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

Prova di lunedì 31 gennaio 2022

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 **2 h : 00 m**

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

esercizio 1 (4 punti) _____

esercizio 2 (5 punti) _____

esercizio 3 (5 punti) _____

esercizio 4 (2 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 sense
sem_t see, hear
int global = 0
```

```
void * hand (void * arg) {
    mutex_lock (&sense)
    sem_post (&see)
    global = 1                                          /* statement A */
    mutex_unlock (&sense)
    global = 2
    sem_wait (&hear)
    mutex_lock (&sense)
    sem_wait (&hear)
    mutex_unlock (&sense)
    return NULL
} /* end hand */
```

```
void * foot (void * arg) {
    mutex_lock (&sense)
    sem_wait (&see)
    global = 3                                          /* statement B */
    mutex_unlock (&sense)
    sem_wait (&hear)
    global = 4                                          /* statement C */
    mutex_lock (&sense)
    sem_post (&hear)
    mutex_unlock (&sense)
    return (void *) 5
} /* end foot */
```

```
void main ( ) {
    pthread_t th_1, th_2
    sem_init (&see, 0, 0)
    sem_init (&hear, 0, 2)
    create (&th_1, NULL, hand, NULL)
    create (&th_2, NULL, foot, NULL)
    join (th_1, NULL)                                   /* statement D */
    join (th_2, &global)
    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 – *hand*	**th_2 – *foot***
subito dopo stat. **A**	*ESISTE*	*PUÒ ESISTERE*
subito dopo stat. **B**	*PUÒ ESISTERE*	*ESISTE*
subito dopo stat. **C**	*ESISTE*	*ESISTE*
subito dopo stat. **D**	*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**			
	sense	*see*	*hear*	*global*
subito dopo stat. **A**	*1*	*1*	*2*	*1*
subito dopo stat. **B**	*1*	*0*	*0 / 1 / 2*	*2 / 3*
subito dopo stat. **C**	*0 / 1*	*0*	*0 / 1*	*2 / 4*
subito dopo stat. **D**	*0 / 1*	*0 / 1*	*0*	*2 / 3 / 4*

Il sistema può andare in stallo (*deadlock*), con uno o più *thread* che si bloccano, in (almeno) **tre 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 – *hand***	**th_2 – *foot***	*global*
1	`1a lock sense`	`wait see`	*0*
2	`2a wait hear`	`2a lock sense`	*2, 4*
3	`--`	`wait hear`	*2, 3*

esercizio n. 2 – processi e nucleo

prima parte – gestione dei processi

```
// programma main.c
int main ( ) {                    // processo P
     pid_t pid = fork ( )
     if (pid == 0) {              // codice eseguito solo da Q
        execl ("/acso/nuovo", "nuovo", NULL)
        exit (-1)
     } // end if
     write (stdout, "Fin!", 4)
     wait (&status)
     exit (0)
} // end main.c
```

```
// programma nuovo.c
sem_t pass
mutex_t lock
char vet [2] = {1 ,2}
```

```
void * one( void * arg) {
  sem_wait (&pass)
  write (stdout, vet [0], 1)
  sem_post (&pass)
  return NULL
} // end one
```

```
void * two (void * arg) {
  mutex_lock (&lock)
  read (stdin, vet [1], 1)
  mutex_unlock (&lock)
  sem_post (&pass)
  return NULL
} // end two
```

```
int main ( ) { // codice eseguito da Q
   pthread_t TH_1, TH_2
   sem_init (&pass, 0, 1)
   pthread_create (&TH_2, NULL, two, NULL)
   pthread_create (&TH_1, NULL, one, NULL)
   pthread_join (TH_2, NULL)
   pthread_join (TH_1, NULL)
   exit (1)
} // fine nuovo.c
```

Un processo **P** esegue il programma **main.c** e crea un processo figlio **Q** che esegue una mutazione di codice (programma **nuovo.c**). La mutazione di codice va a buon fine e viene creata una coppia di thread **TH_1** e **TH_2**. Si simuli l'esecuzione dei vari processi completando tutte le righe presenti nella tabella così come risulta dal codice dato, dallo stato iniziale e dagli eventi indicati.

Si completi la tabella seguente riportando:

- ⟨ PID, TGID ⟩ di ciascun processo (normale o thread) che viene creato
- ⟨ evento oppure identificativo del processo-chiamata di sistema / libreria ⟩ nella prima colonna, dove necessario e in funzione del codice proposto (le istruzioni da considerare sono evidenziate in grassetto)
- 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

identificativo simbolico del processo		**Idle**	**P**	**Q**	**TH_1**	**TH_2**
evento oppure processo-chiamata	**PID**	1	*2*	*3*	*5*	*4*
	TGID	1	*2*	*3*	*3*	*3*
P – pid = fork ( )	**0**	**pronto**	**esec**	**pronto**	**NE**	**NE**
P – write	**1**	*pronto*	*attesa (write)*	*esec*	*NE*	*NE*
Q – execl	**2**	*pronto*	*attesa (write)*	*esec*	*NE*	*NE*
Q – pthread_create TH_2	**3**	*pronto*	*attesa (write)*	*esec*	*NE*	*pronto*
interrupt da RT_clock e scadenza quanto di tempo	**4**	*pronto*	*attesa (write)*	*pronto*	*NE*	*esec*
TH_2 – mutex_lock	**5**	*pronto*	*attesa (write)*	*pronto*	*NE*	*esec*
TH_2 – read	**6**	*pronto*	*attesa (write)*	*esec*	*NE*	*attesa (read)*
Q – pthread_create TH_1	**7**	**pronto**	**attesa (write)**	**esec**	**pronto**	**attesa (read)**
Q – pthread_join TH_2	**8**	*pronto*	*attesa (write)*	*attesa (join T2)*	*esec*	*attesa (read)*
TH_1 – sem_wait	**9**	*pronto*	*attesa (write)*	*attesa (join T2)*	*esec*	*attesa (read)*
TH_1 – write	**10**	*esec*	*attesa (write)*	*attesa (join T2)*	*attesa (write)*	*attesa (read)*
interrupt da stdin, tutti i caratteri richiesti da TH_2 trasferiti	**11**	*pronto*	*attesa (write)*	*attesa (join T2)*	*attesa (write)*	*esec*
TH_2 – mutex_unlock	**12**	*pronto*	*attesa (write)*	*attesa (join T2)*	*attesa (write)*	*esec*
TH_2 – sem_post	**13**	*pronto*	*attesa (write)*	*attesa (join T2)*	*attesa (write)*	*esec*
TH_2 – return	**14**	*pronto*	*attesa (write)*	*esec*	*attesa (write)*	*NE*
interrupt da stdout, tutti i caratteri inviati da TH_1 trasferiti	**15**	*pronto*	*attesa (write)*	*pronto*	*esec*	*NE*
TH_1 – sem_post	**16**	*pronto*	*attesa (write)*	*pronto*	*esec*	*NE*
TH_1 – return	**17**	*pronto*	*attesa (write)*	*esec*	*NE*	*NE*
Q – pthread_join TH_2	**18**	*pronto*	*attesa (write)*	*esec*	*NE*	*NE*
Q – exit	**19**	*esec*	*attesa (write)*	*NE*	*NE*	*NE*

seconda parte – scheduling dei processi

Si consideri uno scheduler CFS con **tre task** caratterizzato da queste condizioni iniziali (già complete):

CONDIZIONI INIZIALI (già complete)							
RUNQUEUE	**NRT**	**PER**	**RQL**	**CURR**	**VMIN**		
	2	6	3	T1	100		
TASK	**ID**	**LOAD**	**LC**	**Q**	**VRTC**	**SUM**	**VRT**
CURRENT	T1	1	0,33	2	1	10	100
RB	T2	2	0,67	4	0,5	20	101

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

```
Events of task t1:   WAIT  at 1.0        WAKEUP after 3.5

Events of task t2:   CLONE at 2.0
```

Simulare l'evoluzione del sistema per **quattro eventi** riempiendo le seguenti tabelle (per indicare le condizioni di rescheduling e altri calcoli eventualmente richiesti, utilizzare le tabelle finali):

EVENTO 1	**TIME**	**TYPE**	**CONTEXT**	**RESCHED**		
	1	*WAIT*	*T1*	*vero*		
RUNQUEUE	**NRT**	**PER**	**RQL**	**CURR**	**VMIN**	
	1	6	*2*	*T2*	*101*	
TASK	**ID**	**LOAD**	**LC**	**Q**	**VRTC**	**SUM**	**VRT**
CURRENT	*T2*	*2*	*1*	*6*	*0,5*	*20*	*101*
RB							
WAITING	*T1*	*1*				*11*	*101*

EVENTO 2	**TIME**	**TYPE**	**CONTEXT**	**RESCHED**		
	3	*CLONE*	*T2*	*falso*		
RUNQUEUE	**NRT**	**PER**	**RQL**	**CURR**	**VMIN**	
	2	6	*4*	*T2*	*102*	
TASK	**ID**	**LOAD**	**LC**	**Q**	**VRTC**	**SUM**	**VRT**
CURRENT	*T2*	*2*	*0,5*	*3*	*0,5*	*22*	*102*
RB	*T3*	*2*	*0,5*	*3*	*0,5*	*0*	*103,50*
WAITING	*T1*	*1*				*11*	*101*

EVENTO 3 (*)		TIME	TYPE	CONTEXT	RESCHED	(*) E' stata considerata corretta anche la soluzione che prevede che T2 CURR non rientri subito nella lista RB e pertanto T3 diventa CURR	
		4	*Q_scade*	*T2*	*vero*		
RUNQUEUE	**NRT**	**PER**	**RQL**	**CURR**	**VMIN**		
	2	6	*4*	*T2*	*102,50*		
TASK	**ID**	**LOAD**	**LC**	**Q**	**VRTC**	**SUM**	**VRT**
CURRENT	*T2*	*2*	*0,5*	*3*	*0,5*	*23*	*102,50*
RB	*T3*	*2*	*0,5*	*3*	*0,5*	*0*	*103,50*
WAITING	*T1*	*1*				*11*	*101*

EVENTO 4		TIME	TYPE	CONTEXT	RESCHED		
		4,5	*WUP*	*T2*	*vero*		
RUNQUEUE	**NRT**	**PER**	**RQL**	**CURR**	**VMIN**		
	3	6	*5*	*T1*	*102,75*		
TASK	**ID**	**LOAD**	**LC**	**Q**	**VRTC**	**SUM**	**VRT**
CURRENT	*T1*	*1*	*0,2*	*1,2*	*1*	*11*	*101*
RB	*T2*	*2*	*0,4*	*2,4*	*0,5*	*23,5*	*102,75*
	T3	*2*	*0,4*	*2,4*	*0,5*	*0*	*103,50*
WAITING							

Calcolo del VRT iniziale del **task T3** creato dalla **CLONE** *eseguita dal **task T2**:*

$T3.VRT\ (iniziale) = VMIN + T3.Q \times T3.VRTC = 102 + 3 \times 0{,}5 = 103{,}5$

Valutazione della cond. di rescheduling alla **CLONE** *eseguita dal **task T2**:*

$T3.VRT + WGR \times T3.LC < T2.VRT \Rightarrow 103{,}50 + 1 \times 0{,}50 = 104 < 102 \Rightarrow$ *falso*

Valutazione della cond. di rescheduling alla **WAKEUP** *eseguita dal **task T1**:*

$T\ 1.VRT + WGR \times T1.LC < T2.VRT \Rightarrow 101 + 1 \times 0{,}20 = 101{,}20 < 102{,}75 \Rightarrow$ *vero*

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 Q)

```
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  7FFFFFFC,  3 , W  ,  P  ,  A  , <-1, 0>

    PT: <c0 :- ->      <c1 :1  R>   <k0 :- ->   <s0 :- ->   <p0 :2  D R>
        <p1 :4  D W>   <p2 :- ->

process P - NPV of PC and SP:  c1, p1

PROCESSO: Q  ****SOLO LE INFORMAZIONI RILEVANTI PER L'ESERCIZIO ******
    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>
          M0 000010000,  4 , W  ,  S  ,  M  , <G,  0>
          P  7FFFFFFC,  3 , W  ,  P  ,  A  , <-1, 0>

process Q - NPV of PC and SP:  c1, p1

____MEMORIA FISICA____(pagine libere: 3)_____________________________
      00 : <ZP>                ||  01 : Pc1 / Qc1 / <XX, 1> ||
      02 : Pp0 / Qp0  D        ||  03 : Qp1  D              ||
      04 : Pp1  D              ||  05 : Qm00 / <G, 0>  D    ||
      06 : Qm01 / <G, 1>  D    ||  07 : ----                ||
      08 : ----                ||  09 : ----                ||

____STATO del TLB_____________________________________________________
      Pc1 : 01 -  0: 1:        ||  Pp1 : 04 -  1: 1:        ||
            -----              ||        -----              ||
            -----              ||        -----              ||
            -----              ||        -----              ||

SWAP FILE:    ----, ----, ----, ----, ----, ----,
LRU ACTIVE:   QM01, QM00, QC1,
LRU INACTIVE: qp1,  qp0,  pp1,  pp0,  pc1,
```

evento 1: *read* (Pc1) – *write* (Pp2)

La pagina virtuale Pc1 da leggere è in memoria fisica. La pagina virtuale Pp2 da scrivere è di growsdown (non ancora allocata in memoria fisica), pertanto si modifica la VMA di pila del task P (aggiungendo la pagina virtuale Pp3 come pagina di growsdown e allocandola in ZP), poi si ha COW da ZP, si alloca la pagina virtuale Pp2 in pagina fisica 07 (e la si registra nel TLB marcandola D e A), e la pagina Pp2 viene inserita in lista Active (in testa con referenza). Il registro SP del processore viene aggiornato di conseguenza.

PT del processo: **P**				
p0: *2 D R*	p1: *4 D W*	p2: *7 W*	*p3: - -*	

process P	NPV of **PC**: *c1*	NPV of **SP**: *p2*

MEMORIA FISICA	
00: <ZP>	01: *Pc1 / Qc1 / <XX, 1>*
02: *Pp0 / Qp0 D*	03: *Qp1 D*
04: *Pp1 D*	05: *Qm00 / <G, 0> D*
06: *Qm01 / <G, 1> D*	07: *Pp2*
08: -----	09: -----

LRU ACTIVE: *PP2, QM01, QM00, QC1,* _____

LRU INACTIVE: *qp1, qp0, pp1, pp0, pc1,* _____

evento 2: *read* (Pc1, Pp0) – *write* (Pp3)

Le pagine virtuali Pc1 e Pp0 da leggere sono già in memoria fisica. La pagina virtuale Pp3 da scrivere è di growsdown (non ancora allocata in memoria fisica), pertanto si modifica la VMA di pila del task P (aggiungendo la pagina virtuale Pp4 come pagina di growsdown e allocandola in ZP), poi si dovrebbe allocare una pagina fisica, ma poiché minfree vale 2 e free vale 2, si chiama PFRA che libera due pagine da lista Inactive, ossia prima la 04 (pp1) e poi la 02 (pp0 / qp0) (si veda come sono ordinate in lista Inactive a partire dalla coda), le quali vanno certamente in swap file perché marcate D (e la pagina Pp1 viene tolta dal TLB), si aggiorna la lista Inactive (togliendo pp1 e pp0 / qp0), si alloca la pagina virtuale Pp3 in pagina fisica 02 (e la si registra nel TLB marcandola D e A), e la pagina Pp3 viene inserita in lista Active (in testa con referenza). Il registro SP del processore viene aggiornato di conseguenza.

PT del processo: **P**				
p0: *s1 R*	p1: *s0 W*	p2: *7 W*	p3: *2 W*	*p4:- -*

process P	NPV of **PC**: *c1*	NPV of **SP**: *p3*

MEMORIA FISICA	
00: <ZP>	01: *Pc1 / Qc1 / <XX, 1>*
02: ~~*Pp0 / Qp0*~~*Pp3*	03: *Qp1 D*
04: ~~*Pp1*~~----	05: *Qm00 / <G, 0> D*
06: *Qm01 / <G, 1> D*	07: *Pp2*
08: -----	09: -----

SWAP FILE	
s0: *Pp1*	s1: *Pp0 / Qp0*
s2: -----	s3: -----
s4: -----	s5: -----

LRU ACTIVE: *PP3, PP2, QM01, QM00, QC1,* _____

LRU INACTIVE: *qp1, pc1,* _____

evento 3: *clone* (S, c0)

Viene creata la VMA T0 (area di pila thread) per il thread S, con le proprietà W (scrivibile), P (privata) e A (anonima), tutte le pagine del task P vengono condivise (inseparabilmente) con S, e va allocata la pagina t00 (cima pila di S). Il task corrente resta P. La nuova pagina condivisa PSt00 viene allocata in pagina fisica 04 e si aggiornano le liste LRU, inserendo la nuova pagina PSt00 (in Active in testa con referenza). Viene aggiornato il TLB inserendo PSt00 (marcandola D e A), dopo avere riportate tutte le modifiche apportate al TLB partendo dal TLB iniziale (si ricordi che qui il TLB viene occupato per righe, riutilizzando le celle lasciate libere da PTE eventualmente cancellate – si vedano i commenti agli eventi precedenti relativi agli inserimenti e alle cancellazioni nel e dal TLB). Il registro SP per il task P, e i registri PC e SP per il task S, vengono aggiornati di conseguenza.

VMA del processo **P/S** (è da compilare solo la riga relativa alla VMA **T0**)							
AREA	NPV iniziale	dimensione	R/W	P/S	M/A	nome file	offset
T0	7FFF F77F E	2	W	P	A	-1	0

PT dei processi: **P/S**				
t00: 4 W	t01: - -			

process P	NPV of **PC**: c1	NPV of **SP**: p3
process S	NPV of **PC**: c0	NPV of **SP**: t00

TLB							
NPV	NPF	D	A	NPV	NPF	D	A
PSc1 : 01 -	0:	1:		~~Pp1~~PSp3 : 02 -	1:	1:	
PSp2 : 07 -	1:	1:		PSt00 : 04 -	1:	1:	
-----				-----			

LRU ACTIVE: PST00, PSP3, PSP2, QM01, QM00, QC1,_____

LRU INACTIVE: qp1, psc1,_____

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.**

process P – NPV of PC and SP: c2, p0

```
____MEMORIA FISICA____(pagine libere: 1)_____________________________
      00 : <ZP>                ||  01 : Pc2 / <X, 2>        ||
      02 : Pp0                 ||  03 : <G, 2>              ||
      04 : Pm00                ||  05 : <F, 0>  D           ||
      06 : <F, 1>  D           ||  07 : ----                ||

____STATO del TLB_____________________________________________________
      Pc2 :  01 -  0: 1:       ||   Pp0 : 02 -  1: 1:       ||
      Pm00 : 04 -  1: 1:       ||         -----             ||
             -----             ||         -----             ||
```

processo/i	file	f_pos	f_count	numero pag. lette	numero pag. scritte
P	F	6000	1	2	0

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

Il processo **P** è in esecuzione. Il file **F** è stato aperto da **P** tramite chiamata **fd1 = *open* (F).**

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 inoltre che la primitiva *close* scrive le pagine dirty di un file solo se *f_count* diventa = 0.

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

eventi 1 e 2: *fork* (Q) *context switch* (Q)

Va creato il task figlio Q, separando la pagina di pila da task padre P. Poiché vale minfree = 1 e free = 1, scatta PFRA per allocare (mediante COW) la pagina di pila del task figlio Q. Le pagine fisiche 03 (<FG, 2>) e 05 (<F, 0>) (sono pagine di page cache non in possesso di alcun task) vengono liberate e il blocco F0 viene scritto su disco (in quanto marcato D). La pagina fisica 02 resta allocata per la pagina di pila del task figlio Q e la pagina fisica 03 viene allocata come pagina di pila per il task padre P (entrambe vengono marcate D). Le pagine virtuali c2 e m00 restano condivise tra P e Q. Il file F aperto viene condiviso (f_count = 2). Il task Q diventa corrente. Una pagina è stata scritta su disco.

MEMORIA FISICA	
00: <ZP>	01: Pc2 / Qc2 / <X, 2>
02: *Qp0 D*	03: ~~*<G, 2>*~~*Pp0 D*
04: *Pm00 / Qm00 D*	05: ~~*<F, 0> D*~~----
06: *<F, 1> D*	07: ----

processo/i	file	f_pos	f_count	numero pag. lette	numero pag. scritte
P, Q	*F*	*6000*	*2*	*2*	*1*

evento 3: *read* (fd1, 500)

Si legge il file F e se ne aggiorna la posizione corrente, restando all'interno del blocco F1 (già presente in memoria fisica). Non cambia niente altro.

processo/i	file	f_pos	f_count	numero pag. lette	numero pag. scritte
P, Q	*F*	*6500*	*2*	*2*	*1*

eventi 4, 5 e 6: fd2 = *open* (F) *read* (fd2, 9000) *close* (fd1)

Il file F viene riaperto dal task Q, tramite una open indipendente, pertanto viene definita un nuova struttura dati "struct file" fd2 per F (solo di Q, con f_count = 1), con attributi (e posizione corrente) indipendenti da fd1. Il task Q legge i blocchi F0, F1 e F2 (la posizione finale 9000 si trova in blocco F2). Il blocco F0 viene (ri)caricato in pagina fisica 05, e il blocco F1 è già in pagina fisica 06. Dato che occorre una pagina fisica dove caricare il blocco F2, e poiché si ha minfree = 1 e ora vale free = 1, scatta PFRA che libera due pagine fisiche: 05 (nessuna scrittura su disco perché il blocco F0 appena ricaricato non è stato modificato) e 06 (scrittura del blocco F1 su disco perché è marcato D). Poi il blocco F2 viene caricato in pagina fisica 05. La close di fd1 da parte di Q fa decrementare il contatore f_count associato (fd1 resta solo del task P). Due pagine sono state lette da disco e una è stata scritta su disco.

MEMORIA FISICA	
00: <ZP>	01: Pc2 / Qc2 / <X, 2>
02: *Qp0 D*	03: *Pp0 D*
04: *Pm00 / Qm00 D*	05: ~~*<F, 0>*~~*<F, 2>*
06: ~~*<F, 1> D*~~----	07: ----

processo/i	file	f_pos	f_count	numero pag. lette	numero pag. scritte
P	*F*	*6500*	*1*	*4*	*2*
Q	*F*	*9000*	*1*		

esercizio n. 4 – domande varie

prima domanda – moduli del SO

stato iniziale: CURR = T, Q = PRONTO

La runqueue contiene due task: il thread **T** e il processo normale **Q**; il sistema non contiene altri task.

Si consideri il seguente evento: il **thread T** è in esecuzione in **modo U** ed esegue l'istruzione di **return** della sua funzione **"fun"** per terminare la sua esecuzione.

Domanda:

- Mostrare le **invocazioni di tutti i moduli** (ed eventuali relativi **ritorni**) eseguiti nel contesto del thread **T** per gestire l'evento indicato.
- Mostrare (in modo simbolico) il l'eveluzione dello **stack di sistema** del thread **T** al termine della gestione dell'evento considerato.

invocazione moduli – numero di righe non significativo

processo	modo	modulo
T	U	fun< // return di fun
T	U	clone // si torna a clone
T	*U*	*>syscall*
T	*U – S*	*SYSCALL: System_Call*
T	*S*	*>sys_exit*
T	*S*	*>wake_up_process*
T	*S*	*>check_preempt_curr<*
T	*S*	*wake_up_process<*
T	*S*	*>schedule*
T	*S*	*>pick_next_task<*
T – Q	*S*	*schedule: context_switch*

Nota: si ricordi che la funzione clone (chiamata in modo U da pthread_create) crea il thread chiamando (via syscall) il servizio sys_clone, poi si passa alla funzione di thread, e quando questa termina (con return), si elimina il thread chiamando (via syscall) il servizio sys_exit. Qui l'evento ha inizio appunto quando la funzione di thread "fun" rientra (con return), e si chiama sys_exit per terminare il task.

evoluzione sStack_T al termine dell'evento – numero di righe non significativo

USP
~~*a schedule da pick_next_task*~~
a sys_exit da schedule
~~*a wake_up_process da check_preempt_curr*~~
~~*a sys_exit da wake_up_process*~~
a System_Call da sys_exit
PSR U
a syscall da System_Call

Nota: la funzione check_preempt_curr non invoca la funzione resched, perché non c'è alcun task in attesa da risvegliare; naturalmente, la pila di sistema verrà in ultimo eliminata poiché il task T va a terminare.

seconda domanda – tabella delle pagine

Date le VMA di un processo P 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	3	R	P	M	X	0
K	0000 0060 0	1	R	P	M	X	3
S	0000 0060 1	4	W	P	M	X	4
D	0000 0060 5	2	W	P	A	-1	0
M0	0000 1000 0	2	W	S	M	G	2
M1	0000 3000 0	1	W	P	M	F	4
M2	0000 4000 0	1	W	P	A	-1	0
T1	7FFF F77F B	2	W	P	A	-1	0
T0	7FFF F77F E	2	W	P	A	-1	0
P	7FFF FFFF C	3	W	P	A	-1	0

1. Decomposizione degli indirizzi virtuali.

		PGD :	**PUD :**	**PMD :**	**PT**
C	0000 0040 0	***0***	***0***	2	***0***
K	0000 0060 0	***0***	***0***	3	***0***
S	0000 0060 1	***0***	***0***	***3***	***1***
D	0000 0060 5	***0***	***0***	***3***	***5***
M0	0000 1000 0	***0***	***0***	***128***	***0***
M1	0000 3000 0	***0***	***0***	384	***0***
M2	0000 4000 0	***0***	***1***	***0***	***0***
T1	7FFF F77F B	***255***	***511***	***443***	***507***
T0	7FFF F77F E	255	511	443	510
P	7FFF FFFF C	***255***	***511***	***511***	***508***

2. Numero di pagine necessarie:

pag PGD: *1*

pag PUD: *2*

pag PMD: *3*

pag PT: *7*

pag totali: *#PGD + #PUD + #PMD + #PT = 13*

3. Numero di pagine virtuali occupate dal processo: *somma di tutte le dimensioni di VMA = 21*

4. Rapporto di occupazione: *13 / 21 = 0,62 = 62 %*

5. Dimensione massima del processo in pagine virtuali:
   con la stessa dimensione di TP il processo può crescere fino a 7 x 512 = 3584 pagine virtuali
6. Rapporto di occupazione con dimensione massima: *13 / 3584 = 0,0036 (circa 3,6 millesimi)*