservi che le tre condizioni A, B e C determinano globalmente la struttura del sistema concorrente, e che nessuna di esse, presa isolatamente, determina univocamente dove esattamente collocare una certa chiamata di sistema all'interno del codice dato e relativamente alle altre chiamate che vanno pure collocate. Beninteso potrebbero esistere linee di ragionamento differenti da quella qui proposta.

spazio libero per minuta

spazio libero per minuta

spazio libero per minuta