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) _____

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**		
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**			
	go	*come*	*stay*	*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) **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			
2			
3			

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	
	PID	1	2			
evento oppure processo-chiamata	TGID	1	2			
P – create TH_3	1	pronto	esec	pronto	NE	NE
	2					
interrupt da RT_clock e scadenza del quanto di tempo	3					
	4					
	5					
P – create TH_1	6					
	7					
	8					
	9					
	10	pronto	attesa	esec	attesa	pronto
	11					
	12					
	13					
	14					
	15	pronto	attesa	attesa	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**			
RUNQUEUE	**NRT**	**PER**	**RQL**	**CURR**	**VMIN**		
		6					
TASK	**ID**	**LOAD**	**LC**	**Q**	**VRTC**	**SUM**	**VRT**
CURRENT							
RB							
WAITING							

EVENTO 2	**TIME**	**TYPE**	**CONTEXT**	**RESCHED**			
RUNQUEUE	**NRT**	**PER**	**RQL**	**CURR**	**VMIN**		
		6					
TASK	**ID**	**LOAD**	**LC**	**Q**	**VRTC**	**SUM**	**VRT**
CURRENT							
RB							
WAITING							

EVENTO 3		TIME	TYPE	CONTEXT	RESCHED		
RUNQUEUE	NRT	PER	RQL	CURR	VMIN		
		6					
TASK	ID	LOAD	LC	Q	VRTC	SUM	VRT
CURRENT							
RB							
WAITING							

EVENTO 4		TIME	TYPE	CONTEXT	RESCHED		
RUNQUEUE	NRT	PER	RQL	CURR	VMIN		
		6					
TASK	ID	LOAD	LC	Q	VRTC	SUM	VRT
CURRENT							
RB							
WAITING							

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

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

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  7FFFFFFB,  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*

PT del processo: **P**				
p0: 3 D W	p1: 2 W	p2: 7 W	p3:	p4:
p5:				

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

MEMORIA FISICA	
00: <ZP>	01:
02:	03:
04:	05:
06:	07:
08:	09:

LRU ACTIVE: _____

LRU INACTIVE: _____

evento 2: *fork* (R)

PT del processo: **R**				
p0:	p1:	p2:	p3:	p4:
p5:				

process R	NPV of **PC**:	NPV of **SP**:

MEMORIA FISICA	
00: <ZP>	01:
02:	03:
04:	05:
06:	07:
08:	09:

LRU ACTIVE: _____

LRU INACTIVE: _____

evento 3: *clone* (S, c0)

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						

PT dei processi: **P/S**				
p0:	p1:	p2:	p3:	p4:
p5:	t00:	t01:		

process P	NPV of **PC**:	NPV of **SP**:
process S	NPV of **PC**:	NPV of **SP**:

MEMORIA FISICA	
00: <ZP>	01:
02:	03:
04:	05:
06:	07:
08:	09:

SWAP FILE	
s0:	s1:
s2:	s3:
s4:	s5:

LRU INACTIVE: _____

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)

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

nome file	f_pos	f_count	numero pag. lette	numero pag. scritte
F				

evento 3: *read* (fd, 7000)

MEMORIA FISICA	
00: <ZP>	01: Pc2 / Qc2 / <X, 2>
02:	03:
04:	05:
06:	07:

nome file	f_pos	f_count	numero pag. lette	numero pag. scritte
F				

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

nome file	f_pos	f_count	numero pag. lette	numero pag. scritte
F				

evento 5: *write* (fd, 1000)

MEMORIA FISICA	
00: <ZP>	01: Pc2 / Qc2 / <X, 2>
02: Qp0 D	03: Pp0 D
04:	05:
06:	07:

nome file	f_pos	f_count	numero pag. lette	numero pag. scritte
F				

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

MEMORIA FISICA	
00: <ZP>	01: Pc2 / Qc2 / <X, 2>
02: Qp0 D	03: Pp0 D
04:	05:
06:	07:

nome file	f_pos	f_count	numero pag. lette	numero pag. scritte
F				
H				

esercizio n. 4 – domande varie (due)

prima domanda – moduli del SO

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

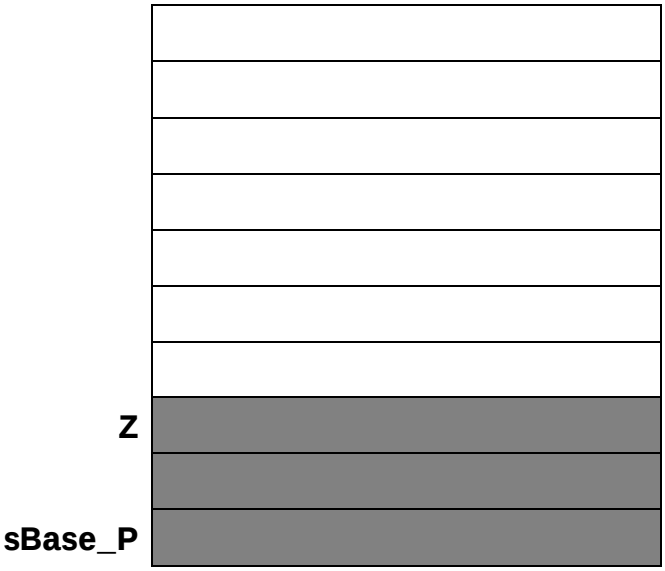
Si consideri il seguente evento: il processo **P** è in esecuzione in **modo U** e si verifica un **interrupt da DMA** di completamento di un'operazione di **lettura da disco**. Si assuma che il processo **Q**, al suo risveglio, abbia acquisito **diritti maggiori** di esecuzione rispetto a **P**.

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

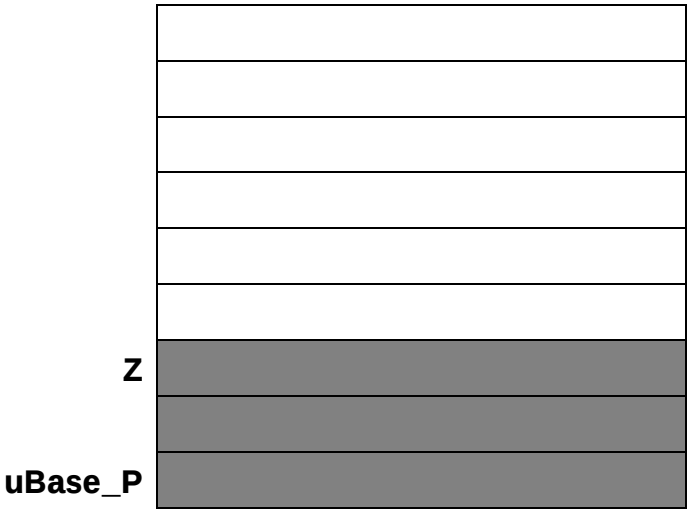
invocazione moduli

processo	modo	modulo
P	U – S	> R_int (E)
P	S	> wakeup
P	S	> check_preempt_curr
P	S	> resched <
P	S	check_preempt_curr <
P	S	wakeup <
P	S	> schedule
P	S	> pick_next_task <
P – Q	S	CONTEXT_SWITCH (schedule)

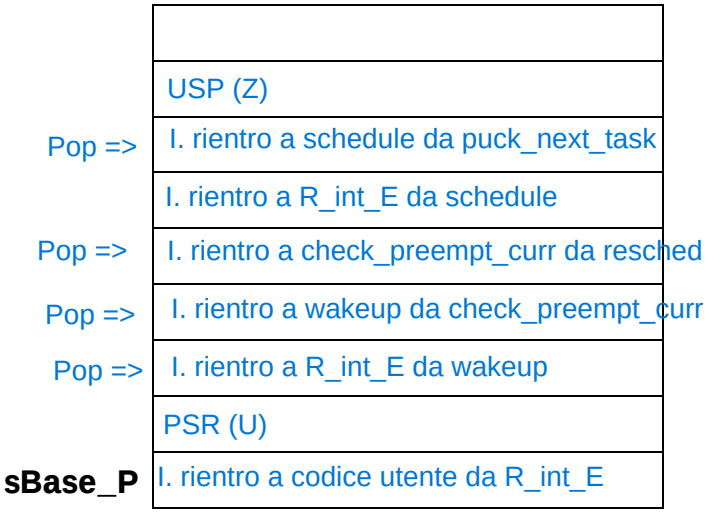


uStack_P – iniziale

contenuto stack al termine dell'evento



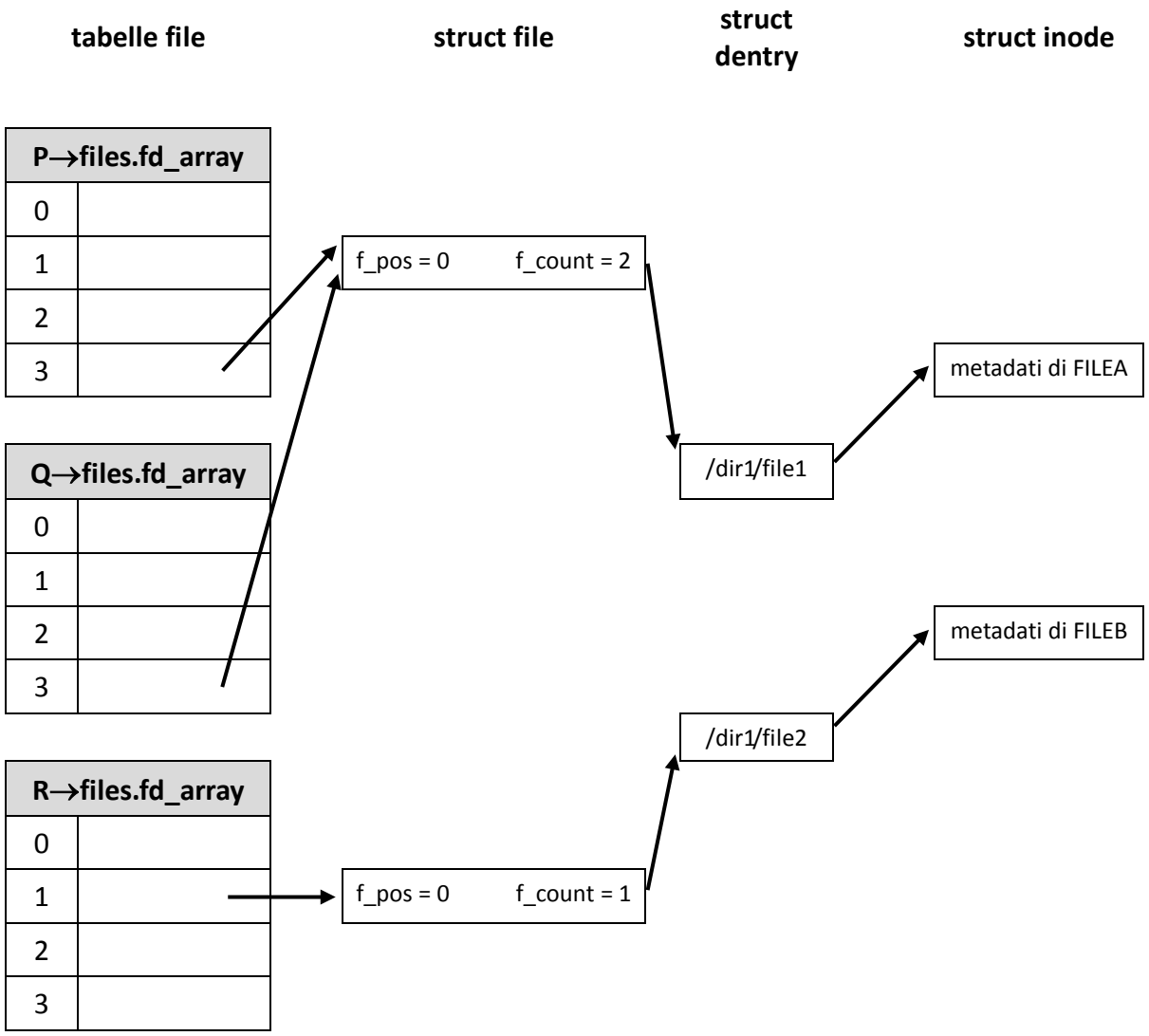
uStack_P – finale



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.



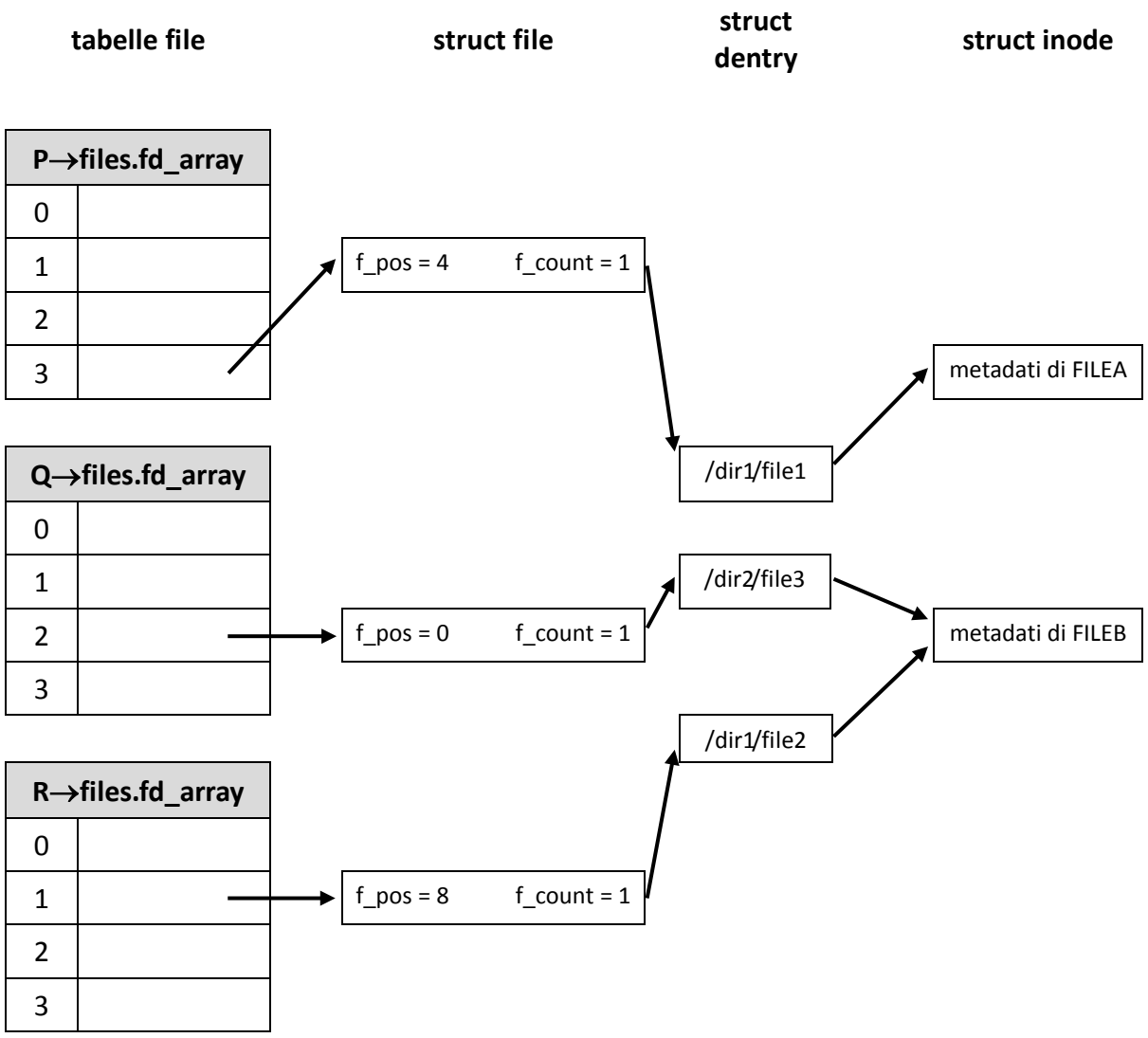
chiamate di sistema eseguite nell'ordine indicato

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(3)
4	Q	close(2)
5	Q	fd1 = open("/dir2/file3")
6		
7		
8		