**Politecnico di Milano**
**Dipartimento di Elettronica, Informazione e Bioingegneria**

**prof.ssa Anna Antola** — **prof. Giuseppe Pelagatti**
**prof. Luca Breveglieri** — **prof.ssa Donatella Sciuto**
**prof. Roberto Negrini** — **prof.ssa Cristina Silvano**

# AXO – Architettura dei Calcolatori e Sistemi Operativi

## SECONDA PARTE di martedì 12 febbraio 2019

**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 (5.5 punti)** _____

**esercizio 4 (1.5 punti)** _____

**voto finale: (16 punti)** _____

## 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 root
sem_t stem, leaf
int global = 0
```

```
void * seed (void * arg) {
   sem_wait (&stem)
   mutex_lock (&root)
   sem_wait (&leaf)
   mutex_unlock (&root)                          /* statement A */
   global = 1
   sem_post (&stem)                              /* statement B */
   return (void * 2)
} /* end seed */
```

```
void * fruit (void * arg) {
   mutex_lock (&root)
   sem_post (&leaf)
   global = 3                                    /* statement C */
   sem_wait (&stem)
   mutex_unlock (&root)
   sem_post (&stem)
   return NULL
} /* end fruit */
```

```
void main ( ) {
   pthread_t th_1, th_2
   sem_init (&stem, 0, 1)
   sem_init (&leaf, 0, 0)
   create (&th_1, NULL, seed, NULL)
   create (&th_2, NULL, fruit, NULL)
   join (th_1, &global)                          /* 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 – seed** | **th_2 – `fruit`** |
| 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 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** | | |
|---|---|---|---|
| | *stem* | *leaf* | *global* |
| subito dopo stat. **A** | 0 | 0 | 3 |
| subito dopo stat. **B** | 1 | 0 | 1 |
| subito dopo stat. **C** | 1 - 0 | 1 | 3 |
| subito dopo stat. **D** | 1 | 0 | 2 |

**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 – seed** | **th_2 – `fruit`** | *global* |
|---|---|---|---|
| **1** | sem_wait(&leaf) | mutex_lock(&root) | 0 |
| **2** | mutex_lock(&root) | sem_wait(&stem) | 3 |

## esercizio n. 2 – processi e nucleo

### prima parte – gestione dei processi

```
// programma prova.c
main ( ) {
   pid1 = fork ( )
   if (pid1 == 0) {          // codice eseguito dal figlio Q
      read (stdin, msg, 5)
      execl ("/acso/NEW_CODE", "NEW_CODE", NULL)
      write (stdout, error_msg, 50)
   } else {
      write (stdout, msg, 25)
      pid1 = wait (&status)
   } /* if */
   exit (0)
}  /* prova */
```

```
// programma NEW_CODE.c
sem_t pass
int glob = 2
```

```
void * BEGIN (void * arg) {
 if (glob == 2) {
   pthread_mutex_lock (&lock)
   sem_post (&pass)
   pthread_mutex_unlock (&lock)
   sem_post (&pass)
  } /* if */
  return NULL
 }  /* BEGIN */
```

```
void * END (void * arg) {

  sem_wait (&pass)
  glob = 3
  pthread_mutex_lock (&lock)
  sem_wait (&pass)
  pthread_mutex_unlock (&lock)
  sem_wait (&pass)
  return NULL
  }  /* END */
```

```
main ( ) {  // codice eseguito da Q
    pthread_t TH_1, TH_2
    sem_init (&pass, 0, 0)
    pthread_create (&TH_2, NULL, END, NULL)
    sem_post (&pass)
    pthread_create (&TH_1, NULL, BEGIN, NULL)
    if (glob == 2) {
       pthread_join (TH_2, NULL)
       pthread_join (TH_1, NULL)
    } else
       exit (-1)
}  /* main */
```

Un processo ***P*** esegue il programma ***prova*** e crea un figlio ***Q*** che esegue una mutazione di codice (programma ***NEW_CODE***). La mutazione di codice va a buon fine e sono 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. Si completi la tabella riportando quanto segue:

- ⟨ *PID*, *TGID* ⟩ di ciascun processo 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**

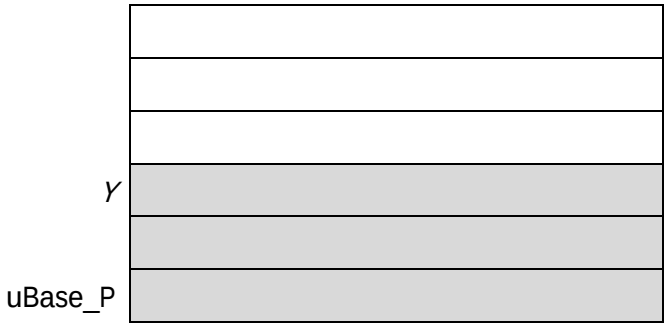
## TABELLA DA COMPILARE

| *identificativo simbolico del processo* | | ***IDLE*** | ***P*** | ***Q*** | TH2 | TH1 |
|---|---|---|---|---|---|---|
| *evento* oppure *processo-chiamata* | *PID* | 1 | 2 | 3 | 4 | 5 |
| | *TGID* | 1 | 2 | 3 | 3 | 3 |
| P –pid1=fork | 0 | pronto | esec | pronto | NE | NE |
| *interrupt* da RT_clock e scadenza quanto di tempo | 1 | pronto | pronto | ESEC | NE | NE |
| Q - read | 2 | pronto | ESEC | A read | NE | NE |
| P - write | 3 | ESEC | A write | A read | NE | NE |
| *interrupt* da *stdout*, tutti i caratteri trasferiti | 4 | pronto | ESEC | A read | NE | NE |
| *interrupt* da RT_clock e scadenza quanto di tempo | 5 | pronto | ESEC | A read | NE | NE |
| Interrupt da DMA_IN, tutti i blocchi letti | 6 | pronto | pronto | esec | NE | NE |
| Q - execl | 7 | pronto | pronto | ESEC | NE | NE |
| Q - pthread_create(TH2) | 8 | pronto | pronto | ESEC | pronto | NE |
| *interrupt* da RT_clock e scadenza quanto di tempo | 9 | pronto | ESEC | pronto | pronto | NE |
| P - wait(&status) | 10 | pronto | A wait | pronto | ESEC | NE |
| TH2 - sem_wait(&pass) | 11 | pronto | A wait | ESEC | A sem | NE |
| Q - sem_post(&pass) | 12 | pronto | A wait | pronto | ESEC | NE |
| Interrupt da RT_clock e s cadenza del quanto di tempo | 13 | pronto | A | esec | pronto | NE |
| Q - pthread_create(TH1) | 14 | pronto | A wait | ESEC | pronto | pronto |
| Q - exit | 15 | esec | A | NE | NE | NE |

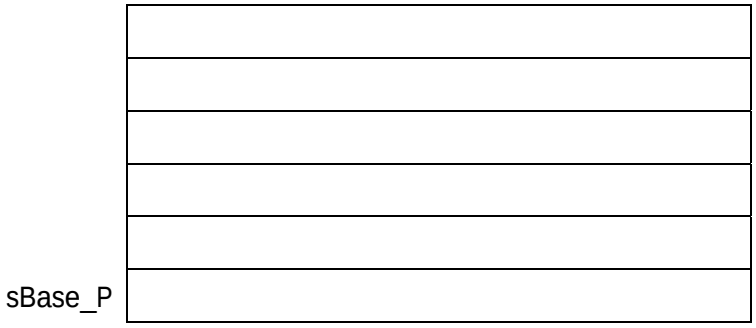
Poichè TH1 e TH2 sono thread figli di Q, quando questo termina anche loro vengono eliminati

**seconda parte – moduli del SO**

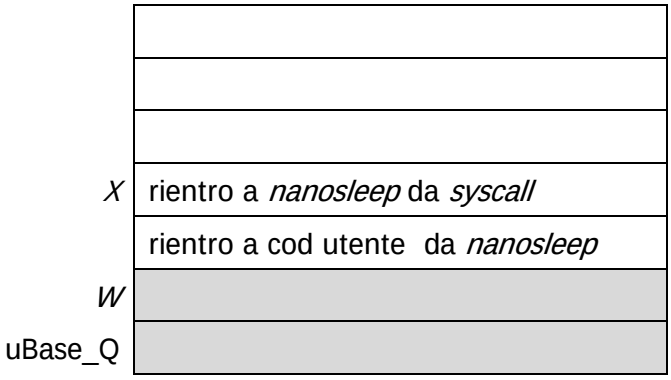
Sono dati due processi ***P*** e ***Q***, dove P è il processo padre di Q. Non ci sono altri processi utente nel sistema. Lo stato iniziale delle pile di sistema e utente dei due processi è riportato qui sotto.



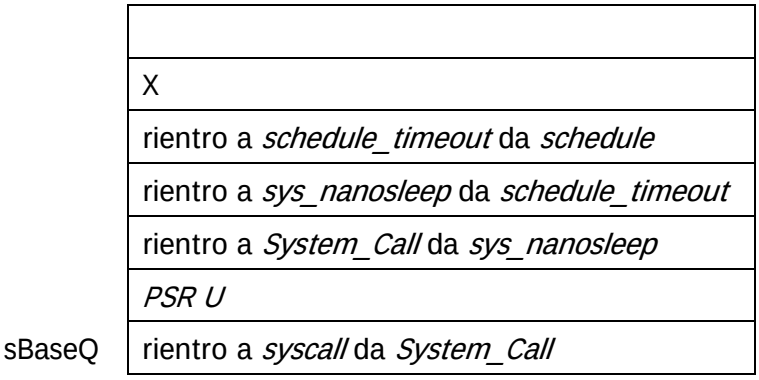

uStack_P – iniziale




sStack_P – iniziale




uStack_Q – iniziale




sStack_Q – iniziale

Si considerino due eventi, **Evento 1** e **Evento 2**, che si verificano in successione.

**Evento 1:** Interrupt da Real Time Clock con *time_out* scaduto. Il risveglio del processo in attesa di *time_out* non comporta l'attivazione di *resched ( )*.

**Evento 2:** il processo correntemente in esecuzione invoca **exit ( )**

Per gli eventi indicati si compili la **tabella di invocazione dei moduli** della pagina successiva, riempiendola in successione con le invocazioni relative a **Evento 1** seguite da quelle relative a **Evento2**.

Per la compilazione si segua lo schema usuale, **mostrando** le invocazioni di tutti i **moduli** (e eventuali relativi ritorni) e precisando processo e modo. La simulazione delle invocazioni dei moduli deve arrivare fino al *context switch* del processo che esegue la *exit*, ma non oltre.

NOTAZIONE da usare per i moduli: > (invocazione), nome_modulo (esecuzione), < (ritorno)

P è in esecuzione

Q si è sospeso per attendere la scadenza di un time-out

**Evento 1 ed Evento 2**

**invocazione moduli** (num. di righe vuote non signif.)

| Tabella di invocazione dei moduli | | |
|---|---|---|
| **processo** | **modo** | **modulo** |
| P | U | "codice utente di P" |
| P | U -> S | >R_int_clock |
| P | S | >task_tick < |
| P | S | >Controlla_timer |
| P | S | >wakeup_process |
| P | S | >enqueue_task < |
| P | S | >check_preempt_curr < |
| P | S | wakeup_process < |
| P | S | Controlla_timer < |
| P | S -> U | R_int_clock < |
| P | U | "codice utente di P" |
| P | U | >exit |
| P | U | >syscall |
| P | U -> S | SYSCALL ( >system_call) |
| P | S | >sys_exit_group |
| P | S | >sys_exit |
| P | S | >wakeup_process < (Non risveglia nessuno) |
| P | S | >schedule |
| P | S | >dequeue < |
| P | S | >pick_next_task < |
| P -> Q | S | schedule ("context_switch") |

## esercizio n. 3 – memoria e file system

### prima parte – 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,  2 , R  ,  P  ,  M  , <X,0>
             S  000000600,  2 , W  ,  P  ,  M  , <X,2>
             P  7FFFFFFFB,  4 , W  ,  P  ,  A  , <-1,0>
    PT: <c0 :1  R>  <c1 :- ->  <s0 :3 R>  <s1 :- ->  <p0 :2 R>
        <p1 :5  W>  <p2 :6 W>  <p3 :- ->
    process P - NPV of PC and SP: c0, p1
PROCESSO: Q ********************* (non interessa) *********************

____MEMORIA FISICA____(pagine libere: 2)_____________________
      00 : <ZP>             ||  01 : Pc0 / Qc0 / <X,0>   ||
      02 : Pp0 / Qp0        ||  03 : Ps0 / Qs0 / <X,2>   ||
      04 : ----             ||  05 : Pp1                 ||
      06 : Pp2              ||  07 : ----                ||

____STATO del TLB_____________________________________________
      Pc0 : 01 -  0: 1:     ||   Pp0 : 02 -  1: 0:       ||
      Ps0 : 03 -  0: 0:     ||   Pp1 : 05 -  1: 0:       ||
      Pp2 : 06 -  1: 0:     ||         -----             ||

SWAP FILE:      Qp1  , ----, ----, ----, ----, ----, ----, ----,

LRU ACTIVE:     PC0,
LRU INACTIVE:   pp1, pp2, pp0, ps0, qs0, qp0, qc0,
```

## evento 1: write (Ps0)

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

| SWAP FILE | |
|---|---|
| s0: | s1: |
| s2: | s3: |

## evento 2: read (Ps1)

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

| SWAP FILE | |
|---|---|
| s0: | s1: |
| s2: | s3: |

## evento 3: clone (R, c1)

| **VMA** del processo **P/R**<br>(compilare solo la riga relativa alla nuova VMA creata) | | | | | | | |
|---|---|---|---|---|---|---|---|
| AREA | NPV iniziale | dimensione | R/W | P/S | M/A | nome file | offset |
| | | | | | | | |

## process R - NPV of PC and SP: _____

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

| SWAP FILE | |
|---|---|
| s0: | s1: |
| s2: | s3: |

## evento 4: context switch (R)

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

| SWAP FILE | |
|---|---|
| s0: | s1: |
| s2: | s3: |

**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: 5)_____________________________
      00 : <ZP>                  ||  01 : Pc0 / <X,0>          ||
      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.

## Evento 1: fd1 = open (F)

| | f_pos | f_count | numero pagine lette | numero pagine scritte |
|---|---|---|---|---|
| **file F** | 0 | 1 | 0 | 0 |

## Eventi 2 e 3: fork (Q), fd2 = open (G)

| MEMORIA FISICA | |
|---|---|
| 00: <ZP> | 01: Pc0 / Qc0 / <X, 0> |
| 02: Qp0 (D) | 03: Pp0 |
| 04: | 05: |
| 06: | 07: |

| | f_pos | f_count | numero pagine lette | numero pagine scritte |
|---|---|---|---|---|
| **file F** | 0 | 2 | 0 | 0 |
| **file G** | 0 | 1 | 0 | 0 |

0 ---- 4096 ---- 8192 ---- 12288 ---- 16384 ---- 20480 ---- 24576
0 1 2 3 4 5

## Evento 4: write (fd1, 9000)

| MEMORIA FISICA | |
|---|---|
| 00: <ZP> | 01: Pc0 / Qc0 / <X, 0> |
| 02: Qp0 (D) | 03: Pp0 |
| 04: <F, 0> (D) | 05: <F, 1> (D) |
| 06: <F, 2> (D) | 07: |

| | f_pos | f_count | numero pagine lette | numero pagine scritte |
|---|---|---|---|---|
| file F | 9000 | 2 | 3 | 0 |

## Evento 5: write (fd2, 3000)

| MEMORIA FISICA | |
|---|---|
| 00: <ZP> | 01: Pc0 / Qc0 / <X, 0> |
| 02: Qp0 (D) | 03: Pp0 |
| 04: <G, 0> (D) | 05: - - - - |
| 06: <F, 2> (D) | 07: |

| | f_pos | f_count | numero pagine lette | numero pagine scritte |
|---|---|---|---|---|
| file F | 9000 | 2 | 3 | 2 |
| file G | 3000 | 1 | 1 | 0 |

## Eventi 6 e 7: close (fd1), close (fd2)

| | f_pos | f_count | numero pagine lette | numero pagine scritte |
|---|---|---|---|---|
| file F | 9000 | 1 | 3 | 3 |
| file G | - - - - | 0 | 1 | 1 |

## esercizio n. 4 – virtual file system (VFS)

Si riportano nel seguito gli elementi di interesse di alcune `struct` necessarie alla gestione, organizzazione e accesso di file e cataloghi.

| | |
|---|---|
| `struct task_struct { ……`<br>`  struct files_struct *files ……}` | Ogni istanza rappresenta un descrittore di processo. |

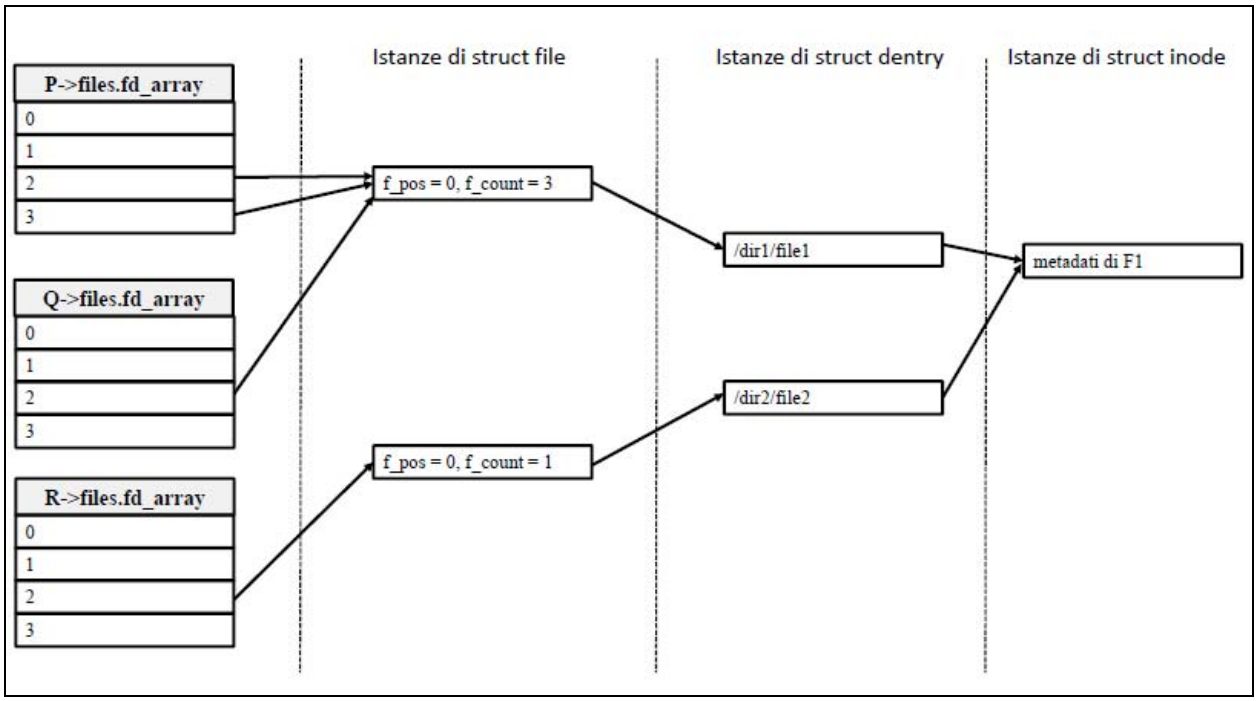
| | |
|---|---|
| `struct files_struct { ……`<br>`  struct file *fd_array [NR_OPEN_DEFAULT] ……}` | `fd_array[]` costituisce la tabella dei (descrittori) dei file aperti da un processo. |

| | |
|---|---|
| `struct file { ……`<br>`  struct dentry *f_dentry`<br>`  off_t f_pos  //posizione corrente`<br>`  f_count  //contatore riferim. al file aperto ……}` | Ogni istanza rappresenta un file aperto nel sistema. |

| | |
|---|---|
| `struct dentry { ……`<br>`  struct inode *d_inode ……}` | Ogni istanza rappresenta un nodo (file o catalogo) nell'albero dei direttori del VFS. |

| | |
|---|---|
| `struct inode {`<br>`  ……  }` | Ogni istanza rappresenta uno e un solo file fisicamente esistente nel volume. |

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



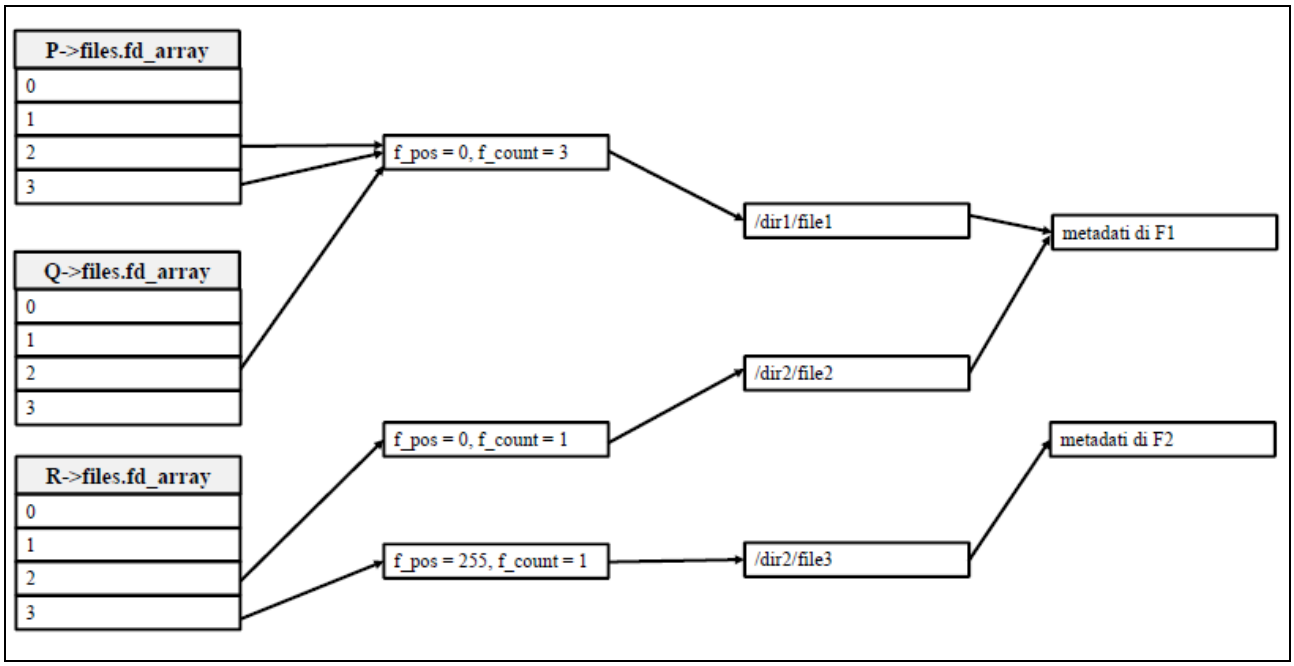

### Chiamate di sistema eseguite nell'ordine indicato

- P: close (2)
- P: fd = open (/dir1/file1, ...)
- P: pid = fork ( )  // il processo Q è stato creato da P
- P: fd1 = dup (fd)
- Il processo R è stato creato da altro processo non di interesse nell'esercizio
- R: link (/dir1/file1, /dir2/file2)
- R: close (2)
- R: fd = open (/dir2/file2)

Si supponga ora di partire dallo stato del VFS mostrato nella figura iniziale. Per ciascuno degli stati successivi, si risponda alle **domande** riportando la chiamata o la sottosequenza di chiamate che può avere generato la creazione di istanze di `struct` del VFS presentate nelle figure.

**Le sole tipologie di chiamate da considerare sono: `open()`, `close()`, `read()`, `dup()`, `link()`**

### Domanda 1

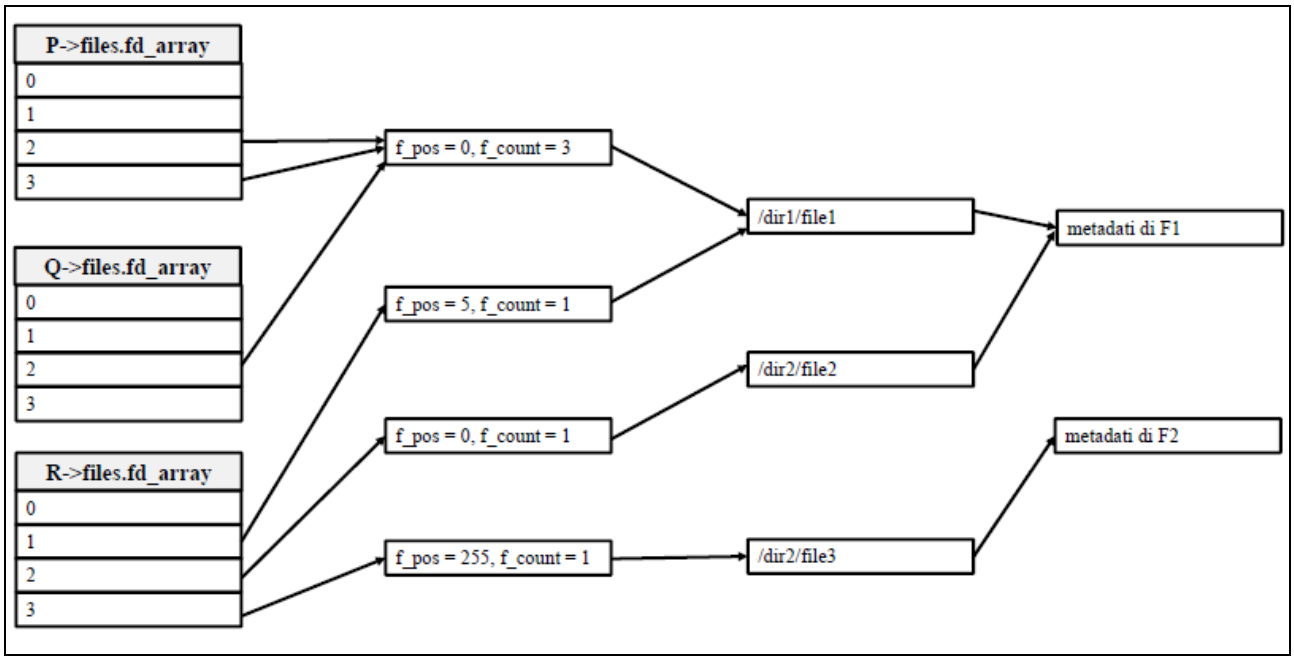



### Chiamata/e di sistema

R: fd1 = open("/2/file3")

R: read(fd1, 255)

### Domanda 2




### Chiamata/e di sistema

R: close(1)

R: fd2 = open("dir1/file1")

R: read(fd2, 5)

**spazio libero per brutta copia o continuazione**