Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

prof.ssa Anna Antola **prof.ssa Donatella Sciuto**
prof. Luca Breveglieri **prof.ssa Cristina Silvano**

AXO – Architettura dei Calcolatori e Sistemi Operativi

SECONDA PARTE – lunedì 20 luglio 2020

Cognome_____ **Nome** _____

Matricola_____ **Codice Persona**_____

Istruzioni – ESAME ONLINE

È vietato consultare libri, eserciziari e appunti, nonché cellulari e altri dispositivi mobili di calcolo o comunicazione. Chiunque non dovesse attenersi alla regola vedrà annullata la propria prova.

La prova va sempre consegnata completando la procedura prevista nel modulo (form) dell'esame con INVIO (SUBMIT) del testo risolto. Se lo studente intende RITIRARSI deve inviare messaggio di posta elettronica (email) al docente dopo avere completata la procedura.

Dallo **HONOR CODE**

In qualsiasi progetto o compito, gli studenti devono dichiarare onestamente il proprio contributo e devono indicare chiaramente le parti svolte da altri studenti o prese da fonti esterne.

Ogni studente garantisce che eseguirà di persona tutte le attività associate all'esame senza alcun aiuto di altri; la sostituzione di identità è un reato perseguibile per legge.

Durante un esame, gli studenti non possono accedere a fonti (libri, note, risorse online, ecc) diverse da quelle esplicitamente consentite.

Durante un esame, gli studenti non possono comunicare con nessun altro, né chiedere suggerimenti.

In caso di esame a distanza, gli studenti non cercano di violare le regole a causa del controllo limitato che il docente può esercitare.

L'accettazione dello Honor Code costituisce prerequisito per l'iscrizione agli esami.

Tempo a disposizione 1 h : 30 m

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

esercizio 1 (4 punti) _____

esercizio 2 (6 punti) _____

esercizio 3 (6 punti) _____

voto finale: (16 punti) _____

BLANK PAGE

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 primitive di libreria NPTL):

```
pthread_mutex_t positive, negative
sem_t one
int global = 0
```

```
void * more (void * arg) {
   mutex_lock (&positive)
      sem_post (&one)
      global = 2                                   /* statement A */
   mutex_lock (&negative)
      sem_wait (&one)
   mutex_unlock (&positive)                        /* statement B */
      sem_wait (&one)
   mutex_unlock (&negative)
   return (void *) 3
} /* end more */
```

```
void * less (void * arg) {
   mutex_lock (&positive)
      sem_wait (&one)
   mutex_unlock (&positive)                        /* statement C */
   global = 1
   mutex_lock (&negative)
      sem_post (&one)                              /* statement D */
   mutex_unlock (&negative)
   return NULL
} /* end less */
```

```
void main ( ) {
   pthread_t th_1, th_2
   sem_init (&one, 0, 1)
   create (&th_1, NULL, more, NULL)
   create (&th_2, NULL, less, NULL)
   join (th_1, &global)                            /* statement E */
   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 – more	**th_2** – less
subito dopo stat. **A**		
subito dopo stat. **B**		
subito dopo stat. **C**		
subito dopo stat. **E**		

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**		
	positive	*negative*	*one*
subito dopo stat. **A**			
subito dopo stat. **C**			
subito dopo stat. **D**			
subito dopo stat. **E**			

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

caso	**th_1** – more	**th_2** – less	*global*
1			
2			
3			

esercizio n. 2 – processi e nucleo

prima parte – gestione dei processi

```
// programma prova.c
main ( ) {
   pid1 = fork ( )                 // P crea Q
   if (pid1 == 0) {                // codice eseguito solo da Q
      execl ("/acso/prog_x", "prog_x", NULL)
      exit (-1)
   } else {
      pid2 = fork ( )              // P crea R
      nanosleep (4)
      if (pid2 == 0) {             // codice eseguito solo da R
         read (stdin, msg, 50)
         exit (-1)
      } else {                     // codice eseguito solo da P
         pid = wait (&status)      // P aspetta la terminazione di uno dei due figli
      } // end_if pid2
   } // end_if pid1
   exit (0)
} // prova
```

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

`void * me (void * arg) {`		`void * you (void * arg) {`
`sem_wait (&far)`		`mutex_lock (&here)`
`sem_wait (&near)`		`sem_post (&far)`
`mutex_lock (&here)`		`sem_wait (&near)`
`sem_post (&near)`		`mutex_unlock (&here)`
`mutex_unlock (&here)`		`sem_wait (&near)`
`return NULL`		`return NULL`
`} // me`		`} // you`

```
main ( ) {  // codice eseguito da Q
    pthread_t th_1, th_2
    sem_init (&near, 0, 2)
    sem_init (&far, 0, 0)
    create (&th_1, NULL, me, NULL)
    create (&th_2, NULL, you, NULL)
    join (th_1, NULL)
    join (th_2, NULL)
    exit (1)
} // main
```

Un processo ***P*** esegue il programma ***prova*** e crea due processi figli ***Q*** e ***R***; il figlio ***Q*** esegue una mutazione di codice (programma ***prog_x***). La mutazione di codice va a buon fine e vengono creati 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 e dagli eventi indicati, e tenendo conto che il processo ***Q*** **ha già eseguito** la primitiva **`create (&th_1, … )`** ma **non** la primitiva **`create (&th_2, … )`**. Si completi la tabella riportando quanto segue:

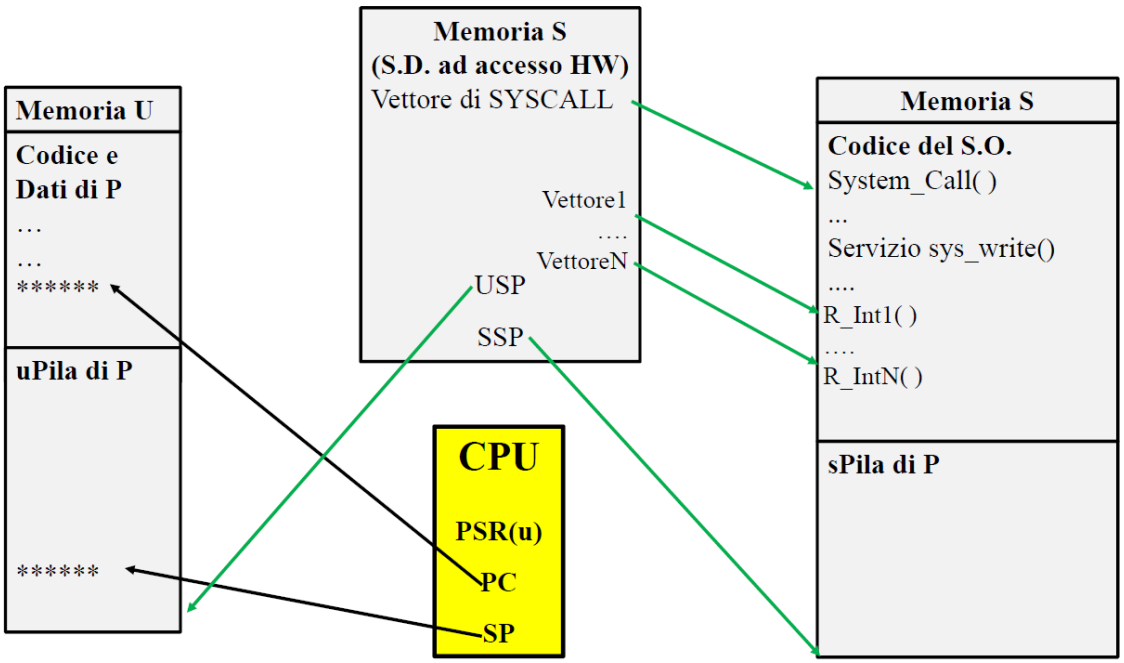
- ⟨ *PID*, *TGID* ⟩ di ciascun processo che viene creato
- ⟨ *identificativo del processo-chiamata di sistema / libreria* ⟩ nella prima colonna, dove necessario e in funzione del codice proposto
- in ciascuna riga lo stato dei processi **al termine dell'evento o della chiamata associata alla riga stessa**; 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_1**	**R**		
evento oppure *processo-chiamata*	*PID*	**1**	**2**					
	TGID	**1**	**2**					
P – nanosleep (4)	**0**	**pronto**	**A nanosleep**	**exec**	**A sem_wait far**	**A nanosleep**		
interrupt da RT_clock e scadenza timer di nanosleep per **R**	1							
	2							
	3							
	4							
	5	pronto	A	A	A	A	exec	
	6	pronto	exec	A	A	A	pronto	
	7							
	8							
	9							
	10							
50 interrupt da std_in, tutti i caratteri trasferiti	11							
	12							
	13	pronto	NE	A	A	NE	exec	
	14							

seconda parte – moduli, pila e strutture dati HW

Si consideri un processo **P** in esecuzione in modo U della funzione *main.* La figura sotto riportata descrive compiutamente, ai fini dell'esercizio, il contesto di **P** in modo U.



Un processo **Q** è in attesa di un evento. I processi **P** e **Q** sono gli unici di interesse nel sistema.

evento A&B: già risolto

A. Durante l'esecuzione del codice utente, si verifica l'interrupt **Interrupt_1**, che manda in esecuzione la routine di risposta a interrupt **R_int_1**.

B. Durante l'esecuzione della routine di risposta **R_int_1** si verifica l'interrupt **Interrupt_2**, che viene accettato e manda in esecuzione la routine di risposta a interrupt **R_int_2**.

Le tabelle sottostanti mostrano la situazione di interesse subito dopo il verificarsi dell'evento **A&B**.

processo P	
PC	// non di interesse
SP	Z – 4
SSP	Z
USP	W
descrittore di P.stato	PRONTO

sPila di P
PSR (S)
a R_int_1 da R_int_2
PSR (U)
a codice utente da R_int_1

RUNQUEUE	
CURR	P
RB.LFT	NULL

Si consideri ora la seguente **serie di eventi**.

evento C

La routine di risposta a interrupt **R_int_2** risveglia il processo **Q**, che viene portato in stato di pronto. Il processo **Q** ha maggiori diritti di esecuzione rispetto al processo **P**.

Completare le tabelle seguenti con i valori assunti dagli elementi **subito dopo l'esecuzione dell'istruzione IRET che termina** la routine di risposta a interrupt **R_int_2**.

processo P	
PC	// non di interesse
SP	
SSP	
USP	
descrittore di P.stato	

sPila di P

RUNQUEUE	
CURR	
RB.LFT	

evento D

Come detto sopra, il processo Q ha maggiori diritti di esecuzione rispetto al processo P.

Completare le tabelle seguenti con i valori assunti dagli elementi **subito dopo la ripresa dell'esecuzione in modo U del processo Q.**

descrittore processo P	
descrittore di P.**SP**	
descrittore di P.**stato**	

sPila di P

RUNQUEUE	
CURR	
RB.LFT	

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, e il processo P è in esecuzione):

```
=== STATO INIZIALE ====

PROCESSO: P  ************************************************************
    VMA : C  000000400,  2,  R,  P,  M,  <X, 0>
          D  000000600,  4,  W,  P,  A,  <-1,0>
          P  7FFFFFFFC,  3,  W,  P,  A,  <-1,0>

PT:  <c0 :1  R>  <c1 :-  ->  <d0 :5  R>  <d1 :7  W>  <d2 :s0 R>  <d3 :-  ->
     <p0 :6  W>  <p1 :4  R>  <p2 :-  ->

process P - NPV of PC and SP:  c0, p1

PROCESSO: R  ************************************************************
    VMA : C  000000400,  2,  R,  P,  M,  <X, 0>
          D  000000600,  4,  W,  P,  A,  <-1,0>
          P  7FFFFFFFC,  3,  W,  P,  A,  <-1,0>

PT:  <c0 :1  R>    <c1 :-  ->  <d0 :5  R>  <d1 :-  ->  <d2 :s0 R>  <d3 :-  ->
     <p0 :2  D W>  <p1 :4  R>  <p2 :-  ->

process R - NPV of PC and SP:  c0, p0

____MEMORIA FISICA____(pagine libere: 3)_____________________________
      00 : <ZP>                ||  01 : Pc0 / Rc0 / <X,0>   ||
      02 : Rp0 D               ||  03 : ----                ||
      04 : Pp1 / Rp1           ||  05 : Pd0 / Rd0           ||
      06 : Pp0                 ||  07 : Pd1                 ||
      08 : ----                ||  09 : ----                ||

____STATO del TLB_____________________________________________________
      Pc0 : 01 -  0: 1:        ||   Pp0 : 06 -  1: 1:       ||
            -----              ||   Pp1 : 04 -  1: 1:       ||
      Pd0 : 05 -  1: 0:        ||   Pd1 : 07 -  1: 0:       ||
            -----              ||         -----             ||

SWAP FILE:    Pd2 / Rd2, ----, ----, ----, ----, ----,

LRU ACTIVE:   PP0, PC0, PP1,

LRU INACTIVE: pd1, pd0, rd0, rp0, rc0, rp1,
```

evento 1 – *read* (Pd2)

PT del processo: **P**				
c0:	d0:	d1:	d2:	d3:
p0:	p1:	p2:		
PT del processo: **R**				
c0:	d0:	d1:	d2:	d3:
p0:	p1:	p2:		

MEMORIA FISICA	
00: <ZP>	01:
02:	03:
04:	05:
06:	07:
08:	09:

SWAP FILE	
s0:	s1:
s2:	s3:
s4:	s5:

Active: _____ **Inactive:** _____

evento 2 – *write* (Pp2)

PT del processo: **P**				
c0:	d0:	d1:	d2:	d3:
p0:	p1:	p2:	p3:	

process P - NPV of PC and SP:

PT del processo: **R**				
c0:	d0:	d1:	d2:	d3:
p0:	p1:	p2:		

process R - NPV of PC and SP:

MEMORIA FISICA	
00: <ZP>	01:
02:	03:
04:	05:
06:	07:
08:	09:

SWAP FILE	
s0:	s1:
s2:	s3:
s4:	s5:

Active: _____ **Inactive:** _____

seconda parte – file system

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

MAXFREE = 3 **MINFREE = 1**

Si consideri la seguente **situazione iniziale**:

```
____MEMORIA FISICA____(pagine libere: 4)_____________________________
      00 : <ZP>                ||  01 : Pc0 / Qc0 / <X,0>   ||
      02 : Qp0  D              ||  03 : Pp0                 ||
      04 : ----                ||  05 : ----                ||
      06 : ----                ||  07 : ----                ||
```

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

ATTENZIONE: nella tabella di descrizione del file **è presente la colonna "processo" in cui va specificato il nome del processo a cui si riferiscono le informazioni "f_pos" e "f_count" (campi di struct file) relative al file indicato**

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

È in esecuzione il processo **P**.

eventi 1 e 2 – fd = *open* (F), fd1 = *open* (G)

processo	file	f_pos	f_count	numero pagine lette	numero pagine scritte
	F				
	G				

evento 3 – *write* (fd, 12000)

MEMORIA FISICA	
00: <ZP>	01:
02:	03:
04:	05:
06:	07:

processo	file	f_pos	f_count	numero pagine lette	numero pagine scritte
	F				

evento 4 – *write* (fd1, 8000)

MEMORIA FISICA	
00: <ZP>	01:
02:	03:
04:	05:
06:	07:

processo	file	f_pos	f_count	numero pagine lette	numero pagine scritte
	F				
	G				

eventi 5, 6 e 7 – *context_switch* (Q), fd2 = *open* (F), *read* (fd2, 200)

NOTA BENE: **al momento del context switch, la pagina p0 di P è marcata dirty nel TLB**

MEMORIA FISICA	
00: <ZP>	01:
02:	03:
04:	05:
06:	07:

processo	file	f_pos	f_count	numero pag. lette	numero pag. scritte
	F				
	F				

eventi 8, 9 e 10 – *context_switch* (P), *lseek* (fd, –4000), *write* (fd, 100)

MEMORIA FISICA	
00: <ZP>	01:
02:	03:
04:	05:
06:	07:

processo	file	f_pos	f_count	numero pag. lette	numero pag. scritte
	F				
	F				
	G				