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 23 luglio 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):

```
mutex_t brother, sister
sem_t cousin
int global = 0
```

```
void * parent (void * arg) {
   mutex_lock (&brother)
   sem_wait (&cousin)
   mutex_unlock (&brother)                          /* statement A */
   global = 1
   mutex_lock (&sister)
   sem_wait (&cousin)
   mutex_unlock (&sister)                           /* statement B */
   sem_post (&cousin)
   return NULL
} /* end parent */
```

```
void * child (void * arg) {
   mutex_lock (&sister)
   global = 2                                       /* statement C */
   sem_post (&cousin)
   mutex_unlock (&sister)
   sem_wait (&cousin)
   return (void * 3)
} /* end child */
```

```
void main ( ) {
   pthread_t th_1, th_2
   sem_init (&cousin, 0, 1)
   create (&th_1, NULL, parent, NULL)
   create (&th_2, NULL, child, NULL)
   join (th_2, &global)                             /* statement D */
   join (th_1, 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 – parent`	**`th_2 – child`**
subito dopo stat. **A**		
subito dopo stat. **B**		
subito dopo stat. **C**		
subito dopo stat. **D**		

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**		
	brother	*cousin*	*global*
subito dopo stat. **A**			
subito dopo stat. **B**			
subito dopo stat. **C**			
subito dopo stat. **D**			

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 – parent`**	**`th_2 – child`**	*global*
1			
2			
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
      execl ("/acso/prog_x", "prog_x", NULL)
      exit (-1)
   } else {
      nanosleep (5)
      write (stdout, msg, 25)
   } /* if */
   exit (0)
}  /* prova */
```

```
// programma prog_x.c
pthread_mutex_t MID = PTHREAD_MUTEX_INITIALIZER
sem_t TOP, BOT
```

void * ALPHA (void * arg) {		void * OMEGA (void * arg) {
pthread_mutex_lock (&MID)		sem_wait (&BOT)
sem_wait (&BOT)		pthread_mutex_lock (&MID)
sem_wait (&TOP)		pthread_mutex_unlock (&MID)
pthread_mutex_unlock (&MID)		sem_post (&TOP)
sem_post (&BOT)		sem_wait (&TOP)
return NULL		return NULL
} /* ALPHA */		} /* OMEGA */

```
main ( ) {  // codice eseguito da Q
    pthread_t TH_1, TH_2
    sem_init (&BOT, 0, 1)
    sem_init (&TOP, 0, 0)

    pthread_create (&TH_1, NULL, ALPHA, NULL)
    pthread_create (&TH_2, NULL, OMEGA, NULL)
    sem_post (&TOP)
    pthread_join (TH_2, NULL)
    pthread_join (TH_1, NULL)
    exit (1)
}  /* main */
```

Un processo ***P*** esegue il programma ***prova*** e crea un figlio ***Q*** che esegue una mutazione di codice (programma ***prog_x***). La mutazione di codice va a buon fine e sono creati i thread ***TH1*** e ***TH2***.

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:

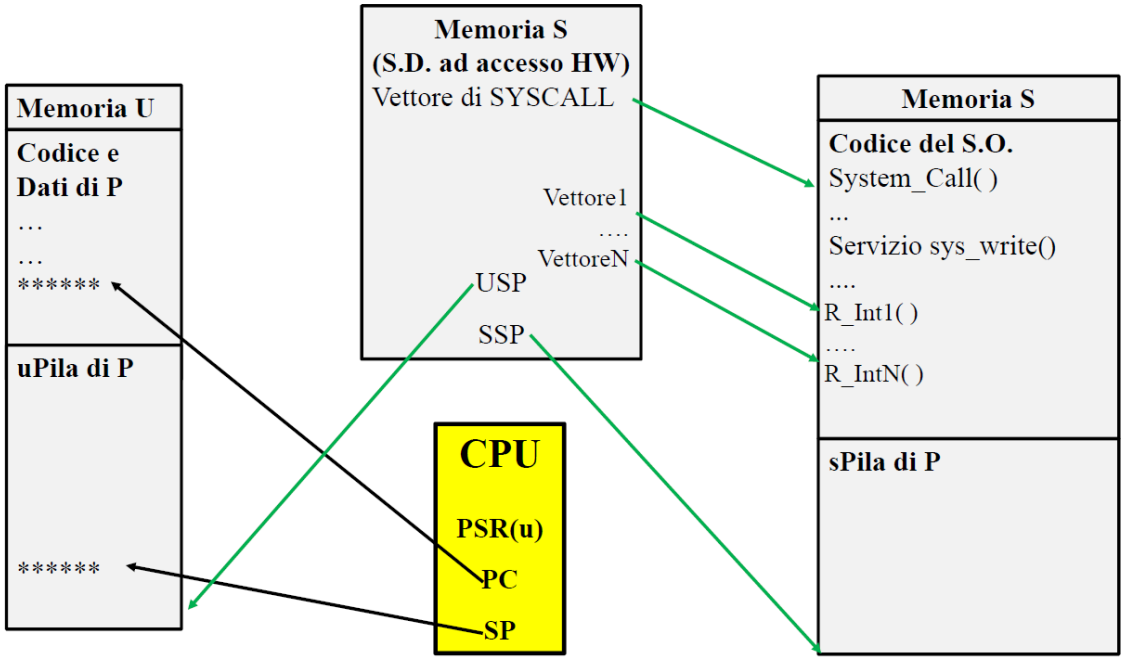
- ⟨ *PID*, *TGID* ⟩ di ciascun processo che viene creato
- ⟨ *identificativo del processo-chiamata di sistema / libreria* ⟩ nella prima colonna, dove necessario e in funzione del codice proposto
- in ciascuna riga lo stato dei processi **al termine dell'evento o della chiamata associata alla riga stessa**; 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**					
	PID	**1**						
evento oppure *processo-chiamata*	*TGID*	**1**						
Q – pthread_create TH1	**0**	**pronto**	**attesa nanosleep**					
interrupt da RT_clock e scadenza quanto tempo	1							
	2							
	3							
interrupt da RT_clock e scadenza timer di *nanosleep*	4							
	5							
	6							
	7	**pronto**	**A**	**pronto**	**esec**	**pronto**		
	8							
	9							
	10	**pronto**	**A**	**pronto**	**esec**	**A**		
	11							
25 interrupt da *stdout,* tutti i caratteri trasferiti	12							
	13							
	14							
	15	**pronto**	**ne**	**A**	**pronto**	**esec**		

seconda parte – stack e strutture dati HW

Si consideri un processo P in esecuzione in modo U della funzione *main.* La figura sotto riportata e i valori nella tabella successiva descrivono compiutamente, ai fini dell'esercizio, il contesto di P.



Si assuma che i valori della situazione iniziale di interesse siano i seguenti:

Processo P	
PC	X
SP	Y
SSP	Z
USP	W
Descrittore di P.stato	EXEC

// è all'interno di *main*

Si consideri la seguente **serie di eventi**.

Evento 1 – **Chiamata a funzione** eseguita dal codice utente di **P**

La funzione *main* esegue una chiamata alla funzione *scrivi_dati ( )* all'indirizzo X + 50.

Domanda: Completare la tabella seguente con i valori assunti dagli elementi subito dopo la chiamata alla funzione *scrivi_dati ( )* **il cui indirizzo iniziale è X + 50** e supponendo che occorra salvare **16 parole sulla pila.**

Processo P	
PC	
SP	
SSP	
USP	
Descrittore di P.stato	

Evento 2 – **Chiamata di sistema** eseguita dal processo **P**

La funzione *scrivi_dati ( )* esegue una chiamata di sistema al servizio *sys_write*.

Si assuma di essere all'interno della funzione *syscall* prima dell'esecuzione dell'istruzione macchina SYSCALL (passaggio di modo) e che i valori della situazione di interesse siano i seguenti:

Processo P	
PC	A
SP	B
SSP	Z
USP	W
Descrittore di P.stato	EXEC

// è all'interno di *syscall* prima di SYSCALL

Domanda: Completare le tabelle seguenti con i valori assunti dagli elementi subito dopo l'esecuzione dell'istruzione macchina SYSCALL.

Processo P	
PC	
SP	
SSP	
USP	
Descrittore di P.stato	

// non di interesse

sPila di P	

Evento 3 – **Interrupt durante** l'esecuzione della precedente chiamata di sistema

Si assuma che si verifichi un interrupt *R_Int1* all'interno della funzione *System_Call* subito dopo l'evento 2.

Domanda: Completare le tabelle seguenti con i valori assunti dagli elementi subito dopo il verificarsi dell'interrupt.

Processo P	
PC	
SP	
SSP	
USP	
Descrittore di P.stato	

// non di interesse

sPila di P	

esercizio n. 3 – memoria e file system

prima parte – memoria

È dato un sistema di memoria caratterizzato dai seguenti parametri generali (**ATTENZIONE a MAXFREE**):

MAXFREE = **3** **MINFREE** = **2**

Situazione iniziale (esiste un solo processo P)

```
PROCESSO: P ******************************************************
    ...

____MEMORIA FISICA____(pagine libere: 3)_____________________
      00 : <ZP>                ||  01 : Pc1 / <X,1>          ||
      02 : ----                ||  03 : Pp2                  ||
      04 : Ps0                 ||  05 : Pp1                  ||
      06 : ----                ||  07 : ----                 ||
____STATO del TLB_____________________________________________
      Pc1 : 01 -  0: 1:   ||          -----          ||
      Ps0 : 04 -  1: 0:   ||   Pp1 : 05 -  1: 0:     ||
      Pp2 : 03 -  1: 0:   ||          -----          ||
            -----         ||          -----          ||
SWAP FILE:      Pp0, ----, ----, ----, ----, ----,
LRU ACTIVE:     PC1
LRU INACTIVE:   pp2, pp1, ps0
```

evento 1: *write* (Pp3, Pd0)

MEMORIA FISICA	
00: <ZP>	01: Pc1 / <X, 1>
02: Pp3	03: Pp2
04: Pd0	05: ----
06: ----	07: ----

SWAP FILE	
s0: Pp0	s1: Ps0
s2: Pp1	s3:

LRU active: PD0, PP3, PC1

LRU inactive: pp2

evento 1 bis: *read* (Pc1) – 4 *kswapd*

LRU active: PC1

LRU inactive: pd0, pp3, pp2

evento 2: *mmap* (0x50000, 3, W, P, M, "F", 2), *read* (Pm01), *write* (Pm01)

MEMORIA FISICA	
00: <ZP>	01: Pc1 / <X, 1>
02: Pm01	03: - - - -
04: Pd0	05: <F, 3>
06: - - - -	07: - - - -

SWAP FILE	
s0: Pp0	s1: Ps0
s2: Pp1	s3: Pp2
s4: Pp3	s5:

LRU active: PM01, PC1

LRU inactive: pd0

evento 3: *read* (Pm02), *write* (Pm02)

MEMORIA FISICA	
00: <ZP>	01: Pc1 / <X, 1>
02: Pm01	03: <F, 4>
04: Pm02	05: - - - -
06: - - - -	07: - - - -

SWAP FILE	
s0: Pp0	s1: Ps0
s2: Pp1	s3: Pp2
s4: Pp3	s5: Pd0

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**:

Un processo P ha aperto un file F con descrittore fd e poi ha creato (fork) un processo Q; il processo P è ancora in esecuzione:

MEMORIA FISICA	
00: <ZP>	01: Pc0 / <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

evento 1: *write* (fd, 12000)

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	12000	2	3	0

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

evento 2: *write* (fd, 2000)

12000 + 2000 = 14000

MEMORIA FISICA	
00: <ZP>	01: Pc0 / Qc0 / <X, 0>
02: Qp0 (D)	03: Pp0
04: <F, 3> (D)	05: - - - -
06: <F, 2> (D)	07:

	f_pos	f_count	numero pagine lette	numero pagine scritte
file F	14000	2	4	2

evento 3: *contextswitch* (Q), *write* (fd, 10000)

14000 + 10000 = 24000

MEMORIA FISICA	
00: <ZP>	01: Pc0 / Qc0 / <X, 0>
02: Qp0 (D)	03: Pp0 (D)
04: <F, 5> (D)	05: - - - -
06: <F, 2> (D)	07:

	f_pos	f_count	numero pagine lette	numero pagine scritte
file F	24000	2	6	4

evento 4: *write* (fd, 40960)

24000 + 40960 = 64960

MEMORIA FISICA	
00: <ZP>	01: Pc0 / Qc0 / <X, 0>
02: Qp0 (D)	03: Pp0 (D)
04: <F, 15> (D)	05:
06: <F, 2> (D)	07:

	f_pos	f_count	numero pagine lette	numero pagine scritte
file F	64960	2	16	14

esercizio n. 4 – virtual file system (VFS)

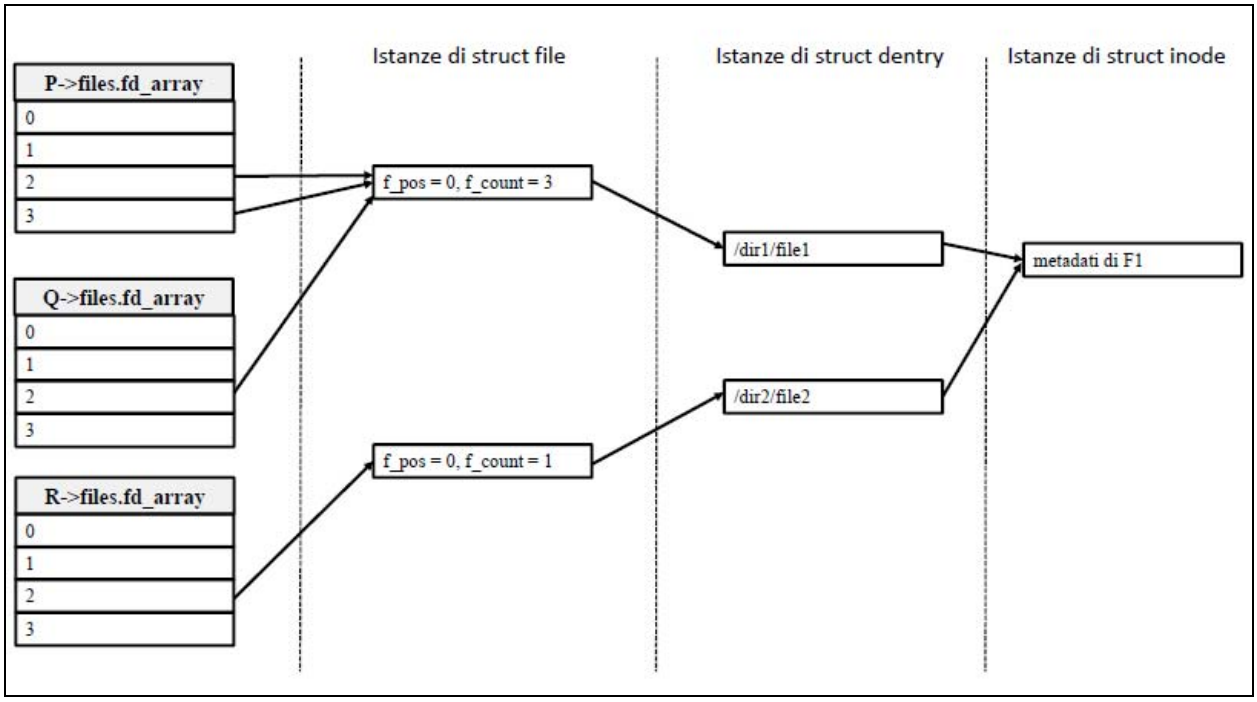
Si riportano nel seguito gli elementi di interesse di alcune `struct` necessarie alla gestione, organizzazione e accesso di file e cataloghi.

`struct task_struct { ……` `  struct files_struct *files ……}`	Ogni istanza rappresenta un descrittore di processo.
`struct files_struct { ……` `  struct file *fd_array [NR_OPEN_DEFAULT] ……}`	`fd_array[]` costituisce la tabella dei (descrittori) dei file aperti da un processo.

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

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



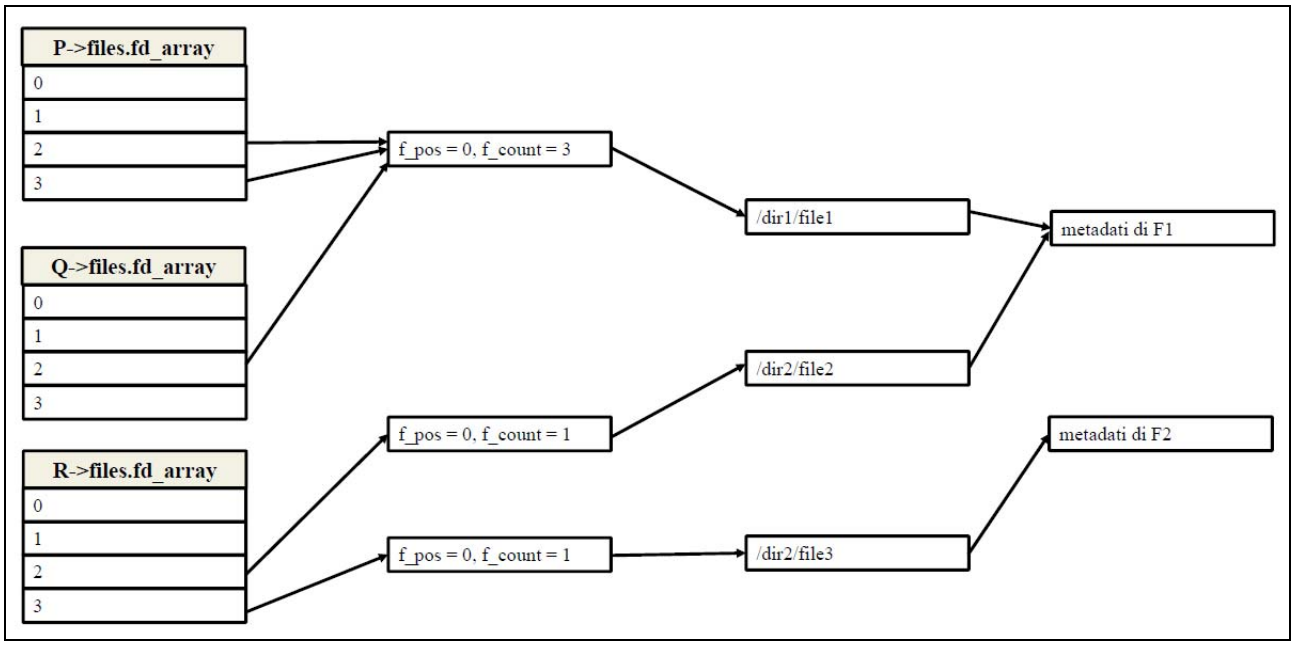
Chiamate di sistema eseguite nell'ordine indicato

- 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()`

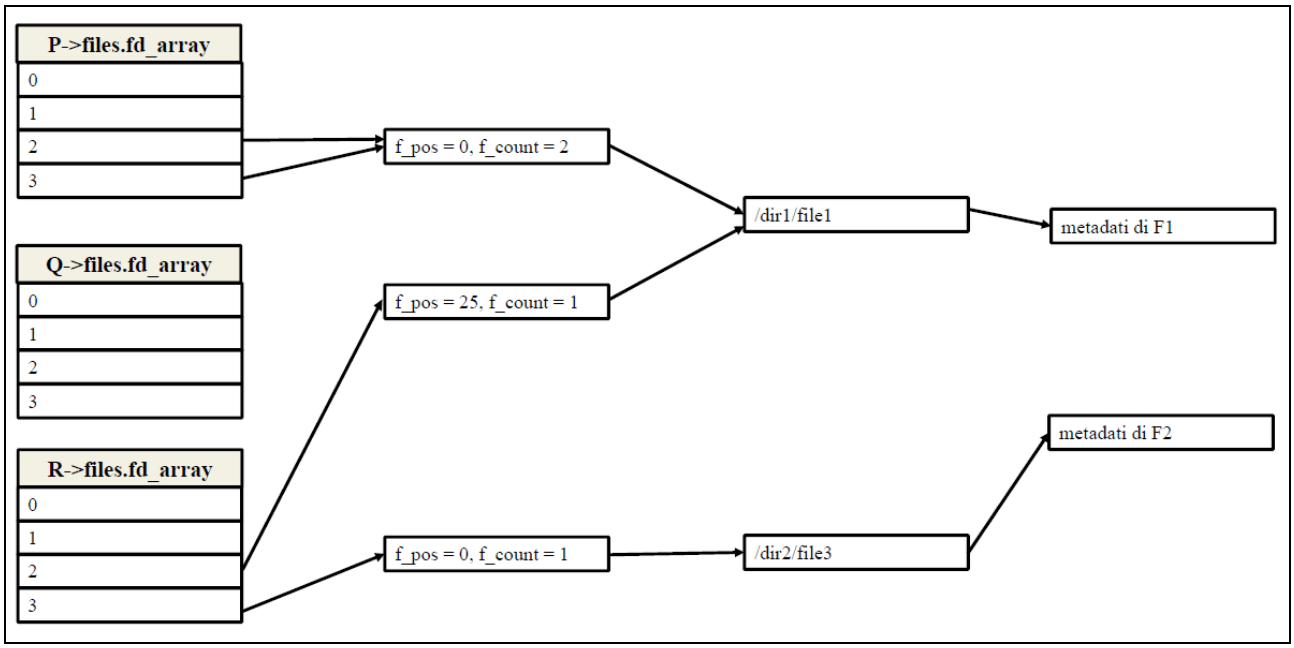
Domanda 1



Chiamata/e di sistema

R: fd1 = open("dir2/file3")

Domanda 2



Chiamata/e di sistema

Q: close(1)

R: close(2)

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

R: read(fd2, 25)

spazio libero per brutta copia o continuazione