**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 venerdì 15 gennaio 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 **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 go, come
sem_t stay
int global = 0
```

```
void * walk (void * arg) {
   mutex_lock (&go)
   sem_post (&stay)
   global = 1                                   /* statement A */
   mutex_unlock (&go)
   global = 2
   mutex_lock (&come)
   sem_wait(&stay)
   mutex_unlock (&come)                         /* statement B */
   sem_wait (&stay)
   return NULL
} /* end walk */
```

```
void * run (void * arg) {
   mutex_lock (&go)
   sem_wait (&stay)
   global = (int) arg                           /* statement C */
   mutex_lock (&come)
   sem_post(&stay)
   mutex_unlock (&come)
   sem_post (&stay)
   mutex_unlock (&go)
   return NULL
} /* end run */
```

```
void main ( ) {
   pthread_t th_1, th_2
   sem_init (&stay, 0, 0)
   create (&th_1, NULL, walk, NULL)
   create (&th_2, NULL, run, void * 3)
   join (th_1, NULL)                            /* statement D */
   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 – *walk*** | **th_2 – *run*** |
| subito dopo stat. **A** | *ESISTE* | *PUÒ ESISTERE* |
| subito dopo stat. **B** | *ESISTE* | *PUÒ ESISTERE* |
| 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** | | | |
|---|---|---|---|---|
| | *go* | *come* | *stay* | *global* |
| subito dopo stat. **A** | *1* | *0* | *1* | *1* |
| subito dopo stat. **B** | *0 / 1* | *0* | *0 / 1* | *2 / 3* |
| subito dopo stat. **C** | *1* | *0 / 1* | *0* | *2 / 3* |
| subito dopo stat. **D** | *0 / 1* | *0* | *0* | *2 / 3* |

**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 – *walk*** | **th_2 – *run*** | *global* |
|---|---|---|---|
| **1** | `lock go` | `wait` | *0* |
| **2** | `prima wait` | `lock come` | *2 / 3* |
| **3** | `seconda wait` | `wait` | *2* |

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

## esercizio n. 2 – processi e nucleo

### prima parte – gestione dei processi

```
// programma ring_b.c
sem_t empty, full
float ring_buf [2], sum
int write_idx = 0, read_idx = 0, queue_size
pthread_mutex_t mux = PTHREAD_MUTEX_INITIALIZER
```

```
void * funz_1 (void * arg) {
  sem_wait (&empty)
  mutex_lock (&mux)
    ring_buf [write_idx] = 3.14f
    write_idx++
  mutex_unlock (&mux)
  sem_post (&full)
  sem_wait (&empty)
    ring_buf [write_idx] = 2.71f
  sem_post (&full)
  return NULL
}  // funz_1
```

```
void * funz_2 (void * arg) {
  sem_wait (&full)
  mutex_lock (&mux)
    sum = ring_buf [read_idx]
    read_idx++
  mutex_unlock (&mux)
  sem_post (&empty)
  sem_wait (&full)
    sum = sum + ring_buf [read_idx]
  sem_post (&empty)
  return NULL
}  // funz_2
```

```
void * funz_3 (void * arg) {
  char msg [50]
  nanosleep (5)
  mutex_lock (&mux)
    queue_size = write_idx - read_idx
  mutex_unlock (&mux)
  printf ("Queue size: %d", queue_size)
  write (stdout, msg, 50)
  return NULL
}  // funz_3
```

```
main ( ) {  // codice eseguito da P
  pthread_t th_1, th_2, th_3
  sem_init (&empty, 0, 2)
  sem_init (&full, 0, 0)
  create (&th_3, NULL, funz_3, NULL)
  create (&th_2, NULL, funz_2, NULL)
  create (&th_1, NULL, funz_1, NULL)
  join (th_3, NULL)
  join (th_2, NULL)
  join (th_1, NULL)
  exit (1)
}  // main
```

Un processo **P** esegue il programma **ring_b** e crea i thread **TH_1**, **TH_2** e **TH_3**. 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 non ha ancora creato nessun thread**. Si completi la tabella riportando quanto segue:

- I valori < *PID*, *TGID* > di ciascun processo che viene creato.
- I valori < *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** | **TH_3** | **TH_2** | *TH_1* |
|---|---|---|---|---|---|---|
| | **PID** | **1** | **2** | *3* | *4* | *5* |
| **evento oppure processo-chiamata** | **TGID** | **1** | **2** | *2* | *2* | *2* |
| **P – create TH_3** | 1 | **pronto** | **esec** | **pronto** | **NE** | **NE** |
| *P – create TH_2* | 2 | *pronto* | *esec* | *pronto* | *pronto* | *NE* |
| **interrupt da RT_clock e scadenza del quanto di tempo** | 3 | *pronto* | *pronto* | *esec* | *pronto* | *NE* |
| *TH_3 – nanosleep* | 4 | *pronto* | *pronto* | *attesa nanosleep* | *esec* | *NE* |
| *TH_2 – sem_wait (full)* | 5 | *pronto* | *esec* | *attesa nanosleep* | *attesa sem_wait* | *NE* |
| **P – create TH_1** | 6 | *pronto* | *esec* | *attesa nanosleep* | *attesa sem_wait* | *pronto* |
| *P – join TH_3* | 7 | *pronto* | *attesa join TH_3* | *attesa nanosleep* | *attesa sem_wait* | *esec* |
| *TH_1 – sem_wait (empty)* | 8 | *pronto* | *attesa join TH_3* | *attesa nanosleep* | *attesa sem_wait* | *esec* |
| *TH_1 – mutex_lock (mux)* | 9 | *pronto* | *attesa join TH_3* | *attesa nanosleep* | *attesa sem_wait* | *esec* |
| *interrupt da RT_clock e scadenza del timer* | 10 | **pronto** | **attesa** *join TH_3* | **esec** | **attesa** *sem_wait* | **pronto** |
| *TH_3 – mutex_lock (mux)* | 11 | *pronto* | *attesa join TH_3* | *attesa lock* | *attesa sem_wait* | *esec* |
| *TH_1 – mutex_unlock (mux)* | 12 | *pronto* | *attesa join TH_3* | *esec* | *attesa sem_wait* | *pronto* |
| *TH_3 – mutex_unlock (mux)* | 13 | *pronto* | *attesa join TH_3* | *esec* | *attesa sem_wait* | *pronto* |
| *TH_3 – write* | 14 | *pronto* | *attesa join TH_3* | *attesa write* | *attesa sem_wait* | *esec* |
| *TH_1 – sem_post (full)* | 15 | **pronto** | **attesa** *join TH_3* | **attesa** *write* | **esec** | **pronto** |

**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** | | |
| | 3 | 6 | 4 | t1 | 100 | | |
| **TASK** | **ID** | **LOAD** | **LC** | **Q** | **VRTC** | **SUM** | **VRT** |
| **CURRENT** | t1 | 1 | 0,25 | 1,5 | 1 | 10 | 100 |
| **RB** | t2 | 2 | 0,50 | 3 | 0,5 | 20 | 100,75 |
| | t3 | 1 | 0,25 | 1,5 | 1 | 30 | 101,25 |

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

**Events of task t2:** **CLONE at 1.0** **EXIT at 1.5**

**Events of task t3:** **WAIT at 1.0** **nella simulazione considerata wakeup non si verifica**

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

| **EVENTO 1** | **TIME** | **TYPE** | **CONTEXT** | **RESCHED** | | |
|---|---|---|---|---|---|---|
| | *1,5* | *Q scade* | *t1* | *vero* | | |
| **RUNQUEUE** | **NRT** | **PER** | **RQL** | **CURR** | **VMIN** | |
| | *3* | 6 | *4* | *t2* | *100,75* | |
| **TASK** | **ID** | **LOAD** | **LC** | **Q** | **VRTC** | **SUM** | **VRT** |
| **CURRENT** | *t2* | *2* | *0,50* | *3* | *0,5* | *20* | *100,75* |
| **RB** | *t3* | *1* | *0,25* | *1,5* | *1* | *30* | *101,25* |
| | *t1* | *1* | *0,25* | *1,5* | *1* | *11,5* | *101,50* |
| | | | | | | | |
| **WAITING** | | | | | | | |

| **EVENTO 2** | **TIME** | **TYPE** | **CONTEXT** | **RESCHED** | | |
|---|---|---|---|---|---|---|
| | *2,5* | *CLONE* | *t2* | *falso* | | |
| **RUNQUEUE** | **NRT** | **PER** | **RQL** | **CURR** | **VMIN** | |
| | *4* | 6 | *6* | *t2* | *101,25* | |
| **TASK** | **ID** | **LOAD** | **LC** | **Q** | **VRTC** | **SUM** | **VRT** |
| **CURRENT** | *t2* | *2* | *1/3=0,33* | *2* | *0,5* | *21* | *101,25* |
| **RB** | *t3* | *1* | *1/6=0,17* | *1* | *1* | *30* | *101,25* |
| | *t1* | *1* | *1/6=0,17* | *1* | *1* | *11,5* | *101,50* |
| | *t4* | *2* | *1/3=0,33* | *2* | *0,5* | *0* | *102,25* |
| **WAITING** | | | | | | | |

| EVENTO 3 | | TIME | TYPE | CONTEXT | RESCHED | | |
|---|---|---|---|---|---|---|---|
| | | *3* | *EXIT* | *t2* | *vero* | | |
| **RUNQUEUE** | **NRT** | **PER** | **RQL** | **CURR** | **VMIN** | | |
| | *3* | 6 | *4* | *t3* | *101,25* | | |
| **TASK** | **ID** | **LOAD** | **LC** | **Q** | **VRTC** | **SUM** | **VRT** |
| **CURRENT** | *t3* | *1* | *0,25* | *1,5* | *1* | *30* | *101,25* |
| **RB** | *t1* | *1* | *0,25* | *1,5* | *1* | *11,5* | *101,50* |
| | *t4* | *2* | *0,5* | *3* | *0,5* | *0* | *102,25* |
| | | | | | | | |
| **WAITING** | | | | | | | |

| EVENTO 4 | | TIME | TYPE | CONTEXT | RESCHED | | |
|---|---|---|---|---|---|---|---|
| | | *4* | *WAIT* | *t3* | *vero* | | |
| **RUNQUEUE** | **NRT** | **PER** | **RQL** | **CURR** | **VMIN** | | |
| | *2* | 6 | *3* | *t1* | *101,50* | | |
| **TASK** | **ID** | **LOAD** | **LC** | **Q** | **VRTC** | **SUM** | **VRT** |
| **CURRENT** | *t1* | *1* | *1/3=0,33* | *2* | *1* | *11,5* | *101,50* |
| **RB** | *t4* | *2* | *2/3=0,67* | *4* | *0,5* | *0* | *102,25* |
| | | | | | | | |
| | | | | | | | |
| **WAITING** | *t3* | *1* | | | | *31* | *102,25* |

Calcolo del *VRT iniziale* del **task t4** creato dalla **CLONE** eseguita dal **task t2**:

$$t4.VRT\ (iniziale) = VMIN + t4.Q \times t4.VRTC = 101{,}25 + 2 \times 0{,}5 = 102{,}25$$

Valutazione della *condizione di rescheduling* alla **CLONE** eseguita dal **task t2**:

$$t4.VRT + WGR \times t4.LC < t2.VRT \Rightarrow 102{,}25 + 1 \times 0{,}33 = 102{,}58 < 101{,}25 \Rightarrow falso$$

**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** (esiste un processo P)

```
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  7FFFFFFFB,  4,  W,  P,  A,  <-1, 0>

    PT: <c0 :-  ->  <c1 :1  R>  <k0 :- ->  <s0 :- ->  <p0 :3  D W>
        <p1 :2  W>  <p2 :7  W>  <p3 :- ->

    process P - NPV of PC and SP:  c1, p2

____MEMORIA FISICA____(pagine libere: 3)_____________________________
      00 : <ZP>                 ||  01 : Pc1 / <XX, 1>        ||
      02 : Pp1                  ||  03 : Pp0  D               ||
      04 : ----                 ||  05 : <G, 1>               ||
      06 : <G, 2>               ||  07 : Pp2                  ||
      08 : ----                 ||  09 : ----                 ||

____STATO del TLB_____________________________________________________
      Pc1 : 01 -  0: 1:         ||  Pp0 : 03 -  1: 0:         ||
      Pp1 : 02 -  1: 1:         ||  Pp2 : 07 -  1: 1:         ||
            -----               ||        -----               ||
            -----               ||        -----               ||

SWAP FILE:    ----, ----, ----, ----, ----, ----,
LRU ACTIVE:   PP2,  PP1,  PC1,
LRU INACTIVE: pp0,
```

## evento 1: *read* (Pc1) – *write* (Pp3, Pp4) – 4 *kswapd*

*Prima iterazione dell'evento: la pagina Pp3 da scrivere è di growsdown (non ancora allocata in memoria fisica), pertanto si modifica la VMA di P (aggiungendo la pagina virtuale Pp4 come growsdown), poi si ha COW, si alloca Pp3 in pagina fisica 04, e la pagina Pp3 viene inserita in lista Active; la pagina Pp4 da scrivere è di growsdown (appena aggiunta e non ancora allocata in memoria fisica), pertanto si modifica la VMA di P (aggiungendo la pagina virtuale Pp5 come growsdown), poi si ha COW con free =2, dunque si attiva PFRA che libera le pagine fisiche 05 e 06 di page cache, si alloca Pp4 in pagina fisica 05; e la pagina Pp4 viene inserita in lista Active; poi agisce kswapd e aggiorna le liste. Successivamente si hanno le altre tre iterazioni dell'evento, le quali aggiornano progressivamente le liste – con le pagine Pc1, Pp3 e Pp4 che risultano sempre accedute – a fino a raggiungere stabilità.*

| **PT** del processo: **P** | | | | |
|---|---|---|---|---|
| p0: 3 D W | p1: 2 W | p2: 7 W | p3: *4 W* | p4: *5 W* |
| p5: – – | | | | |

| **process P** | NPV of **PC**: *c1* | NPV of **SP**: *p4* |
|---|---|---|

| **MEMORIA FISICA** | |
|---|---|
| 00: <ZP> | 01: *Pc1 / <X, 1>* |
| 02: *Pp1* | 03: *Pp0 D* |
| 04: *Pp3* | 05: *Pp4* |
| 06: | 07: *Pp2* |
| 08: | 09: |

**LRU ACTIVE**: *PP4, PP3, PC1* **LRU-INACTIVE**: *pp2, pp1, pp0*

## evento 2: *fork* (R)

*Per COW la pagina condivisa Pp4 / Rp4 (pagina di cima pila) viene sdoppiata: la pagina Pp4 viene allocata nella nuova pagina fisica 06, mentre per Rp4 si mantiene la vecchia pagina fisica 05 marcandola dirty. Tutte le altre pagine diventano condivise tra P e R (e potenzialmente separabili tramite COW, se l'area cui appartengono è privata e scrivibile).*

| **PT** del processo: **R** | | | | |
|---|---|---|---|---|
| p0: *3 D R* | p1: *2 R* | p2: *7 R* | p3: *4 R* | p4: *5 D W* |
| p5: *- -* | | | | |

| **process R** | NPV of **PC**: *c1* | NPV of **SP**: *p4* |
|---|---|---|

| **MEMORIA FISICA** | |
|---|---|
| 00: <ZP> | 01: *Pc1 / Rc1 / <X, 1>* |
| 02: *Pp1 / Rp1* | 03: *Pp0 / Rp0 D* |
| 04: *Pp3 / Rp3* | 05: *Rp4 D* |
| 06: *Pp4* | 07: *Pp2 / Rp2* |
| 08: | 09: |

**LRU ACTIVE**: *RP4, RP3, RC1, PP4, PP3, PC1*

**LRU INACTIVE**: *rp2, rp1, rp0, pp2, pp1, pp0*

## evento 3: *clone* (S, c0)

*Viene creata la VMA T0 (area di pila) per il thread S, tutte le pagine di P vengono condivise (inseparabilmente) con S, e va allocata la pagina t00 (cima pila di S). Poiché vale free = 2, si attiva PFRA che libera la pagina fisica 03 deallocando la pagina condivisa Pp0 / Rp0 e scaricandola in swap file (in quanto anonima) poiché è dirty (marcata D nel descrittore di pagina fisica), e libera la pagina fisica 02 deallocando la pagina condivisa Pp1 / Rp1 e scaricandola in swap file (in quanto anonima) poiché è dirty (bit D = 1 nel TLB iniziale). La nuova pagina condivisa PSt00 viene allocata in pagina fisica 02 e si aggiornano le liste LRU, togliendo (da Inactive) le pagine deallocate e inserendo la nuova pagina PSt00 (in Active, qui non richiesta).*

| **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** | | | | |
|---|---|---|---|---|
| p0: *s0 R* | p1: *s1 R* | p2: *7 R* | p3: *4 R* | p4: *6 W* |
| p5: *- -* | t00: *2 W* | t01: *- -* | | |

| **process P** | NPV of **PC**: *c1* | NPV of **SP**: *p4* |
|---|---|---|
| **process S** | NPV of **PC**: *c0* | NPV of **SP**: *t00* |

| **MEMORIA FISICA** | |
|---|---|
| 00: <ZP> | 01: *PSc1 / Rc1 / <X, 1>* |
| 02: *PSt00* | 03: |
| 04: *PSp3 / Rp3* | 05: *Rp4 D* |
| 06: *PSp4* | 07: *PSp2 / Rp2* |
| 08: | 09: |

| **SWAP FILE** | |
|---|---|
| s0: *PSp0 / Rp0* | s1: *PSp1 / Rp1* |
| s2: | s3: |

**LRU INACTIVE**: *rp2, pp2,* _____

**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 / <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:        ||        -----                ||
             -----              ||        -----                ||
```

| nome file | f_pos | f_count | numero pag. lette | numero pag. scritte |
|---|---|---|---|---|
| **F** | **6000** | **1** | **2** | **0** |

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.
Il processo **P** è in esecuzione. Il file **F** è stato aperto da **P** tramite chiamata **fd = *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 che la primitiva *close* scrive le pagine dirty di un file solo se *f_count* diventa = 0.

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

*Fork crea il processo figlio Q e deve allocare la pagina di (cima) pila Qp0, separandola da Pp0 e marcandola D (dirty). Dato che vale free = 1, viene invocato PFRA che libera le pagine fisiche 03 e 05, poiché entrambe sono di page cache, e poi la pagina fisica 03 viene allocata a Pp0, mentre la pagina fisica 02 resta allocata a Qp0. La commutazione di contesto svuota (flush) il TLB, pertanto le pagine Pp0 e Pm00 di P (la seconda è condivisa con Q) vengono marcate D (dirty) nei rispettivi descrittori di pagina fisica, poiché nel TLB esse figurano dirty al momento del flush.*

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

| nome file | f_pos | f_count | numero pag. lette | numero pag. scritte |
|---|---|---|---|---|
| **F** | *6000* | *2* | *2* | *1* |

## evento 3: *read* (fd, 7000)

*Il file F viene letto da posizione 6000 a 13000, ossia nelle pagine 1, 2 e 3. La pagina <F, 1> è già allocata in pagina fisica 06 e viene acceduta, mentre la pagina <F, 2> viene caricata da disco in pagina fisica 05 e viene acceduta. Per la pagina <F, 3> vale free = 1, dunque viene invocato PFRA che libera la pagina fisica 05 (ossia <F, 2>) senza scaricarla su disco poiché non è dirty, e libera la pagina fisica 06 (ossia <F, 1>) scaricandola su disco poiché è dirty; entrambe sono di page cache. Poi la pagina <F, 3> viene caricata da disco in pagina fisica 05 e viene acceduta. Operazioni disco: due pagine lette e una scritta.*

| MEMORIA FISICA | |
|---|---|
| 00: <ZP> | 01: Pc2 / Qc2 / <X, 2> |
| 02: *Qp0 D* | 03: *Pp0 D* |
| 04: *Pm00 / Qm00 D* | 05: *<F, 3>* |
| 06: *----* | 07: *----* |

| nome file | f_pos | f_count | numero pag. lette | numero pag. scritte |
|---|---|---|---|---|
| **F** | *13000* | *2* | *4* | *2* |

## evento 4: *lseek* (fd, –8000) // offset negativo !

| nome file | f_pos | f_count | numero pag. lette | numero pag. scritte |
|---|---|---|---|---|
| **F** | *5000* | *2* | *4* | *2* |

## evento 5: *write* (fd, 1000)

*Il file F viene letto da posizione 5000 a 6000, ossia nella pagina 1. La pagina <F, 1> viene caricata da disco in pagina fisica 06 e acceduta (scritta e marcata D). Operazioni disco: una pagina letta.*

| MEMORIA FISICA | |
|---|---|
| 00: <ZP> | 01: Pc2 / Qc2 / <X, 2> |
| 02: Qp0 D | 03: Pp0 D |
| 04: *Pm00 / Qm00 D* | 05: *<F, 3>* |
| 06: *<F, 1> D* | 07: *----* |

| nome file | f_pos | f_count | numero pag. lette | numero pag. scritte |
|---|---|---|---|---|
| **F** | *6000* | *2* | *5* | *2* |

## evento 6: fd1 = *open* (H), *write* (fd1, 9000)

*Il file H viene aperto e letto da posizione 0 a 9000, ossia nelle pagine 0, 1 e 2. Viene invocato PFRA che libera le pagine fisiche 05 e 06, poiché entrambe sono di page cache (la seconda è dirty e viene scaricata su disco), poi le pagine <H, 0> e <H, 1> vengono caricate da disco in pagina fisica 05 e in pagina fisica 06, rispettivamente, e accedute (scritte e marcate D). Per la pagina <H, 2> viene invocato nuovamente PFRA, che libera nuovamente le pagine fisiche 05 e 06 (entrambe sono dirty e vengono scaricate su disco), poi la pagina <H, 2> viene caricata da disco in pagina fisica 05 e acceduta (scritta e marcata D). Operazioni disco: file F una pagina scritta, file H tre pagine lette e due scritte.*

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

| nome file | f_pos | f_count | numero pag. lette | numero pag. scritte |
|---|---|---|---|---|
| **F** | *6000* | *2* | *5* | *3* |
| **H** | *9000* | *1* | *3* | *2* |

## esercizio n. 4 – domande varie (due)

### prima domanda – moduli del SO

**stato iniziale: CURR = P, Q = ATTESA (E) di lettura da disco**

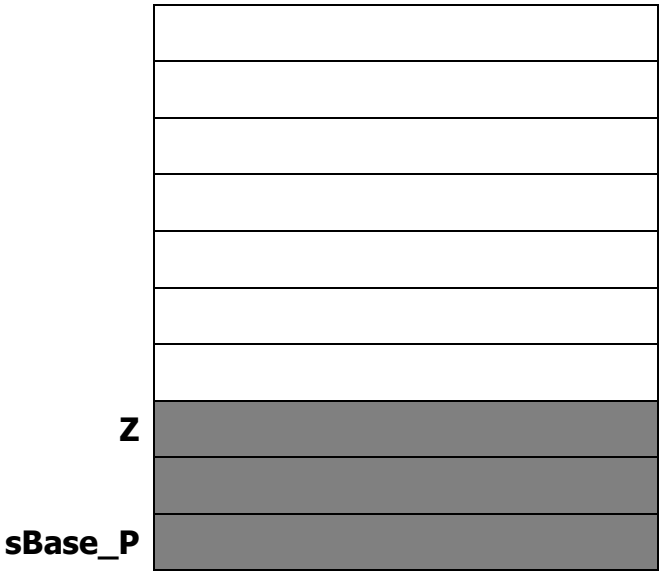
Si consideri il seguente evento: il processo **P** è in esecuzione in **modo U** e si verifica un **interrupt da DMA** di completamento di un'operazione di **lettura da disco**. Si assuma che il processo **Q**, al suo risveglio, abbia acquisito **diritti maggiori** di esecuzione rispetto a **P**.

**domanda**

- mostrare le **invocazioni** di tutti i moduli (ed eventuali relativi ritorni) eseguiti nel contesto del processo **P** per gestire l'evento indicato
- mostrare (in modo simbolico) il contenuto dello **stack utente** e dello **stack di sistema** del processo **P** al termine della gestione dell'evento considerato

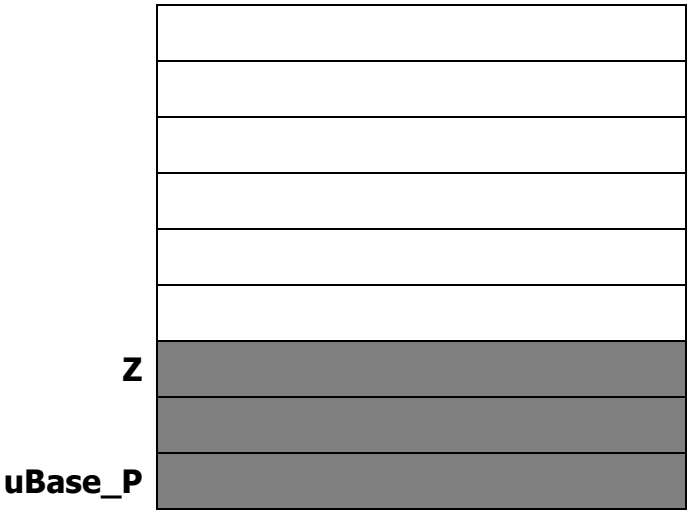
### invocazione moduli

| processo | modo | modulo |
|---|---|---|
| P | U – S | > R_int (E) |
| *P* | *S* | > *wakeup* |
| *P* | *S* | > *check_preempt_curr* |
| *P* | *S* | > *resched (set TNR)* < |
| *P* | *S* | *check_preempt_curr* < |
| *P* | *S* | *wakeup* < |
| *P* | *S* | > *schedule* |
| *P* | *S* | > *pick_next_task* < |
| P – Q | *S* | *context_switch* |
| | | |

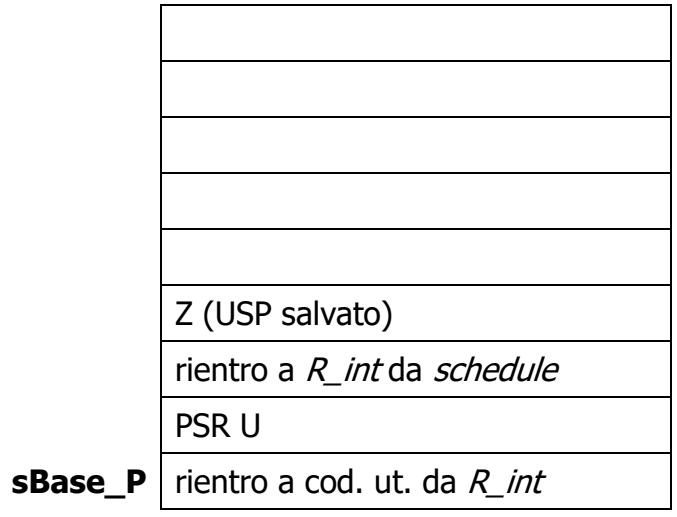



**uStack_P – iniziale**

### contenuto stack al termine dell'evento



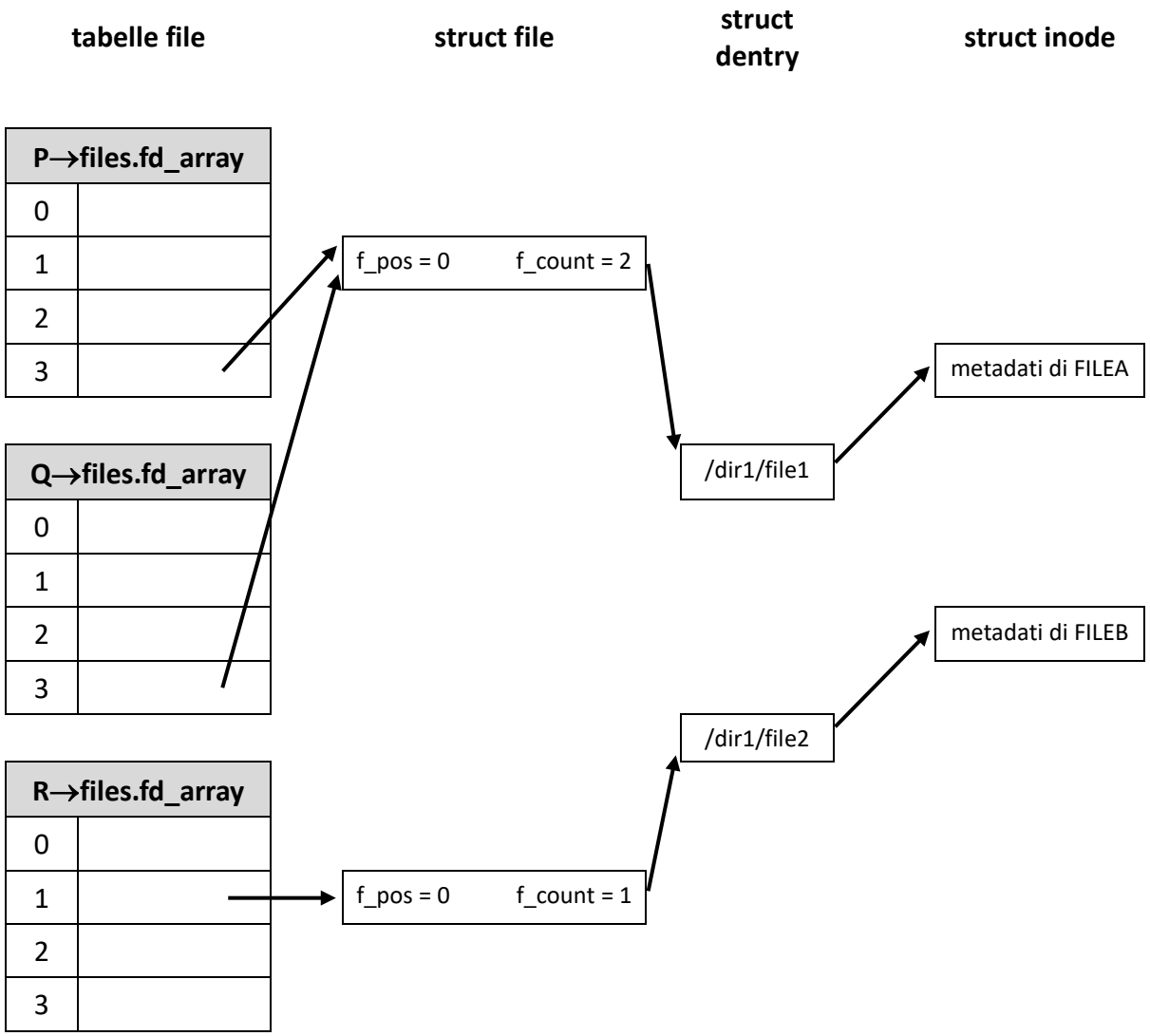

**uStack_P – finale**




**sStack_P – finale**

**seconda domanda – struttura del file system**

La figura sottostante è una rappresentazione dello stato del VFS raggiunto dopo l'esecuzione in sequenza di un certo numero di chiamate di sistema sotto riportate.



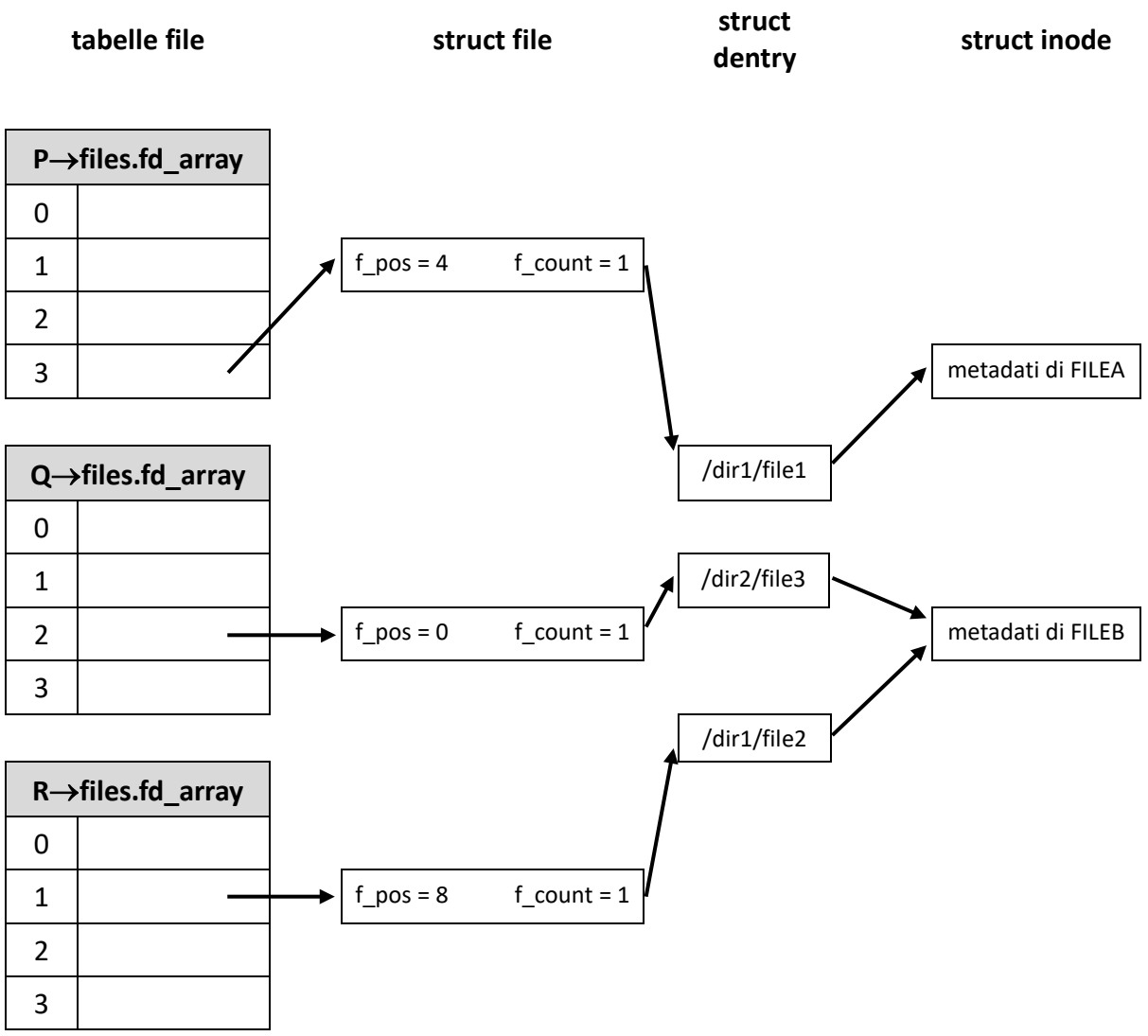

**chiamate di sistema eseguite nell'ordine indicato**

1) **P** fd = *open* ("/dir1/file1", ...)
2) **P** pid = *fork* ( ) // il processo padre P crea il processo figlio Q
3) un altro processo (qui non considerato) crea il processo **R**
4) **R** *close* (1)
5) **R** fd = *open* ("/dir1/file2", …)
6) **R** *link* ("/dir1/file2", "/dir2/file3")

Ora si supponga di partire dallo stato del VFS mostrato nella figura iniziale e si risponda alla **domanda** alla pagina seguente, riportando la **sequenza di chiamate di sistema** che può avere generato la nuova situazione di VFS mostrata nella figura successiva. Valgono questi vincoli:

- i soli tipi di **chiamata** da considerare sono: **`open, close, read`**
- lo **scheduler** mette in esecuzione i processi in questo ordine: **P**, **R**, **Q**

**domanda**




**sequenza di chiamate di sistema** (numero di righe non significativo)

| # | processo | chiamata di sistema |
|---|---|---|
| 1 | *P* | *read (fd, 4)* |
| 2 | *R* | *read (fd, 8)* |
| 3 | *Q* | *close (fd)* |
| 4 | *Q* | *close (2)* |
| 5 | *Q* | *fd = open ("/dir2/file3", ...)* |
| 6 | | |
| 7 | | |
| 8 | | |

*Nota: le due close alle righe 3 e 4 sono commutabili.*