



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 7 febbraio 2018

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 zone
```

```
sem_t stay, leave
```

```
int global = 0
```

```
void * sign (void * arg) {
```

```
    sem_wait (&leave)
```

```
    mutex_lock (&zone)
```

```
    sem_wait (&stay)
```

```
    mutex_unlock (&zone)
```

```
/* statement A */
```

```
    mutex_lock (&zone)
```

```
    sem_post (&leave)
```

```
    global = 1
```

```
/* statement B */
```

```
    mutex_unlock (&zone)
```

```
    return (void * 2)
```

```
} /* end sign */
```

```
void * tag (void * arg) {
```

```
    mutex_lock (&zone)
```

```
    global = 3
```

```
    sem_post (&stay)
```

```
    mutex_unlock (&zone)
```

```
/* statement C */
```

```
    mutex_lock (&zone)
```

```
    global = 4
```

```
    sem_wait (&leave)
```

```
    mutex_unlock (&zone)
```

```
    return NULL
```

```
} /* end tag */
```

```
void main ( ) {
```

```
    pthread_t th_1, th_2
```

```
    sem_init (&stay, 0, 0)
```

```
    sem_init (&leave, 0, 1)
```

```
    create (&th_1, NULL, sign, NULL)
```

```
    create (&th_2, NULL, tag, 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 – sign	th_2 – tag
subito dopo stat. A	<i>ESISTE</i>	<i>ESISTE</i>
subito dopo stat. B	<i>ESISTE</i>	<i>ESISTE</i>
subito dopo stat. C	<i>PUÒ ESISTERE</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>leave</i>	<i>global</i>
subito dopo stat. A	<i>0</i>	<i>3 / 4</i>
subito dopo stat. B	<i>1</i>	<i>1</i>
subito dopo stat. C	<i>0 / 1</i>	<i>1 / 2 / 3</i>

Il sistema può andare in stallo (*deadlock*), con uno o più *thread* che si bloccano, in **tre 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:

caso	th_1 – sign	th_2 – tag
1	<i>wait stay</i>	<i>1a lock zone</i>
2	<i>2a lock zone</i>	<i>wait leave</i>
3	<i>wait leave</i>	<i>-</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	
execl ("/acso/check_randomness", "check_randomness", "/dev/random", NULL)	
write (stdout, error_msg, 50)	
} else {	
pid1 = wait (&status)	
write (stdout, msg, 25)	
} /* if */	
exit (0)	
} /* prova */	
// programma check_randomness.c	
uint8_t file_data[200]	
fd file_fd	
uint8_t buffer	
sem_t produced, consumed	
void * PRODUCER (void * arg) {	void * CONSUMER (void * arg) {
for (int i = 0; i < 200; ++i) {	for (int i = 0; i < 200; ++i) {
sem_wait (&consumed)	sem_wait (&produced)
read (file_fd, &buffer, 1)	file_data[i] = buffer
sem_post (&produced)	sem_post (&consumed)
} /* for */	} /* for */
return NULL	return NULL
} /* PRODUCER */	} /* CONSUMER */
main () { // codice eseguito da Q	
pthread_t TH_1, TH_2	
sem_init (&consumed, 0, 1)	
sem_init (&produced, 0, 0)	
file_fd = open (argv [1], O_RDONLY)	
pthread_create (&TH_2, NULL, CONSUMER, NULL)	
pthread_create (&TH_1, NULL, PRODUCER, NULL)	
pthread_join (TH_2, NULL)	
pthread_join (TH_1, NULL)	
float average = compute_average (file_data)	
float standard_deviation = compute_stdev (file_data, average)	
int randomness = (int) (average / standard_deviation) * 100	
exit (randomness > 10)	
} /* main */	

Un processo **P** esegue il programma **prova** e crea un figlio **Q** che esegue una mutazione di codice (programma **check_randomness**). 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:

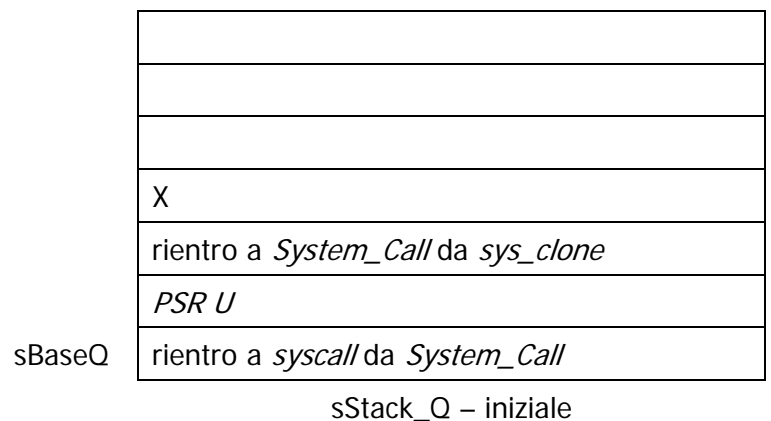
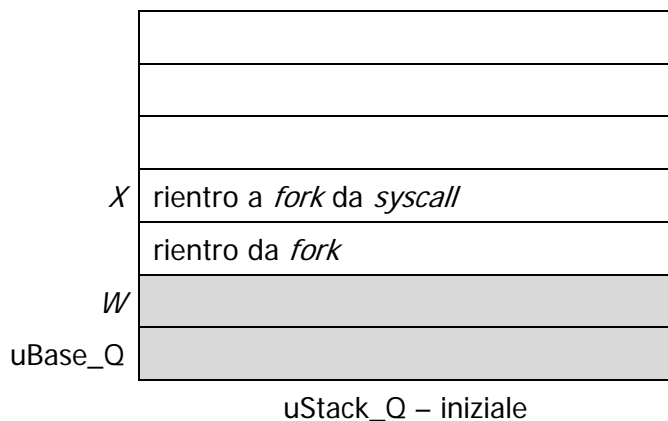
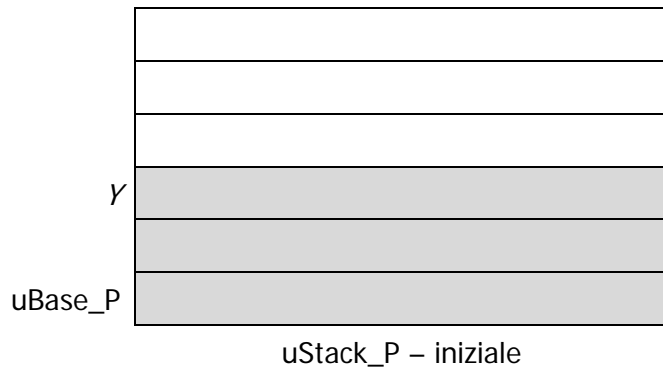
- $\langle PID, TGID \rangle$ di ciascun processo che viene creato
- $\langle \text{evento oppure identificativo del processo-chiamata di sistema / libreria} \rangle$ nella prima colonna, dove necessario e in funzione del codice proposto (le istruzioni da considerare sono evidenziate in grassetto)
-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

identificativo simbolico del processo		<i>IDLE</i>	<i>P</i>	<i>Q</i>	<i>TH_2</i>	<i>TH_1</i>
evento oppure processo-chiamata	<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
interrupt da RT_clock e scadenza quanto di tempo	1	pronto	pronto	esec	NE	NE
Q – exec/	2	pronto	pronto	esec	NE	NE
Q – open	3	pronto	esec	A open	NE	NE
interrupt da DMA_in, tutti i blocchi richiesti trasferiti	4	pronto	pronto	esec	NE	NE
Q – pthread_create TH2	5	pronto	pronto	esec	pronto	NE
Q – pthread_create TH1	6	pronto	pronto	esec	pronto	pronto
Q – pthread_join TH2	7	pronto	esec	A join TH2	pronto	pronto
P – wait	8	pronto	A wait	A join	esec	pronto
TH_2 – sem_wait (prod)	9	pronto	A wait	A join	A sem wait prod	esec
TH_1 – sem_wait (cons)	10	pronto	A wait	A join	A sem wait	esec
TH_1 – read	11	esec	A wait	A join	A sem wait	A read
interrupt da DMA_in, tutti i blocchi richiesti trasferiti	12	pronto	A wait	A join	A sem wait	esec
TH_1 – sem_post (prod)	13	pronto	A wait	A join	esec	pronto
TH_2 – sem_post (cons)	14	pronto	A wait	A join	esec	pronto
TH_2 – sem_wait (prod)	15	pronto	A wait	A join	A sem wait prod	esec
TH_1 – sem wait (cons)	16	pronto	A wait	A join	A sem wait	esec

seconda parte – moduli del SO

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



- **Si indichi** lo stato dei processi così come deducibile dallo stato iniziale delle pile:

P : *in esecuzione*

Q : *pronto (appena creato)*

- Per l'evento indicato **si mostrino** le invocazioni di tutti i **moduli** (e eventuali relativi ritorni) per la gestione dell'evento stesso (precisando processo e modo) e – come specificato nella descrizione – il **contenuto delle pile** utente e di sistema di P.

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

Evento: *wait*

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

contenuto della pila

<i>processo</i>	<i>modo</i>	<i>modulo</i>
<i>P</i>	<i>U</i>	<i>> wait</i>
<i>P</i>	<i>U</i>	<i>> syscall</i>
<i>P</i>	<i>U-S</i>	<i>SYSCALL: >system_call</i>
<i>P</i>	<i>S</i>	<i>> sys_wait</i>
<i>P</i>	<i>S</i>	<i>> schedule</i>
<i>P</i>	<i>S</i>	<i>> pick_next_task <</i>
<i>P - Q</i>	<i>S</i>	<i>schedule: context_switch</i>
<i>Q</i>	<i>S</i>	<i>schedule <</i>
<i>Q</i>	<i>S-U</i>	<i>system_call: SYSRET<</i>
<i>Q</i>	<i>U</i>	<i>syscall <</i>
<i>Q</i>	<i>U</i>	<i>fork <</i>
<i>Q</i>	<i>U</i>	<i>codice utente</i>

sBase_P

<i>USP salvato = Y - 2</i>
<i>rientro a sys_wait da schedule</i>
<i>rientro a system_call da sys_wait</i>
<i>PSR U</i>
<i>rientro a syscall da system_call</i>

sStack_P

Y - 2

Y - 1

Y

uBase_P

<i>rientro a wait da syscall</i>
<i>rientro a codice utente da wait</i>

uStack_P

esercizio n. 3 – memoria e file system

prima parte – memoria

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

MAXFREE = 3 MINFREE = 2

Si consideri la seguente **situazione iniziale**:

PROCESSO: P

VMA: ...

PT: <c0 :1 R> <c1 :- ->
 <d0 :s1 R> <d1 :7 W> <d2 :2 W> <d3 :- ->
 <p0 :6 W> <p1 :5 R> <p2 :- ->

process P - NPV of PC and SP: c0, p1

PROCESSO: Q

...

PROCESSO: R

VMA: ...

PT: <c0 :1 R> <c1 :- ->
 <d0 :s1 R> <d1 :- -> <d2 :- -> <d3 :- ->
 <p0 :3 D W> <p1 :5 R> <p2 :- ->

process R - NPV of PC and SP: c0, p0

MEMORIA FISICA (pagine libere: 3)			
00 : <ZP>	01 : Pc0 / Qc0 / Rc0 / <X,0>		
02 : Pd2	03 : Rp0 D		
04 : ----	05 : Pp1 / Rp1		
06 : Pp0	07 : Pd1		
08 : ----	09 : ----		

STATO del TLB			
Pc0 : 01 - 0: 1:	Pp0 : 06 - 1: 1:		
Pd2 : 02 - 1: 0:	Pp1 : 05 - 1: 1:		
Pd1 : 07 - 1: 0:	-----		
-----	-----		

SWAP FILE: Qp0, Pd0 / Rd0, ----, ----, ----, ----,

LRU ACTIVE: PP0, PC0, PP1,

LRU INACTIVE: pd2, pd1, rp0, rc0, rp1, qc0,

Si rappresenti l'effetto dei seguenti eventi consecutivi sulle strutture dati della memoria compilando esclusivamente le tabelle fornite per ciascun evento (l'assenza di una tabella significa che non è richiesta la compilazione della corrispondente struttura dati).

ATTENZIONE: le Tabelle sono PARZIALI – riempire solamente le celle indicate

evento 1 – read (Pd0)

PT del processo: P				
c0: 1 R	d0: 4 R	d1: 7 W	p0: 6 W	p2: - -
PT del processo: R				
c0: 1 R	d0: 4 R	d1: - -	p0: 3 D W	p2: - -

MEMORIA FISICA	
00: <ZP>	01: Pc0 / Qc0 / Rc0 / <X,0>
02: Pd2	03: Rp0 D
04: Pd0 / Rd0	05: Pp1 / Rp1
06: Pp0	07: Pd1
08:	09:

SWAP FILE	
s0: Qp0	s1: Pd0 / Rd0
s2:	s3:
s4:	s5:

Active: _____ PD0, PP0, PC0, PP1, Inactive: _____ pd2, pd1, rp0, rc0, rp1, qc0, rd0

evento 2 – write (Pp2)

PT del processo: P				
c0: 1 R	d0: 4 R	d1: s3 W	p0: 6 W	p2: 3 W
PT del processo: R				
c0: 1 R	d0: 4 R	d1: - -	p0: s2 W	p2: - -

MEMORIA FISICA	
00: <ZP>	01: Pc0 / Qc0 / Rc0 / <X,0>
02: Pd2	03: Pp2
04: Pd0 / Rd0	05: Pp1 / Rp1
06: Pp0	07:
08:	09:

SWAP FILE	
s0: Qp0	s1: Pd0 / Rd0
s2: Rp0	s3: Pd1
s4:	s5:

seconda parte – file system

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

MAXFREE = 3 MINFREE = 2

Si consideri la seguente **situazione iniziale**:

MEMORIA FISICA (pagine libere: 5)			
00 : <ZP>		01 : Pc1 / <X,1>	
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.

evento 1 – fd = *open* (F)

	f_pos	f_count	numero pagine lette	numero pagine scritte
file F	0	1		

evento 2 – write (fd, 16000)

MEMORIA FISICA	
00: <ZP>	01: Pc1 / <X,1>
02: Pp0	03: <F,0> D <F,3> D
04: <F,1> D ----	05: <F,2> D
06: ----	07: ----

	f_pos	f_count	numero pagine lette	numero pagine scritte
file F	16000	1	4	2

eventi 3 e 4 – lseek (fd, -11000) write (fd, 16000)

MEMORIA FISICA	
00: <ZP>	01: Pc1 / <X,1>
02: Pp0	03: <F,3> D <F,4> D
04: <F,1> D <F,5> D	05: <F,2> D
06: ----	07: ----

	f_pos	f_count	numero pagine lette	numero pagine scritte
file F	21000	1	7	4

esercizio n. 4 – domanda – file system

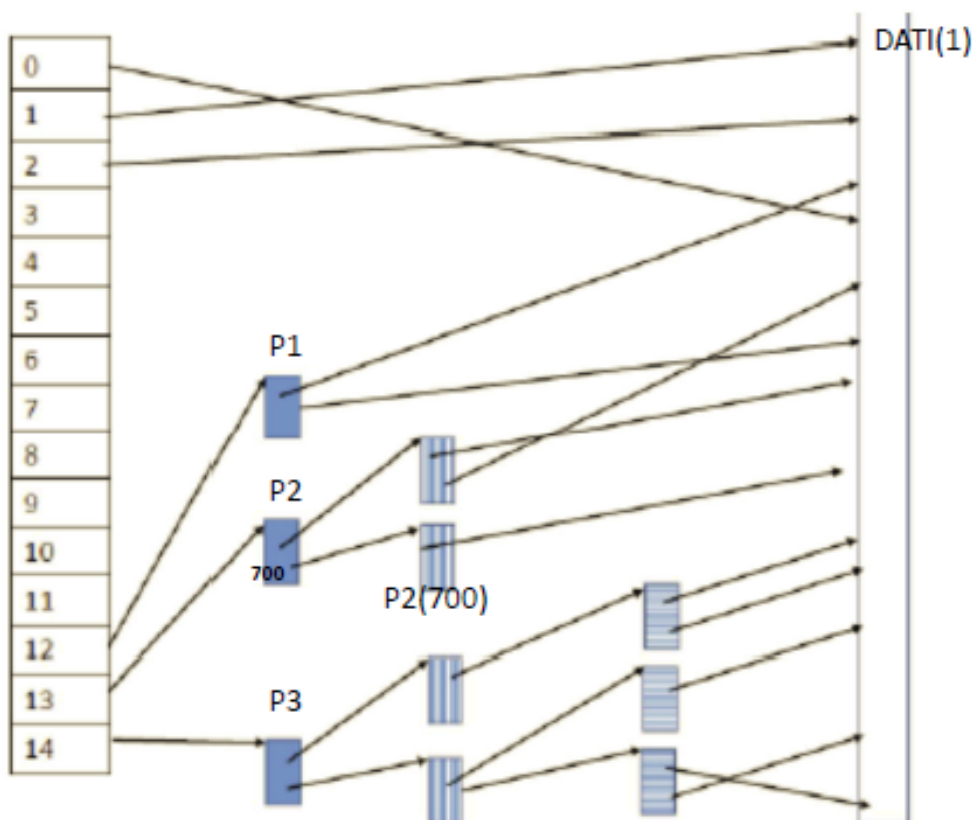
Si consideri il FS ext2 con dimensione di blocco = dimensione pagina = **4 Kbyte**.

La figura seguente mostra il meccanismo con cui vengono referenziati i blocchi / pagine dati di un file, tramite il suo i-node. Ogni puntatore occupa **4 byte** e quindi un blocco di puntatori contiene **1024** puntatori, numerati **da 0 a 1023**.

Con riferimento alla figura si consideri la seguente notazione:

- DATI (N) indica la pagina dati del file in posizione N; DATI (0) è la prima pagina dati del file
- P1, P2 e P3 sono i tre blocchi contenenti puntatori di indirezione semplice
- $P_i(j)$ indica un blocco di puntatori al secondo livello di indirezione, raggiunto dal puntatore in posizione j contenuto nel blocco i di indirezione semplice; per esempio, nella figura qui sotto, P2 (700) indica il blocco di secondo livello di indirezione raggiunto dal puntatore in posizione 700 del blocco P2 di indirezione semplice (si ricordi che i puntatori sono numerati partendo da 0, dunque il puntatore in posizione 700 è il 701-esimo del blocco P2)

contenuto dello i-node:
sono rappresentati solo i
puntatori per referenziare
i blocchi / pagine del file



- a) Indicare il massimo numero di blocchi dati di file (in questo caso tale numero coincide con il numero di pagine del file) accessibili tramite ogni singolo livello di indirezione.

Ogni blocco contiene 1024 puntatori.

livello di indirezione	massimo numero di blocchi o pagine accessibili
diretto	<i>12</i>
1 livello di indirezione (P1)	<i>1024 = 1 K pagine</i>
2 livelli di indirezione (P2)	<i>1024 × 1024 = 1 M pagine</i>
3 livelli di indirezione (P3)	<i>1024 × 1024 × 1024 = 1 G pagine</i>

- b) Si supponga che un programma esegua in sequenza le seguenti operazioni su un file F:

1. *open* (la open non legge il file, ma solo lo i-node)
2. *seek* (FP) – si posiziona all'inizio della pagina del file di numero FP, cioè all'inizio di DATI (FP), usando, quando necessario, il numero adeguato di blocchi puntatore
3. *read* (NUM) – NUM è il numero di pagine del file che vengono lette

Indicare la sequenza di blocchi dati e di blocchi puntatore trasferiti da disco in memoria, e il numero totale di blocchi (dati + puntatore) trasferiti, nei casi seguenti. Il primo caso è già compilato come esempio.

FP = 4 NUM = 2	FP = 11 NUM = 2	FP = 1035 NUM = 2	FP = 1035 NUM = 3
DATI (4)	<i>DATI (11)</i>	<i>P1</i>	<i>P1</i>
DATI (5)	<i>P1</i>	<i>DATI (1035)</i>	<i>DATI (1035)</i>
	<i>DATI (12)</i>	<i>P2</i>	<i>P2</i>
		<i>P2 (0)</i>	<i>P2 (0)</i>
		<i>DATI (1036)</i>	<i>DATI (1036)</i>
			<i>DATI (1037)</i>
Numero Totale di trasferimenti da disco nei diversi casi			
<i>2</i>	<i>3</i>	<i>5</i>	<i>6</i>