



Politecnico di Milano

Dipartimento di Elettronica, Informazione e Bioingegneria

prof.ssa Anna Antola
prof. Luca Breveglieri
prof. Roberto Negrini

prof. Giuseppe Pelagatti
prof.ssa Donatella Sciuto
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) _____

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 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	<i>thread</i>	
	th_1 – seed	th_2 – fruit
subito dopo stat. A	<i>ESISTE</i>	<i>PUÒ ESISTERE</i>
subito dopo stat. B	<i>ESISTE</i>	<i>PUÒ ESISTERE</i>
subito dopo stat. C	<i>ESISTE</i>	<i>ESISTE</i>
subito dopo stat. D	<i>NON ESISTE</i>	<i>PUÒ ESISTERE</i>

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		
	<i>stem</i>	<i>leaf</i>	<i>global</i>
subito dopo stat. A	<i>0</i>	<i>0</i>	<i>3</i>
subito dopo stat. B	<i>1</i>	<i>0</i>	<i>1</i>
subito dopo stat. C	<i>0 / 1</i>	<i>1</i>	<i>3</i>
subito dopo stat. D	<i>1</i>	<i>0</i>	<i>2</i>

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	<i>global</i>
1	<i>wait leaf</i>	<i>lock root</i>	<i>0</i>
2	<i>lock root</i>	<i>wait stem</i>	<i>3</i>

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) {		void * END (void * arg) {
if (glob == 2) {		
pthread_mutex_lock (&lock)		sem_wait (&pass)
sem_post (&pass)		glob = 3
pthread_mutex_unlock (&lock)		pthread_mutex_lock (&lock)
sem_post (&pass)		sem_wait (&pass)
} /* if */		pthread_mutex_unlock (&lock)
return NULL		sem_wait (&pass)
} /* BEGIN */		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:

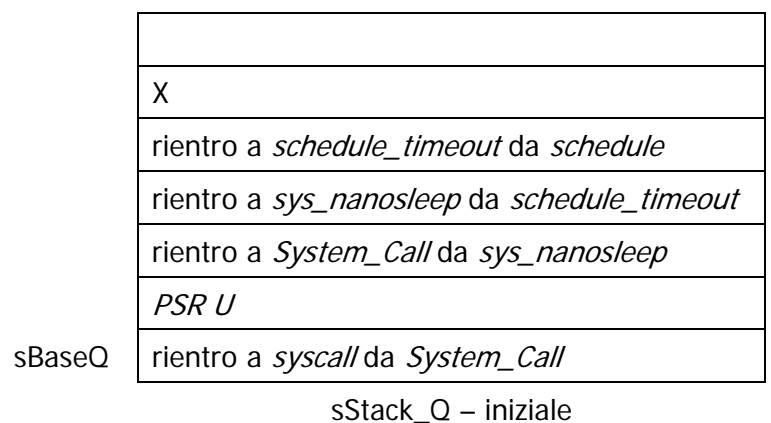
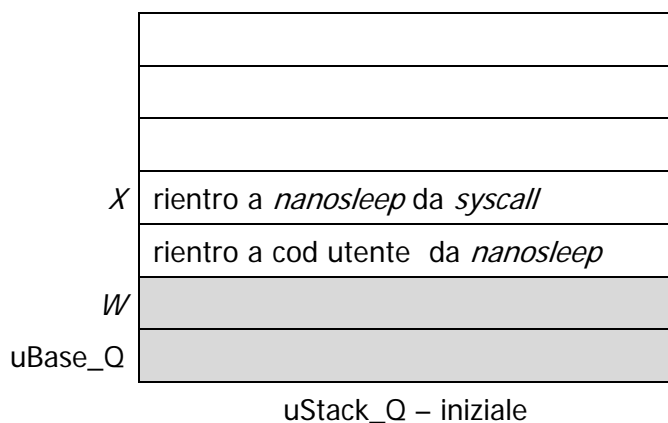
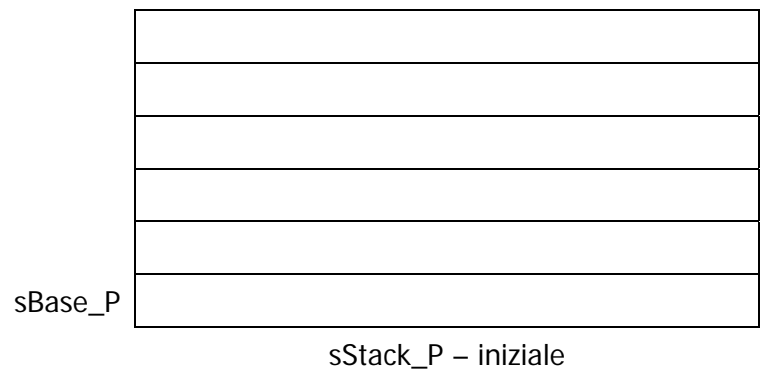
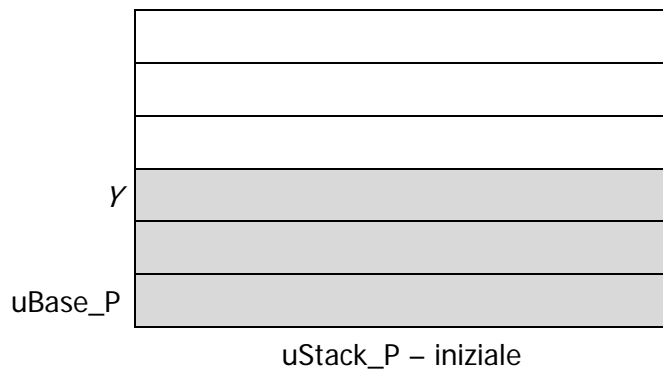
- $\langle PID, TGID \rangle$ di ciascun processo che viene creato
- $\langle \text{evento oppure identificativo del processo-chiamata di sistema / libreria} \rangle$ nella prima colonna, dove necessario e in funzione del codice proposto (le istruzioni da considerare sono evidenziate in grassetto)
- 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

<i>identificativo simbolico del processo</i>		<i>IDLE</i>	<i>P</i>	<i>Q</i>	<i>TH_2</i>	<i>TH_1</i>
<i>evento oppure processo-chiamata</i>	<i>PID</i>	1	2	3	4	5
	<i>TGID</i>	1	2	3	3	3
P –pid1=fork	0	pronto	esec	pronto	NE	NE
<i>interrupt</i> da RT_clock e scadenza quanto di tempo	1	<i>pronto</i>	<i>pronto</i>	<i>esec</i>	<i>NE</i>	<i>NE</i>
<i>Q – read</i>	2	<i>pronto</i>	<i>esec</i>	<i>A read</i>	<i>NE</i>	<i>NE</i>
<i>P– write</i>	3	<i>esec</i>	<i>A write</i>	<i>A read</i>	<i>NE</i>	<i>NE</i>
<i>interrupt</i> da <i>stdout</i> , tutti i caratteri trasferiti	4	<i>pronto</i>	<i>esec</i>	<i>A read</i>	<i>NE</i>	<i>NE</i>
<i>interrupt</i> da RT_clock e scadenza quanto di tempo	5	<i>pronto</i>	<i>esec</i>	<i>A read</i>	<i>NE</i>	<i>NE</i>
<i>interrupt</i> da <i>stdin</i> , tutti i caratteri trasferiti	6	pronto	pronto	esec	NE	NE
<i>Q-execI</i>	7	<i>pronto</i>	<i>pronto</i>	<i>esec</i>	<i>NE</i>	<i>NE</i>
<i>Q – pthread_create TH2</i>	8	<i>pronto</i>	<i>pronto</i>	<i>esec</i>	<i>pronto</i>	<i>NE</i>
<i>interrupt</i> da RT_clock e scadenza quanto di tempo	9	pronto	<i>esec</i>	<i>pronto</i>	<i>pronto</i>	<i>NE</i>
<i>P – wait</i>	10	<i>pronto</i>	<i>A wait</i>	<i>pronto</i>	<i>esec</i>	<i>NE</i>
<i>TH_2 – sem_wait</i>	11	<i>pronto</i>	<i>A wait</i>	<i>esec</i>	<i>A sem wait</i>	<i>NE</i>
<i>Q – sem_post</i>	12	<i>pronto</i>	<i>A wait</i>	<i>pronto</i>	<i>esec</i>	<i>NE</i>
<i>interrupt</i> da RT_clock e scadenza quanto di tempo	13	pronto	<i>A wait</i>	esec	pronto	NE
<i>Q– pthread_create TH1</i>	14	<i>pronto</i>	<i>A wait</i>	<i>esec</i>	<i>pronto</i>	<i>pronto</i>
<i>Q – exit</i>	15	esec	<i>A wait</i>	NE	NE	NE

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.



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 **Evento 2**.

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)

Evento 1 ed Evento 2

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

Tabella di invocazione dei moduli		
processo	modo	modulo
P	$U-S$	$> R_int (CK)$
P	S	$> task_tick <$
P	S	$> Controlla_timer$
P	S	$> wakeup_process$
P	S	$> check_preeempt_curr <$
P	S	$wakeup_process <$
P	S	$Controlla_timer <$
P	$S-U$	$R_int (CK): IRET <$
P	U	Codice utente
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 nessun processo)
P	S	$> schedule$
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)

la pagina Ps0 causa COW e viene riallocata

MEMORIA FISICA	
00: <ZP>	01: Pc0 / Qc0 / <X,0>
02: Pp0 / Qp0	03: Qs0 / <X,2>
04: Ps0	05: Pp1
06: Pp2	07: ----

SWAP FILE	
s0: Qp1	s1:
s2:	s3:

evento 2: read (Ps1)

PFRA – pagine liberate da inactive: Qs0 e Ps0; Qs0 non va in swap, perché è ancora mappata sul file X, mentre Ps0, essendo stata scritta, non lo è più e va in swap; viene caricata la pagina Ps1

MEMORIA FISICA	
00: <ZP>	01: Pc0 / Qc0 / <X,0>
02: Pp0 / Qp0	03: Ps1 / <X,3>
04:	05: Pp1
06: Pp2	07: ----

SWAP FILE	
s0: Qp1	s1: Ps0
s2:	s3:

evento 3: clone (R, c1)

viene creata la VMA di pila thread

VMA del processo P/R (compilare solo la riga relativa alla nuova VMA creata)							
AREA	NPV iniziale	dimensione	R/W	P/S	M/A	nome file	offset
T0	7FFFF77FE	2	W	P	A	-1	0

process R - NPV of PC and SP: _____ c1, t00

viene allocata la pagina di pila del thread, Rt00; il processo P e il thread R condividono tutte le pagine

MEMORIA FISICA	
00: <ZP>	01: PRc0 / Qc0 / <X,0>
02: PRp0 / Qp0	03: PRs1 / <X,3>
04: PRt00	05: PRp1
06: PRp2	07: ----

SWAP FILE	
s0: Qp1	s1: PRs0
s2:	s3:

evento 4: context switch (R)

PFRA – pagine liberate da inactive: PRp0 (condivisa anche con Q) e PRp2; entrambe vanno in swap (area pila anonima); viene caricata la pagina iniziale di codice di R, PRc1

MEMORIA FISICA	
00: <ZP>	01: PRc0 / Qc0 / <X,0>
02: PRc1 / <X,1>	03: PRs1 / <X,3>
04: PRt00	05: PRp1

06: ----	07: ----
----------	----------

SWAP FILE	
s0: $Qp1$	s1: $PRs0$
s2: $PRp0 / Qp0$	s3: $PRp2$

seconda parte – file system

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

$$\text{MAXFREE} = 2 \quad \text{MINFREE} = 1$$

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)

le pagine di pila di P e Q si separano (entrambe vengono modificate, ma Qp0 è marcata dirty nella tabella delle pagine perché non è nel TLB); il file F risulta aperto per P e Q, il file G solo per Q

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

Evento 4: write (fd1, 9000)

vengono lette da disco e caricate in memoria tre pagine del file F, e poi vengono scritte in memoria

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)

PFRA – vengono scritte su disco due pagine del file F, e poi vengono liberate; viene letta da disco e caricata in memoria una pagina del file G, e poi viene scritta in memoria

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)

il processo P (ma non il processo Q) chiude i suoi descrittori e scrive su disco la pagina del file G (che non risulta più aperto da nessun processo); il file F risulta ancora aperto dal processo Q

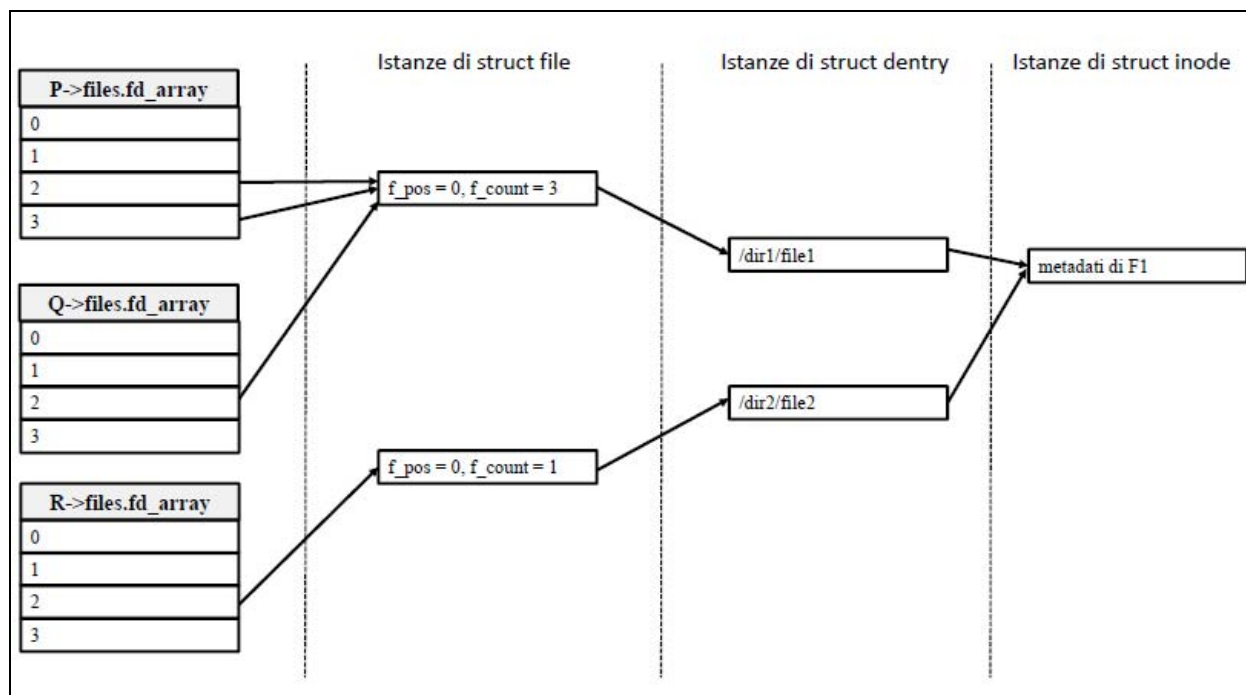
	f_pos	f_count	numero pagine lette	numero pagine scritte
file F	9000	1	3	2
file G	---	0	1	1

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.

<pre>struct task_struct { struct files_struct *files}</pre>	Ogni istanza rappresenta un descrittore di processo.
<pre>struct files_struct { struct file *fd_array [NR_OPEN_DEFAULT]}</pre>	fd_array[] costituisce la tabella dei (descrittori) dei file aperti da un processo.
<pre>struct file { struct dentry *f_dentry off_t f_pos //posizione corrente f_count //contatore riferim. al file aperto</pre>	Ogni istanza rappresenta un file aperto nel sistema.
<pre>struct dentry { struct inode *d_inode</pre>	Ogni istanza rappresenta un nodo (file o catalogo) nell'albero dei direttori del VFS.
<pre>struct inode { }</pre>	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.



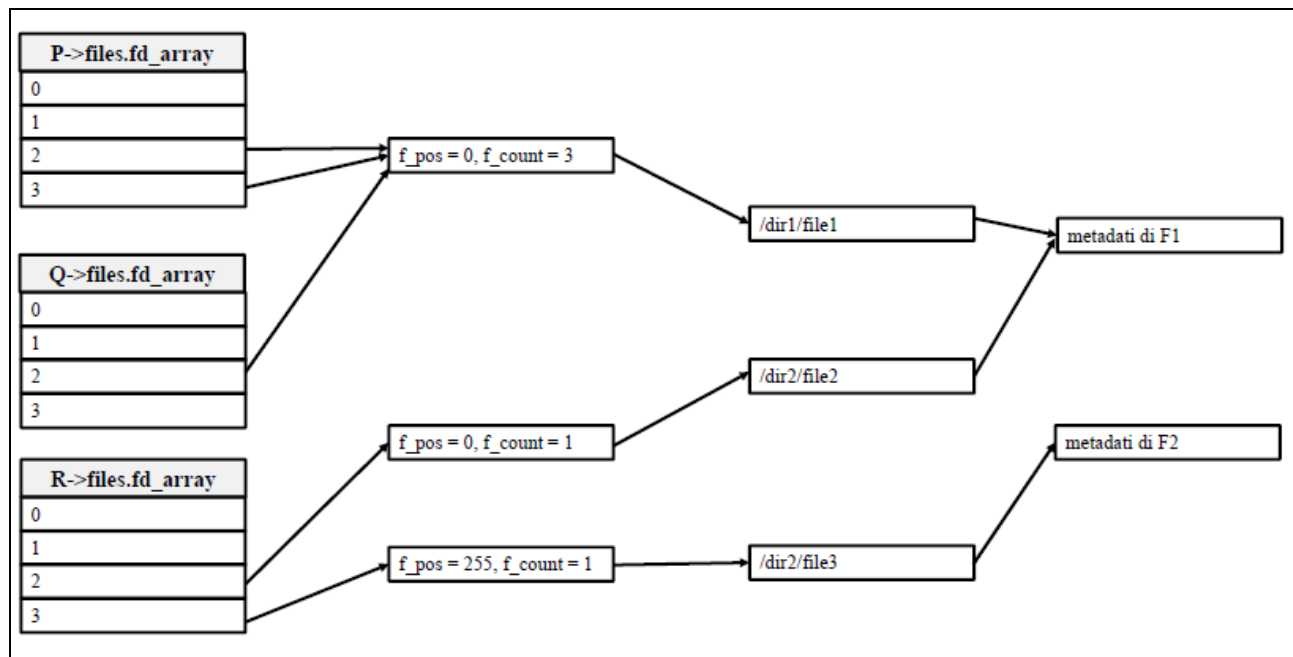
Chiamate di sistema eseguite nell'ordine indicato

- 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

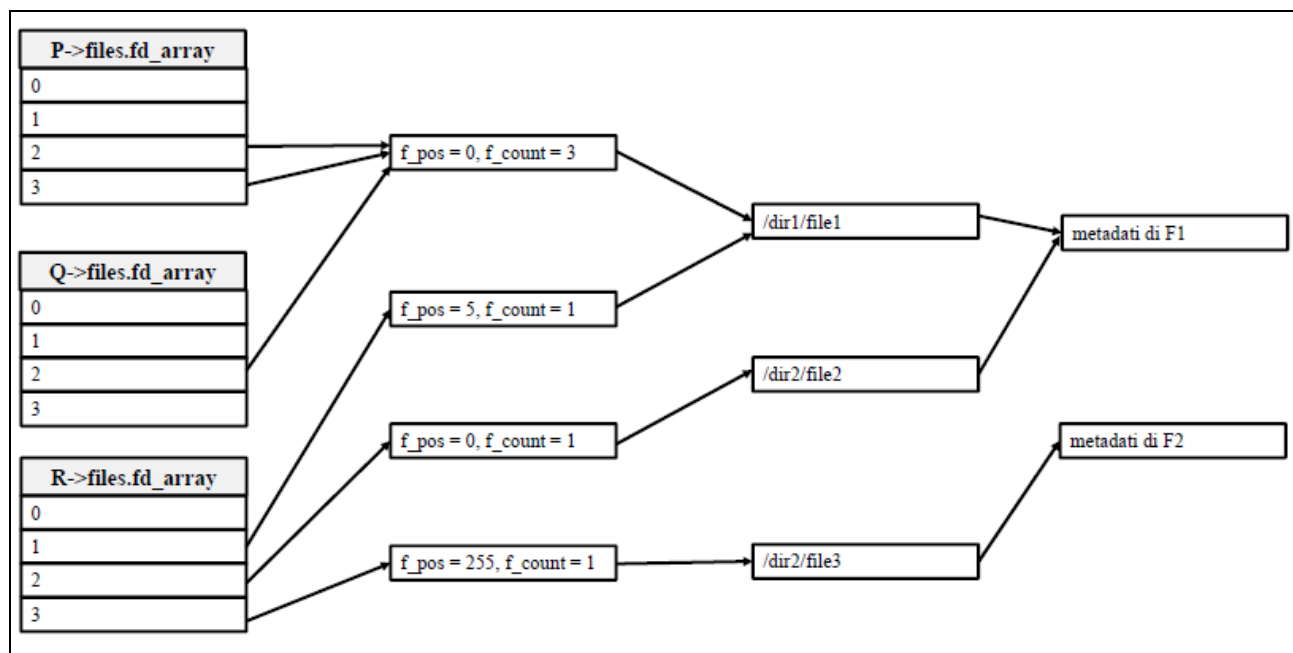


Chiamata/e di sistema

R: `fd1 = open (/dir2/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