Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

prof. Luca Breveglieri **prof.ssa Donatella Sciuto**
prof. Gerardo Pelosi **prof.ssa Cristina Silvano**

AXO – Architettura dei Calcolatori e Sistemi Operativi

SECONDA PARTE – martedì 2 febbraio 2021

Cognome_____ **Nome** _____

Matricola _____**Firma** _____

Istruzioni

- Si scriva solo negli spazi previsti nel testo della prova e non si separino i fogli.
- Per la minuta si utilizzino le pagine bianche inserite in fondo al fascicolo distribuito con il testo della prova. I fogli di minuta se staccati vanno consegnati intestandoli 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 1 h : 30 m

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

esercizio 1 (4 punti) _____

esercizio 2 (5 punti) _____

esercizio 3 (6 punti) _____

esercizio 4 (1 punti) _____

voto finale: (16 punti) _____

CON SOLUZIONI (in corsivo)

esercizio n. 1 – programmazione concorrente

Si consideri il programma C seguente (gli "`#include`" e le inizializzazioni dei *mutex* sono omessi, come anche il prefisso `pthread` delle funzioni di libreria NPTL):

```
pthread_mutex_t power
sem_t strong, weak
int global = 0
```

```
void * master (void * arg) {
   mutex_lock (&power)
   sem_post (&strong)
   global = 1                                          /* statement A */
   mutex_unlock (&power)
   mutex_lock (&power)
   sem_wait(&weak)
   mutex_unlock (&power)
   global = 2
   sem_wait (&strong)
   return NULL
} /* end master */
```

```
void * slave (void * arg) {
   mutex_lock (&power)
   sem_wait (&strong)
   global = 3                                          /* statement B */
   sem_post(&weak)
   mutex_unlock (&power)                               /* statement C */
   sem_wait (&weak)
   return (void *) 4
} /* end slave */
```

```
void main ( ) {
   pthread_t th_1, th_2
   sem_init (&strong, 0, 1)
   sem_init (&weak, 0, 0)
   create (&th_1, NULL, master, NULL)
   create (&th_2, NULL, slave, NULL)
   join (th_2, &global)                                /* statement D */
   sem_post(&weak)
   join (th_1, 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 – *master*	**th_2 – *slave***
subito dopo stat. **A**	*ESISTE*	*PUÒ ESISTERE*
subito dopo stat. **B**	*ESISTE*	*ESISTE*
subito dopo stat. **C**	*PUÒ ESISTERE*	*ESISTE*
subito dopo stat. **D**	*ESISTE*	*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 supponga che il mutex valga 1 se occupato, e valga 0 se libero

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**			
	power	*strong*	*weak*	*global*
subito dopo stat. **A**	*1*	*1 / 2*	*0 / 1*	*1 / 4*
subito dopo stat. **B**	*1*	*1*	*0*	*3*
subito dopo stat. **D**	*0 / 1*	*0 / 1*	*0*	*1 / 4*

Il sistema può andare in stallo (*deadlock*), con uno o più *thread* che si bloccano, in (almeno) **due casi diversi**. Si chiede di precisare il comportamento dei thread in **due casi**, indicando gli statement dove avvengono i blocchi e i possibili valori della variabile *global*:

caso	**th_1 – *master***	**th_2 – *slave***	*global*
1	***wait weak***	***lock power***	***1***
2	***terminato***	***wait weak***	***2 / 3***
3			

Nota: i deadlock possibili sono tutti e soli i due indicati in tabella.

esercizio n. 2 – processi e nucleo

prima parte – gestione dei processi

```
// programma prova.c
main ( ) {
   pid = fork ( )              // P crea Q
   if (pid == 0) {             // codice eseguito da Q
      write (stdout, o_msg, 15)
      execl ("/acso/prog_x", "prog_x", NULL)
      exit (-1)
   } else {                    // codice eseguito da P
      read (stdin, i_msg, 5)
      pid = wait (&status)
   } // end_if pid
   exit (0)
} // prova
```

```
// programma prog_x.c
// dichiarazione e inizializzazione dei mutex presenti nel codice
```

void * walk (void * arg) {		void * run (void * arg) {
mutex_lock (&go)		mutex_lock (&go)
sem_post (&stay)		sem_wait (&stay)
mutex_unlock (&go)		mutex_lock (&come)
mutex_lock (&come)		sem_post (&stay)
sem_wait (&stay)		mutex_unlock (&come)
mutex_unlock (&come)		sem_post (&stay)
sem_wait (&stay)		mutex_unlock (&go)
return NULL		return NULL
} //end		} // end

```
main ( ) {  // codice eseguito da Q
   pthread_t th_1, th_2
   sem_init (&stay, 0, 0)
   create (&th_2, NULL, run, NULL)
   create (&th_1, NULL, walk, NULL)
   nanosleep (4)
   join (th_2, NULL)
   join (th_1, NULL)
   exit (1)
} // main
```

Un processo ***P*** esegue il programma ***prova*** e crea un processo figlio ***Q*** che esegue una mutazione di codice (programma ***prog_x***). La mutazione di codice va a buon fine e ***Q*** crea i thread ***th_1*** e ***th_2.***

Si simuli l'esecuzione dei processi completando tutte le righe presenti nella tabella così come risulta dal codice dato, dallo stato iniziale, dagli eventi indicati, e nell'ipotesi che il processo ***Q*** abbia **già eseguito** **`execl`** ma **non** ancora **`create (&th_2)`**. Si completi la tabella riportando quanto segue:

- ⟨ *PID*, *TGID* ⟩ di ciascun processo che viene creato
- ⟨ *identificativo del processo-chiamata di sistema / libreria* ⟩ nella prima colonna, dove necessario e in funzione del codice proposto
- in ciascuna riga lo stato dei processi **al termine dell'evento o della chiamata associata alla riga stessa**; si noti che la prima riga della tabella **potrebbe essere solo parzialmente completata**

TABELLA DA COMPILARE (numero di colonne non significativo)

identificativo simbolico del processo		**IDLE**	**P**	**Q**	*TH_2*	*TH_1*		
evento oppure *processo-chiamata*	*PID*	**1**	**2**	*3*	*4*	*5*		
	TGID	**1**	**2**	*3*	*3*	*3*		
P – read (..)	**0**	**pronto**	**attesa (read)**	**esec**	**NE**	**NE**		
Q – create (th_2)	1	*pronto*	*A*	*Esec*	*pronto*	*NE*		
Q – create (th_1)	2	*pronto*	*A*	*Esec*	*pronto*	*pronto*		
interrupt da real-time clock e scadenza del quanto di tempo	3	pronto	A	pronto	esec	pronto		
5 interrupt da STD_IN tutti i byte trasferiti	4	pronto	esec	pronto	pronto	pronto		
interrupt da *real-time clock* e scadenza del quanto di tempo	5	*pronto*	*pronto*	*pronto*	*pronto*	*esec*		
TH_1 – lock (&go)	6	*pronto*	*pronto*	*pronto*	*pronto*	*esec*		
TH_1 – post (&stay)	7	*pronto*	*pronto*	*pronto*	*pronto*	*esec*		
interrupt da *real-time clock* e scadenza del quanto di tempo	8	*pronto*	*pronto*	*esec*	*pronto*	*pronto*		
Q – nanosleep ( )	9	*pronto*	*pronto*	*attesa (nano)*	*esec*	*pronto*		
TH_2 – lock (&go)	10	*pronto*	*esec*	*A*	*attesa (lock go)*	*pronto*		
P – wait ( )	11	*pronto*	*attesa (wait)*	*A*	*A*	*E*		
TH_1 – unlock (&go)	12	pronto	A	A	esec	pronto		

seconda parte – moduli, pila e strutture dati HW

Sono dati due processi normali **P** e **Q**. Non ci sono altri processi utente nel sistema. Lo stato iniziale delle pile di sistema e utente dei due processi è parzialmente riportato qui sotto.

Y	
uBase_P	

uStack_P – iniziale

sBase_P	

sStack_P – iniziale

X	rientro a *mutex_lock (&pass)* da *syscall*
	rientro a cod. ut. da *mutex_lock (&pass)*
W	
uBase_Q	

uStack_Q – iniziale

	X
	rientro a *...* da *...*
	rientro a *...* da *...*
	rientro a *System_Call* da *...*
	PSR U
sBase_Q	rientro a *syscall* da *System_Call*

sStack_Q – iniziale

Si consideri l'**evento** seguente: **P** esegue *mutex_unlock* **(&pass)** e si ha *preemption*.

Si mostrino le invocazioni di tutti i **moduli** (e eventuali relativi ritorni) fino al **momento** in cui il processo **Q** è tornato in esecuzione nel **modulo di SO in cui era stato sospeso** (tabella a pagina seguente), e **si mostri** lo stato delle pile del processo **P** in quel preciso **momento** (qui sotto).

	rientro a *mutex_unlock (pass)* da *syscall*
	rientro a cod. ut. da *mutex_unlock (pass)*
Y	
uBase_P	

uStack_P – finale

	Y -2 (USP salvato)
	~~rientro a *schedule* da *pick_next_task*~~
	rientro a *System_Call* da *schedule*
	~~rientro a *check_preempt_curr* da *resched*~~
	~~rientro a *wake_up* da *check_preempt_curr*~~
	~~rientro a *sys_futex (wake)* da *wake_up*~~
	~~rientro a *System_Call* da *sys_futex (wake)*~~
	PSR U
sBase_P	rientro a *syscall* da *System_Call*

sStack_P – finale

Si risponda alle seguenti domande:

1) Indicare il **modulo** di SO in cui il processo **Q** si trova nel **momento** preciso del suo ritorno in esecuzione: *schedule*

2) Indicare il **modulo** di SO in cui il processo **Q** era stato sospeso: *wait_event*

3) Indicare il **valore di USP** nel momento in cui **Q** è tornato in esecuzione: *X*

invocazione moduli (numero di righe vuote non significativo)

tabella di invocazione dei moduli		
processo	**modo**	**modulo**
P	U	> mutex_unlock (&pass)
P	*U*	*> syscall*
P	*U-S*	*> System_Call*
P	*S*	*> sys_futex (wake)*
P	*S*	*> wake_up*
P	*S*	*> check_preempt_curr*
P	*S*	*> resched (set TNR) <*
P	*S*	*check_preempt_curr <*
P	*S*	*wake_up <*
P	*S*	*sys_futex (wake) <*
P	*S*	*> schedule*
P	*S*	*> pick_next_task <*
P-Q	*S*	*context_switch*
Q	*S*	*schedule <*
Q	*S*	*wait_event <*

esercizio n. 3 – memoria e file system

prima parte – gestione dello spazio di memoria

È dato un sistema di memoria caratterizzato dai seguenti parametri generali:

MAXFREE = **3** **MINFREE** = **2**

situazione iniziale (esistono un processo P e un processo R)

```
PROCESSO: P  ********************************************************
    VMA : C  000000400,  2 , R  ,  P  ,  M  , <XX,0>
          K  000000600,  1 , R  ,  P  ,  M  , <XX,2>
          S  000000601,  1 , W  ,  P  ,  M  , <XX,3>
          P  7FFFFFFF9,  6 , W  ,  P  ,  A  , <-1,0>
    PT: <c0 :- -> <c1 :1  R> <k0 :- -> <s0 :- -> <p0 :s0 R> <p1 :2  R>
        <p2 :7  R> <p3 :4  R> <p4 :6  W> <p5 :- ->
    process P - NPV of PC and SP:  c1, p4
PROCESSO: R  ***** non di interesse per l'esercizio ******************

____MEMORIA FISICA____(pagine libere: 3)_____________________________
      00 : <ZP>                ||  01 : Pc1/Rc1/<XX,1>      ||
      02 : Pp1/Rp1             ||  03 : ----                ||
      04 : Pp3/Rp3             ||  05 : Rp4 D               ||
      06 : Pp4                 ||  07 : Pp2/Rp2             ||
      08 : ----                ||  09 : ----                ||

____STATO del TLB_____________________________________________________
      Pc1 : 01 -  0: 1:        ||         -----             ||
      Pp1 : 02 -  1: 0:        ||   Pp2 : 07 -  1: 0:       ||
      Pp3 : 04 -  1: 0:        ||   Pp4 : 06 -  1: 0:       ||
            -----              ||         -----             ||
            -----              ||         -----             ||

SWAP FILE:     Pp0 / Rp0, ----, ----, ----, ----, ----,
LRU ACTIVE:    PC1,
LRU INACTIVE: pp4,  pp3,  rp4,  rp3,  rc1,  rp2,  rp1,  pp2,  pp1,
```

evento 1: *read* (Pp0), *write* (Pp0)

per la lettura fa swap-in di pp0/rp0 in 03, abilitato COW, LRU list aggiornate, pagina lasciata anche nello swap file. Per la scrittura a causa di COW scatta PFRA che libera 02 (pp1/rp1 in swap file per D in TLB), e 07 (pp2/rp2 in swap file per D in TLB. LRU lis aggiornate. la pp0 sdoppiata viene allocata in 02 e scritta e il riferimento nello swap file viene eliminato.

PT del processo: **P**				
p0: *2 W*	p1: *s1 R*	p2: *s2 R*	p3: *4 R*	p4: *6 W*
P5: – –				

process P	NPV of **PC**: *c1*	NPV of **SP**: *p0*

MEMORIA FISICA	
00: <ZP>	01: Pc1 / Rc1 /<XX, 1>
02: *Pp0*	03: *Rp0*
04: *Pp3 / Rp3*	05: *Rp4 D*
06: *Pp4*	07: *----*
08: *----*	09: *----*

SWAP FILE	
s0: *Rp0*	s1: *Pp1 / Rp1*
s2: *Pp2 / Rp2*	s3:

s4:	s5:

LRU ACTIVE: *PP0, PC1,* _____

LRU INACTIVE: *pp4, pp3, rp4, rp3, rc1, rp0,* _____

evento 2: ***read*** **(Pp1)**

swap-in di pp1/rp1 in 07, LRU list aggiornate, pagina lasciata anche nello swap file.

process P	NPV of **PC**: *c1*	NPV of **SP**: *p1*

MEMORIA FISICA	
00: <ZP>	01: Pc1 / Rc1 /<XX, 1>
02: *Pp0*	03: *Rp0*
04: *Pp3 / Rp3*	05: *Rp4 D*
06: *Pp4*	07: *Pp1 / Rp1*
08: *----*	09: *----*

SWAP FILE	
s0: *Rp0*	s1: *Pp1 / Rp1*
s2: *Pp2 / Rp2*	s3:
s4:	s5:

LRU ACTIVE: *PP1, PP0, PC1,* _____

LRU INACTIVE: *pp4, pp3, rp4, rp3, rc1, rp0, rp1* _____

evento 3: ***mmap*** **(0x000030000000, 3, W, S, M, "F", 2)** **VMA M0**

mmap **(0x000040000000, 2, W, P, A, -1, 0)** **VMA M1**

(NON è richiesto di compilare nulla per questo evento)

evento 4: ***read*** **(Pm10, Pm11, Pm01),** ***write*** **(Pm10)**

read pm10 e pm11 in ZP abilitate COW. read pm01 fa scattare PFRA che libera 03 (rp0 swapout giò presente in swapfile) e 05 (rp4 in swap file perché D). LRU list aggiornate di conseguenza. Pm01 viene caricata in 03. Write Pm10 fa allocare 05.

PT del processo: **P**				
m00: - -	m01: *3 W*	m02: - -	m10: *5 W*	m11: *0 R*

MEMORIA FISICA	
00: <ZP> / *Pm11*	01: Pc1 / Rc1 /<XX, 1>
02: *Pp0*	03: *Pm01 / <F,3>*
04: *Pp3 / Rp3*	05: *Pm10*
06: *Pp4*	07: *Pp1 / Rp1*
08: *----*	09: *----*

SWAP FILE	
s0: *Rp0*	s1: *Pp1 / Rp1*

s2: *Pp2 / Rp2*	s3: *Rp4*
s4:	s5:

LRU ACTIVE: *PM01, PM11, PM10, PP1, PP0, PC1,* _____

LRU INACTIVE: *pp4, pp3, rp3, rc1, rp1* _____

seconda parte – file system

È dato un sistema di memoria caratterizzato dai seguenti parametri generali:

MAXFREE = 2 **MINFREE = 1**

Si consideri la seguente **situazione iniziale**:

```
____MEMORIA FISICA____(pagine libere: 1)_____________________________
      00 : <ZP>                  ||  01 : Pc2/Qc2/<X,2>        ||
      02 : Qp0    D              ||  03 : <F,0>    D           ||
      04 : <F,1>    D            ||  05 : <F,2>    D           ||
      06 : Pp0    D              ||  07 : ----                 ||

____STATO del TLB_____________________________________________________
      Qc2 : 01 -  0: 1:          ||   Qp0 : 02 -  1: 1:        ||
            -----                ||         -----              ||
            -----                ||         -----              ||
```

nome file	f_pos	f_count	numero pag. lette	numero pag. scritte
F	**9000**	**2**	**3**	**0**

Per ciascuno dei seguenti eventi compilare le tabelle richieste con i dati relativi al contenuto della memoria fisica, delle variabili del FS relative ai file indicati e al numero di accessi a disco effettuati in lettura e in scrittura.

ATTENZIONE: il numero di pagine lette o scritte di un file è cumulativo, ossia è la somma delle pagine lette o scritte su quel file da tutti gli eventi precedenti oltre a quello considerato. Si ricorda che la primitiva *close* scrive le pagine dirty di un file solo se *f_count* diventa = 0.

Il file **F** è stato aperto da **P** tramite chiamata **fd = *open* (F)**. Quindi **P** ha creato il figlio **Q**.

Il processo **Q** è ora in esecuzione, come si può anche desumere dallo stato del TLB.

eventi 1, 2 e 3: fd1 = *open* (G), *write* (fd1, 4000), *read* (fd, 1000)

la scrittura di G0 attiva PFRA che libera 03 e 04 scrivendo F0 e F1 sul file F. G0 viene allocato in 03. la lettura di 1000 byte di F utilizza la pag F2 già caricata in memoria.

MEMORIA FISICA	
00: <ZP>	01: Pc2 / Qc2 / <X, 2>
02: *Qp0 D*	03: *<G, 0> D*
04: *----*	05: *<F, 2> D*
06: *Pp0 D*	07: *----*

nome file	f_pos	f_count	numero pag. lette	numero pag. scritte
G	*4000*	*1*	*1*	*0*
F	*10000*	*2*	*3*	*2*

eventi 4 e 5: *fork* (R), *context switch* (R)

Fork crea il processo figlio R e deve allocare la pagina di (cima) pila Rp0, separandola da Qp0 e marcandola D (dirty). La pagina 02 rimane allocata a Rp0, mentre la 04 viene allocata alla pagina di pila del padre Q, cioè Qp0.

MEMORIA FISICA	
00: <ZP>	01: *Pc2 /Qc2 /Rc2 <X, 2>*
02: *Rp0 D*	03: *<G, 0> D*
04: *Qp0 D*	05: *<F, 2> D*
06: *Pp0 D*	07: *----*

nome file	f_pos	f_count	numero pag. lette	numero pag. scritte
G	*4000*	*2*	*1*	*0*
F	*10000*	*3*	*3*	*2*

evento 6: *exit* (Q) (il processo **R** esegue exit e va in esecuzione **Q**)

MEMORIA FISICA	
00: <ZP>	01: *Pc2 /Qc2 <X, 2>*
02: *----*	03: *<G, 0> D*
04: *Qp0 D*	05: *<F, 2> D*
06: *Pp0 D*	07: *----*

nome file	f_pos	f_count	numero pag. lette	numero pag. scritte
G	*4000*	*1*	*1*	*0*
F	*10000*	*2*	*3*	*2*

eventi 7 e 8: *close* (fd1), *close* (fd)

nome file	f_pos	f_count	numero pag. lette	numero pag. scritte
G	*----*	*0*	*1*	*1*
F	*10000*	*1*	*3*	*2*

esercizio n. 4 – tabella delle pagine

Date le VMA di un processo sotto riportate, definire:

1. la decomposizione degli indirizzi virtuali dell'NPV iniziale di ogni area secondo la notazione **PGD : PUD : PMD : PT**
2. il numero di pagine necessarie in ogni livello della gerarchia e il numero totale di pagine necessarie a rappresentare la Tabella delle Pagine (TP) del processo
3. il numero di pagine virtuali occupate dal processo
4. il rapporto tra l'occupazione della TP e la dimensione virtuale del processo in pagine
5. la dimensione virtuale massima del processo in pagine, senza dover modificare la dimensione della TP
6. il rapporto relativo

VMA del processo **P**							
AREA	NPV iniziale	dimensione	R/W	P/S	M/A	nome file	offset
C	0000 0040 0	2	R	P	M	X	0
K	0000 0060 0	1	R	P	M	X	3
D	0000 0060 1	16	W	P	A	-1	0
M0	0000 AB00 0	4	W	S	M	F	5
T0	7FFF F77F D	3	W	P	A	-1	0
P	7FFF FFFF C	3	W	P	A	-1	0

Decomposizione degli indirizzi virtuali

		PGD :	**PUD :**	**PMD :**	**PT**
C	0000 0040 0	*0*	*0*	*2*	*0*
K	0000 0060 0	*0*	*0*	*3*	*0*
D	0000 0060 1	*0*	*0*	*3*	*1*
M0	0000 AB00 0	*0*	*2*	*344*	*0*
T0	7FFF F77F D	*255*	*511*	*443*	*509*
P	7FFF FFFF C	*255*	*511*	*511*	*508*

Numero di pagine necessarie

# pag PGD	*1*
# pag PUD	*2*
# pag PMD	*3*
# pag PT	*5*
# pag totali	*11*

Numero di pagine virtuali occupate dal processo	*29*

Rapporto di occupazione	*11/29 = 0,379*

Dimensione massima del processo in pagine virtuali	*Con la stessa dimensione di TP il processo può crescere fino a 5 x 512 = 2560 pagine virtuali*

Rapporto di occupazione con dimensione massima	*11 / 2560 = 0,0043*

spazio libero per brutta copia o continuazione