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 – giovedì 10 settembre 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 (?? punti) _____

esercizio 2 (?? punti) _____

esercizio 3 (?? 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 neutral
sem_t one, zero
int global = 0
```

```
void * min (void * arg) {
   mutex_lock (&neutral)
      sem_wait (&one)
      global = 1                                   /* statement A */
      sem_wait (&one)
      sem_post (&zero)                             /* statement B */
   mutex_unlock (&neutral)
   return NULL
} /* end min */
```

```
void * max (void * arg) {
   mutex_lock (&neutral)
      sem_post (&one)
   mutex_unlock (&neutral)                         /* statement C */
   global = 2
   mutex_lock (&neutral)
      sem_wait (&zero)
   mutex_unlock (&neutral)
   global = 3                                      /* statement D */
   return NULL
} /* end max */
```

```
void main ( ) {
   pthread_t th_1, th_2
   sem_init (&one, 0, 1)
   sem_init (&zero, 0, 0)
   create (&th_1, NULL, min, NULL)
   create (&th_2, NULL, max, NULL)
   join (th_1, NULL)                               /* 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 – min	**th_2** – max
subito dopo stat. **A**	Esiste	Può esistere
subito dopo stat. **B**	Esiste	Esiste
subito dopo stat. **C**	Può esistere	Esiste
subito dopo stat. **E**	Non esiste	Può esistere

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**		
	neutral	*one*	*zero*
subito dopo stat. **A**	1	1 - 0	0
subito dopo stat. **C**	1 - 0	2 - 1 - 0	0 - 1
subito dopo stat. **D**	0	0	0
subito dopo stat. **E**	1 - 0	0	1 - 0

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** – min	**th_2** – max	*global*
1	sem_wait(&one) (2°)	mutex_lock(&neutral)	1
2	mutex_lock(&neutral)	sem_wait(&zero)	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 da Q
      pid2 = fork ( )                 // Q crea R
      if (pid2 == 0) {                // codice eseguito da R
         read (stdin, msg, 50)
         exit (2)
         } else {                     // codice eseguito da Q
            pid = wait (&status)      // Q aspetta la terminazione di R
         } // end_if pid2
     } else {                         // codice eseguito da P
         execl ("/acso/prog_x", "prog_x", NULL)
         exit (-1)
   } // end_if pid1
   exit (0)                           // codice eseguito da Q
} // prova
```

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

void * more (void * arg) {		void * less (void * arg) {
mutex_lock (&positive)		mutex_lock (&positive)
sem_wait (&one)		sem_wait (&one)
mutex_unlock (&positive)		sem_wait (&one)
sem_post (&one)		mutex_unlock (&positive)
sem_wait (&one)		sem_post (&one)
return NULL		return NULL
} // more		} // less

```
main ( ) {  // codice eseguito da P
    pthread_t th_1, th_2
    sem_init (&one, 0, 2)
    create (&th_1, NULL, more, NULL)
    create (&th_2, NULL, less, NULL)
    join (th_2, NULL)
    join (th_1, NULL)
    exit (1)
} // main
```

Un processo ***P*** esegue il programma ***prova***. Il processo ***P*** crea il processo figlio ***Q*** e poi il processo ***P*** esegue i thread ***th_1*** e ***th_2***. Il processo ***Q*** crea un processo figlio ***R***.

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 ***P*** **ha già eseguito** la primitiva **`create (&th_1, …)` ma non** la primitiva **`create (&th_2, …)`**
- il processo (thread) ***th_1*** **ha già eseguito** la primitiva **`unlock (&positive)` ma non** la primitiva **`sem_post (&one)`**

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**	TH2	R	
evento oppure *processo-chiamata*	*PID*	**1**	**2**	3	4	5	6	
	TGID	**1**	**2**	3	2	2	6	
Q – pid2 = fork (Q crea R)	0	pronto	pronto (da più tempo)	esec	pronto	NE	NE	
interrupt da *RT_clock* e scadenza quanto di tempo per il processo in esecuzione	1	pronto	ESEC	pronto	pronto	NE	NE	
P - pthread_create(TH2)	2	pronto	ESEC	pronto	pronto	pronto	NE	
P - pthread_join(TH2)	3	pronto	A join	pronto	ESEC	pronto	NE	
th_1 – *sem_post (one)* dopo l'invocazione il semaforo *one* vale 2	4	pronto	A join	pronto	ESEC	pronto	NE	
interrupt da *RT_clock* e scadenza quanto di tempo per il processo in esecuzione	5	pronto	A join	ESEC	pronto	pronto	NE	
Q - fork	6	pronto	A join	ESEC	pronto	pronto	pronto	
Q - wait(R)	7	pronto	A join	A wait	pronto	ESEC	pronto	
TH2 - mutex_lock(&positive)	8	pronto	A join	A wait	pronto	ESEC	pronto	
TH2 - sem_wait(&one)	9	pronto	A join	A wait	pronto	ESEC	pronto	
TH2 - sem_wait(&one)	10	pronto	A join	A wait	pronto	ESEC	pronto	
TH2 - mutex_unlock(&positive)	11	pronto	A	A	pronto	esec	pronto	
TH2 - sem_post(&one)	12	pronto	A join	A wait	pronto	ESEC	pronto	
TH2 - return	13	pronto	A join	A wait	ESEC	NE	pronto	
TH1 - sem_wait(&one)	14	pronto	A join	A wait	A sem	NE	ESEC	

seconda parte – scheduling CFS

Si consideri uno Scheduler CFS con **3 task** caratterizzato da queste condizioni iniziali (**da completare**):

CONDIZIONI INIZIALI (da completare)							
RUNQUEUE	**NRT**	**PER**	**RQL**	**CURR**	**VMIN**		
	3	6	4	t1	100		
TASK	**ID**	**LOAD**	**LC**	**Q**	**VRTC**	**SUM**	**VRT**
CURRENT	t1	1	0.25	1.5	1	50	101.0
RB	t2	1	0.25	1.5	1	60	102.0
	t3	2	0.5	3	0.5	70	103.0

Durante l'esecuzione dei task si verificano i seguenti eventi:

```
Events of task t1:  EXIT at 2.0;
Events of task t2:  WAIT at 1.0;   WAKEUP after 1.0;
Events of task t3:  WAIT at 1.5;   WAKEUP after 0.5;
```

Simulare l'evoluzione del sistema per **4 eventi** riempiendo le seguenti tabelle.

Indicare la valutazione delle condizioni di preemption per l'evento di WAKEUP nell'apposito spazio alla fine dell'esercizio.

EVENTO		**TIME**	**TYPE**	**CONTEXT**	**RESCHED**	
		1.5	s.q.d.t	T1	TRUE	T1 -> VRT = 101 + 1.5 * 1 = 102.5
RUNQUEUE	**NRT**	**PER**	**RQL**	**CURR**	**VMIN**	
	3	6	4	T2	102	

TASK	**ID**	**LOAD**	**LC**	**Q**	**VRTC**	**SUM**	**VRT**
CURRENT	T2	1	0.25	1.5	1	60	102
RB	T1	1	0.25	1.5	1	51.5	102.5
	T3	2	0.5	3	0.5	70	103
WAITING							

EVENTO		**TIME**	**TYPE**	**CONTEXT**	**RESCHED**	
		2.5	wait	T2	TRUE	T2 -> VRT = 102 + 1 * 1 = 103
RUNQUEUE	**NRT**	**PER**	**RQL**	**CURR**	**VMIN**	
	2	6	3	T1	102.5	

TASK	**ID**	**LOAD**	**LC**	**Q**	**VRTC**	**SUM**	**VRT**
CURRENT	T1	1	1/3	2	1	51.5	102.5
RB	T3	2	2/3	4	0.5	70	103
WAITING	T2	1				62.5	103

EVENTO		TIME	TYPE	CONTEXT	RESCHED	
		3	exit	T1	TRUE	T1 -> VRT = 102. 5 + 0.5 * 1 = 103
RUNQUEUE	NRT	PER	RQL	CURR	VMIN	
	1	6	2	T3	103	

TASK	ID	LOAD	LC	Q	VRTC	SUM	VRT
CURRENT	T3	2	1	6	0.5	70	103
RB							
WAITING	T2	1				62.5	103

EVENTO		TIME	TYPE	CONTEXT	RESCHED	
		3.5	wake up	T3	TRUE	T3 -> VRT = 103 + 0.5 * 0.5 = 103.25
RUNQUEUE	NRT	PER	RQL	CURR	VMIN	T2 -> VRT = 103
	2	6	6	T2	103.25	

TASK	ID	LOAD	LC	Q	VRTC	SUM	VRT
CURRENT	T2	1	1/3	2	1	62.5	103
RB	T3	2	2/3	4	0.5	70.5	103.25
WAITING							

Condizioni di rescheduling a **wake_up** del **task t2**:

wake_up:	103 + 1 * 1/3 = 103.33 > 103.25 => TRUE

Condizioni di rescheduling a **wake_up** del **task t3**:

wake_up:

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 = 2 **MINFREE = 1**

Situazione iniziale (esistono due processi P e Q)

```
PROCESSO: P  ********************************************************
    VMA : C  000000400,  1 , R  ,  P  ,  M  , <X, 0>
          S  000000600,  2 , W  ,  P  ,  M  , <X, 1>
          D  000000602,  2 , W  ,  P  ,  A  , <-1,0>
          M0 000001000,  4 , W  ,  P  ,  M  , <F, 0>
          M1 000002000,  4 , W  ,  S  ,  M  , <G, 0>
          P  7FFFFFFC,  3 , W  ,  P  ,  A  , <-1,0>

    PT: <c0 :1 R>  <s0 :- ->  <s1 :- ->  <d0 :- ->  <d1 :- ->  <p0 :2 R>
        <p1 :4 W>  <p2 :- ->  <m00:6 W>  <m01:- ->  <m02:- ->  <m03:- ->
        <m10:3 W>  <m11:- ->  <m12:- ->  <m13:- ->

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

PROCESSO: Q  ****(i dettagli di questo processo non interessano) *******


____MEMORIA FISICA____(pagine libere: 2)_____________________________
      00 : <ZP>                 ||  01 : Pc0 / Qc0 / <X,0>   ||
      02 : Pp0 / Qp0            ||  03 : Pm10 / <G,0>        ||
      04 : Pp1                  ||  05 : ----                ||
      06 : Pm00                 ||  07 : ----                ||

____STATO del TLB_____________________________________________________
      Pc0  : 01 -  0: 1:        ||   Pp0  : 02 -  1: 0:      ||
      Pp1  : 04 -  1: 0:        ||   Pm00 : 06 -  1: 0:      ||
      Pm10 : 03 -  1: 0:        ||         -----             ||
            -----               ||         -----             ||

SWAP FILE:    Qp1  , ----, ----, ----, ----, ----, ----, ----,
LRU ACTIVE:   PC0,
LRU INACTIVE: pp1, pm10, pm00, pp0, qp0, qc0,
```

evento 1: *write* (Pp0)

PT del processo: **P**				
P0: :2 W	P1: :4 W	P2: :- -	M00: :6 W	M10: :3 W

process P - NPV of PC and SP:

MEMORIA FISICA	
00: <ZP>	01: Pc0 / Qc0 / <X, 0>
02: Pp0	03: Pm10 / <G, 0>
04: Pp1	05: Qp0
06: Pm00	07:

SWAP FILE	
s0: Qp1	s1:
s2:	s3:

evento 2: *write* (Ps1)

MEMORIA FISICA	
00: <ZP>	01: Pc0 / Qc0 / <X, 0>
02: Ps1	03: Pm10 / <G, 0>
04: Pp1	05: <X, 2>
06: Pm00	07:

SWAP FILE	
s0: Qp1	s1: Qp0
s2: Pp0	s3:

LRU ACTIVE: PS1, PC0

LRU INACTIVE: pp1, pm10, pm00, qc0

evento 3: *write* (Pp2, Pp3)

MEMORIA FISICA	
00: <ZP>	01: Pc0 / Qc0 / <X, 0>
02: Ps1	03: Pm10 / <G, 0>
04: Pp1	05: Pp2
06: Pp3	07:

SWAP FILE	
s0: Qp1	s1: Qp0
s2: Pp0	s3: Pm00
s4:	s5:

evento 4: *write* (Pp4)

ACTIVE: PP3, PP2, PS1, PC0 INACTIVE: pp1, pm10, qc0

MEMORIA FISICA	
00: <ZP>	01: Pc0 / Qc0 / <X, 0>
02: Ps1	03: Pp4
04: - - - -	05: Pp2
06: Pp3	07:

SWAP FILE	
s0: Qp1	s1: Qp0
s2: Pp0	s3: Pm00
s4: Pp1	s5:

LRU ACTIVE: PP4, PP3, PP2, PS1, PC0

LRU INACTIVE: qc0

process P - NPV of PC and SP: c0, p4

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: 5)_____________________________
      00 : <ZP>                  ||  01 : Pc2 / <X, 2>        ||
      02 : Pp0                   ||  03 : ----                ||
      04 : ----                  ||  05 : ----                ||
      06 : ----                  ||  07 : ----                ||
```

Per ognuno 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.

È in esecuzione il processo **P**.

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 fcount diventa = 0.

Eventi 1 e 2: fd = *open* (“F”), *write* (fd, 6000)

MEMORIA FISICA	
00: <ZP>	01: Pc2 / <X, 2>
02: Pp0	03: <F, 0> (D)
04: <F, 1> (D)	05:
06:	07:

	f_pos	f_count	numero pagine lette	numero pagine scritte
file F	6000	1	2	0

Evento 3: *close* (fd)

MEMORIA FISICA	
00: <ZP>	01: Pc2 / <X, 2>
02: Pp0	03: <F, 0>
04: <F, 1>	05:
06:	07:

	f_pos	f_count	numero pagine lette	numero pagine scritte
file F	----	0	2	2

Eventi 4 e 5: *fork* (“Q”), *context_switch* (“Q”)

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

MEMORIA FISICA	
00: <ZP>	01: Pc2 / Qc2 / <X, 2>
02: Pp0 (D)	03: <F, 0>
04: <F, 1>	05: Qp0 (D)
06:	07:

	f_pos	f_count	numero pagine lette	numero pagine scritte
file F	- - - -	0	2	2

Eventi 6 e 7: fd = *open* (“F”), *read* (8000)

	f_pos	f_count	numero pagine lette	numero pagine scritte
file F	8000	1	2	2

Evento 8: *write* (fd, 10000)

MEMORIA FISICA	
00: <ZP>	01: Pc2 / Qc2 / <X, 2>
02: Pp0 (D)	03: <F, 3> (D)
04: <F, 4> (D)	05: Qp0 (D)
06:	07:

	f_pos	f_count	numero pagine lette	numero pagine scritte
file F	18000	1	5	4

Evento 9: *close* (fd)

	f_pos	f_count	numero pagine lette	numero pagine scritte
file F	- - - -	0	5	6