POLITECNICO DI MILANO

Esercizi
Logica Digitale

Alessandro A. Nacci
alessandro.nacci@polimi.it

ACSO
2014/2014


START

Cosa facciamo

- **Oggi, Mercoledì 19 Novembre** _ *A. Nacci*
  - File System e Prestazioni I/U
  - Thread e Parallelsimo

- **Domani, Giovedì 20 Novembre** _ *D. Sciuto*
  - Memoria Virtuale
  - Processi
  - …

FAQ su Memoria Virtuale

È lecito pensare (dal punto di vista degli esercizi) che la memoria fisica e la MMU abbiano la stessa dimensione?

No, ne' dal punto di vista reale ne' dal punto di vista degli esercizi, le due dimensioni sono indipendenti.

Il numero di pagine residenti viene contato in base alle pagine allocate in memoria fisica oppure a quelle in MMU?

In base a quelle della MMU, dove per ogni processo va inserita una riga per ogni pagina, anche se e' condivisa.

Dopo una fork, le pagine che il processo padre condivide con il figlio vanno anche contate come pagine residenti del figlio?

Si

Quando un processo P accede ad una pagina condivisa già caricata in memoria da un processo R (tra i quali non esiste legame di "parentela"), questa pagina viene contata come pagina residente del processo R?

Questa pagina va inserita nella MMU per il processo R e quindi conta come pagina residente

Es0: Performance

Una stampante opera in **DMA** con blocchi da **512 byte**. Il suo **driver** possiede un buffer di **10 K byte**. Un processo richiede la stampa di **32 K byte** sulla periferica.

Si risponda alle domande seguenti:

1. Quante interrupt vanno gestite nel corso dell'esecuzione dell'intera operazione ?
2. Per quante volte viene risvegliato il processo nel corso dell'intera operazione ?

Es0: Performance

Una stampante opera in **DMA** con blocchi da **512 byte**. Il suo **driver** possiede un buffer di **10 K byte**. Un processo richiede la stampa di **32 K byte** sulla periferica.

Si risponda alle domande seguenti:

1. Quante interrupt vanno gestite nel corso dell'esecuzione dell'intera operazione ?
2. Per quante volte viene risvegliato il processo nel corso dell'intera operazione ?

Quante interrupt vanno gestite nel corso dell'esecuzione dell'intera operazione ?

32 K byte / 512 byte = 64 interrupt

Es0: Performance

Una stampante opera in **DMA** con blocchi da **512 byte**. Il suo **driver** possiede un buffer di **10 K byte**. Un processo richiede la stampa di **32 K byte** sulla periferica.

Si risponda alle domande seguenti:

1. Quante interrupt vanno gestite nel corso dell'esecuzione dell'intera operazione ?
2. Per quante volte viene risvegliato il processo nel corso dell'intera operazione ?

Quante interrupt vanno gestite nel corso dell'esecuzione dell'intera operazione ?

32 K byte / 512 byte = 64 interrupt

Per quante volte viene risvegliato il processo nel corso dell'intera operazione ?

⌈32 K byte / 10 K byte ⌉ = ⌈3,2 ⌉ volte = 4 volte

Es1.a: File System

```
/* vari #include omessi */
int main()
{
  int   fd, fd2;
  pid_t pid;
  int   status;
  int   i;

  close(0);
  fd = open("/tmp/datafile", O_RDONLY);
  printf("padre:  pid = %i\n", getpid());
  fd2 = dup(fd);
  pid = fork();
  if (pid==0) {
    char     buffer    [4096];
    unsigned checksum  = 0;
    unsigned chunk_size;

    printf("figlio: pid = %i\n", getpid());
    while(chunk_size = read(fd2, buffer, 4096)) {
      for (i=0; i < chunk_size; i++)
         checksum += buffer[i];
    }
    printf("checksum = %u\n", checksum);

    sleep(100);

    close(fd2);
    exit(EXIT_SUCCESS);
  }

  for (i=0; i<3;i++) printf("padre:  cur = %u\n", lseek(fd,0,SEEK_CUR));
  wait(&status);
  return EXIT_SUCCESS;
}
```

Il programma calcola la checksum (cioè la somma dei valori di tutti i byte) del file "/tmp/datafile", della dimensione esatta di 7 MB. Lanciando in esecuzione il programma, si ottengono i seguenti messaggi:

```
padre:  pid = 12998
figlio: pid = 12999
padre:  cur = 90112
padre:  cur = 4448256
padre:  cur = 7340032
checksum = 120193024
```

Dopo aver emesso i messaggi seguenti, il figlio è bloccato nel corpo della chiamata sleep(100), e il padre nella wait().

Completare la tabella globale dei file aperti e quelle dei file aperti dei processi 12998 e 12999 valide nell'istante considerato.

Es1.a: File System

```
/* vari #include omessi */
int main()
{
  int   fd, fd2;
  pid_t pid;
  int   status;
  int   i;

  close(0);
  fd = open("/tmp/datafile", O_RDONLY);
  printf("padre:  pid = %i\n", getpid());
  fd2 = dup(fd);
  pid = fork();
  if (pid==0) {
    char     buffer    [4096];
    unsigned checksum  = 0;
    unsigned chunk_size;

    printf("figlio: pid = %i\n", getpid());
    while(chunk_size = read(fd2, buffer, 4096)) {
      for (i=0; i < chunk_size; i++)
         checksum += buffer[i];
    }
    printf("checksum = %u\n", checksum);

    sleep(100);

    close(fd2);
    exit(EXIT_SUCCESS);
  }

  for (i=0; i<3;i++) printf("padre:  cur = %u\n", lseek(fd,0,SEEK_CUR));
  wait(&status);
  return EXIT_SUCCESS;
}
```

```
padre:  pid = 12998
figlio: pid = 12999
padre:  cur = 90112
padre:  cur = 4448256
padre:  cur = 7340032
checksum = 120193024
```

VOLUME

Tabella degli i-node

numero i-node	tipo	blocco
22	directory	310
23	directory	311
24	normal	312

Blocco 310 n° i-node — nome file / directory

8	etc
7	dev
14	lib
23	tmp
...	...

Blocco 311 n° i-node — nome file / directory

5	lock
24	datafile

Blocco 312

Es1.a: File System - *Soluzione*

```
/* vari #include omessi */
int main()
{
  int   fd, fd2;
  pid_t pid;
  int   status;
  int   i;

  close(0);
  fd = open("/tmp/datafile", O_RDONLY);
  printf("padre:  pid = %i\n", getpid());
  fd2 = dup(fd);
  pid = fork();
  if (pid==0) {
    char     buffer    [4096];
    unsigned checksum  = 0;
    unsigned chunk_size;

    printf("figlio: pid = %i\n", getpid());
    while(chunk_size = read(fd2, buffer, 4096)) {
      for (i=0; i < chunk_size; i++)
          checksum += buffer[i];
    }
    printf("checksum = %u\n", checksum);

    sleep(100);

    close(fd2);
    exit(EXIT_SUCCESS);
  }

  for (i=0; i<3;i++) printf("padre:  cur = %u\n", lseek(fd,0,SEEK_CUR));
  wait(&status);
  return EXIT_SUCCESS;
}
```

```
padre:  pid = 12998
figlio: pid = 12999
padre:  cur = 90112
padre:  cur = 4448256
padre:  cur = 7340032
checksum = 120193024
```

Tabella dei file aperti del processo 12998

0	***12***
1	***Stdout***
2	***Stderr***
3	***12***
4	
5	

Tabella dei file aperti del processo 12999

0	***12***
1	***Stdout***
2	***Stderr***
3	***12***
4	
5	
6	

Tabella globale dei file aperti

indice	Pos.Corr.	Cont. Usi	Punt
10	**xxx**	**xxx**	**xxx**
11	**xxx**	**xxx**	**xxx**
12	***7340032***	***4***	***24***
13			
14			
15			
16			
17			
18			
19			

Es1.b: File System

Le soluzioni sono online

Si consideri la situazione descritta nel caso precedente: il processo figlio richiede la lettura di 7MB a segmenti di 4 kB. Calcolate il tempo T durante il quale il processore è occupato dal sistema operativo per svolgere tale trasferimento, per conto del processo figlio.
Dovrete calcolare T nei 4 diversi casi elencati sotto. Considerate sempre valide le seguenti ipotesi:

- devono essere letti solo i blocchi dati del file (l'i-node e i blocchi di puntatori sono già in memoria);
- la lunghezza di blocco è 512 byte;

caso 1) Il file risiede su un disco gestito in DMA; il canale DMA trasferisce solo un blocco alla volta. La routine di servizio all'interrupt che lancia una nuova operazione di lettura oppure risveglia il processo P quando è terminata l'intera operazione impiega 100 μs. Indicare il valore di T (in ms) esplicitando il calcolo eseguito.

T=

caso 2) Il file risiede su un disco gestito in DMA; il canale DMA trasferisce fino a 16kbyte in un unico trasferimento; il file è allocato in modo che i blocchi siano contigui sul disco. La routine di interrupt che lancia una nuova operazione di lettura oppure risveglia il processo P quando è terminata l'intera operazione impiega 100 μs. Indicare il valore di T (in ms) esplicitando il calcolo eseguito.

T= μs

caso 3) Il file risiede su un disco gestito in DMA; il canale DMA trasferisce fino a 16kbyte in un unico trasferimento; il file è allocato in modo che i blocchi siano contigui sul disco a 6 alla volta. La routine di interrupt che lancia una nuova operazione di lettura oppure risveglia il processo P quando è terminata l'intera operazione impiega 100 μs.

T=μs

caso 4) Il file risiede su un disco privo di DMA (ad esempio, un floppy disk). La routine di interrupt che lancia una nuova operazione di lettura oppure risveglia il processo P quando è terminata l'intera operazione impiega 10 μs. Indicare il valore di T esplicitando il calcolo eseguito.

T=

Es1.c: File System

Le soluzioni sono online

Si consideri la situazione: un processo P ha richiesto i due servizi seguenti, numerati (1) e (2). I servizi vengono attivati in successione, ma sono separati da altre istruzioni C, non rilevanti.

```
(1)   fd = open("/tmp/datafile", O_RDONLY); /* il file testo esiste */
(2)   read(fd2, buffer, 4096);
```

Per ciascuno dei due servizi sopra indicati, calcolate il tempo effettivo che il processore dedica all'esecuzione del servizio.

Valgono le seguente ipotesi sul comportamento del sistema operativo:

- la lista degli i-node viene tenuta sempre interamente anche in memoria,
- il contenuto del catalogo radice `(/)` viene tenuto costantemente in memoria;
- ogniqualvolta si deve effettuare una ricerca all'interno di un catalogo, il contenuto del catalogo viene copiato per intero da disco a memoria; terminata l'operazione, la copia in memoria viene subito eliminata;
- quando un file viene aperto in modo `O_RDONLY`, la posizione corrente è fissata all'inizio del file;
- quando un file ordinario viene aperto, il suo contenuto resta, per il momento, sul disco; se necessario i blocchi del file interessati delle varie operazioni verranno trasferiti da disco a memoria o viceversa, quando si dovrà leggere il file o scriverlo, rispettivamente.

Cataloghi e file ordinari risiedono su un disco gestito in DMA, e il canale di DMA è in grado di trasferire solamente un blocco alla volta.

In particolare, si considerino valide le ipotesi seguenti:

- la lunghezza di blocco è di: 2 kiloByte (2 kB)
- la dimensione del catalogo "`tmp`" è di: 20 kiloByte (20 kB)
- la dimensione del file ordinario "`datafile`" è di: 7 MegaByte (7 MB)
- il servizio di una interruzione (interrupt) per il trasferimento di un blocco da disco a memoria impiega una durata costante di 200 μs;
- la ricerca di un riferimento (nome simbolico, i-node) in un catalogo in memoria impiega una durata costante di 2.5 ms;
- il trasferimento del contenuto di una struttura dati utente del processo ai buffer di sistema o viceversa ha una durata proporzionale alle dimensioni della struttura dati, calcolabile tramite la formula: 30 ns $\times$ numero di byte da trasferire.

Si deve indicare per ciascuno dei due tempi richiesti lo svolgimento analitico dei calcoli; non si accettano valori privi di spiegazione.

Es2: Threads

Si consideri la bozza di sistema concorrente seguente (in linguaggio C), con due thread per i quali sono già previsti un mutex (`boh`), e due semafori (`mah_X` e `mah_Y`) inizializzati a zero.

I due thread vanno completati (secondo le modalità spiegate qui di seguito e tenendo conto della tabella mostrata a pagina seguente) dove ci sono righe lasciate appositamente vuote. In una riga vuota si può aggiungere **una sola** chiamata di sistema: o relativa al mutex (`lock` oppure `unlock`), o relativa a uno dei due semafori (`post` oppure `wait`); la riga (o eventualmente un intero gruppo di righe consecutive) può anche restare inutilizzata (o inutilizzato). Il numero di righe vuote **non è significativo**.

```
pthread_mutex_t boh;
sem_t mah_X, mah_Y;
int globale = 0;

void * uno (void * arg) {
    sem_wait _____
    pthread_mutex_lock _____
    globale = 1;                    /* statement A */
    sem_post _____

    _____
    pthread_mutex_unlock _____
    return NULL;
} /* end uno */

void * due (void * arg) {
    pthread_mutex_lock _____        /* statement B */
    sem_post _____

    _____
    globale = 2;                    /* statement C */
    pthread_mutex_unlock _____
    sem_wait _____
    return NULL;
} /* end due */

void main ( ) {
    pthread_t th_1, th_2;
    sem_init (&mah_X, 0, 0);
    sem_init (&mah_Y, 0, 0);
    pthread_create (&th_1, NULL, uno, NULL);
    pthread_create (&th_2, NULL, due, NULL);
    pthread_join (th_1, NULL);
    pthread_join (th_2, NULL);
return;
} /* end main */
```

Es2: Threads

Si consideri la bozza di sistema concorrente seguente (in linguaggio C), con due thread per i quali sono già previsti un mutex (`boh`), e due semafori (`mah_X` e `mah_Y`) inizializzati a zero.

I due thread vanno completati (secondo le modalità spiegate qui di seguito e tenendo conto della tabella mostrata a pagina seguente) dove ci sono righe lasciate appositamente vuote. In una riga vuota si può aggiungere **una sola** chiamata di sistema: o relativa al mutex (`lock` oppure `unlock`), o relativa a uno dei due semafori (`post` oppure `wait`); la riga (o eventualmente un intero gruppo di righe consecutive) può anche restare inutilizzata (o inutilizzato). Il numero di righe vuote **non è significativo**.

```
pthread_mutex_t boh;
sem_t mah_X, mah_Y;
int globale = 0;

void * uno (void * arg) {
    sem_wait (&mah_X) _____________________
    pthread_mutex_lock _____________________
    globale = 1;                    /* statement A */
    sem_post _____________________

    _____________________________________________
    pthread_mutex_unlock _____________________
    return NULL;
} /* end uno */

void * due (void * arg) {
    pthread_mutex_lock _____  * statement B */
    sem_post _____________________

    _____________________________________________
    globale = 2;                    /* statement C */
    pthread_mutex_unlock _____________________
    sem_wait _____________________
    return NULL;
} /* end due */

void main ( ) {
    pthread_t th_1, th_2;
    sem_init (&mah_X, 0, 0);
    sem_init (&mah_Y, 0, 0);
    pthread_create (&th_1, NULL, uno, NULL);
    pthread_create (&th_2, NULL, due, NULL);
    pthread_join (th_1, NULL);
    pthread_join (th_2, NULL);
return;
} /* end main */
```

Es2: Threads

Si consideri la bozza di sistema concorrente seguente (in linguaggio C), con due thread per i quali sono già previsti un mutex (`boh`), e due semafori (`mah_X` e `mah_Y`) inizializzati a zero.

I due thread vanno completati (secondo le modalità spiegate qui di seguito e tenendo conto della tabella mostrata a pagina seguente) dove ci sono righe lasciate appositamente vuote. In una riga vuota si può aggiungere **una sola** chiamata di sistema: o relativa al mutex (`lock` oppure `unlock`), o relativa a uno dei due semafori (`post` oppure `wait`); la riga (o eventualmente un intero gruppo di righe consecutive) può anche restare inutilizzata (o inutilizzato). Il numero di righe vuote **non è significativo**.

```
pthread_mutex_t boh;
sem_t mah_X, mah_Y;
int globale = 0;

void * uno (void * arg) {
    sem_wait (&mah_X) _____
    pthread_mutex_lock (&boh) _____

    globale = 1;                    /* statement A */

    sem_post _____

    _____
    pthread_mutex_unlock _____
    return NULL;
} /* end uno */

void * due (void * arg) {
    pthread_mutex_lock _____ * statement B */

    sem_post _____

    _____
    globale = 2;                    /* statement C */

    pthread_mutex_unlock _____
    sem_wait _____
    return NULL;
} /* end due */

void main ( ) {
    pthread_t th_1, th_2;
    sem_init (&mah_X, 0, 0);
    sem_init (&mah_Y, 0, 0);
    pthread_create (&th_1, NULL, uno, NULL);
    pthread_create (&th_2, NULL, due, NULL);
    pthread_join (th_1, NULL);
    pthread_join (th_2, NULL);
return;
} /* end main */
```

Es2: Threads

Si consideri la bozza di sistema concorrente seguente (in linguaggio C), con due thread per i quali sono già previsti un mutex (`boh`), e due semafori (`mah_X` e `mah_Y`) inizializzati a zero.

I due thread vanno completati (secondo le modalità spiegate qui di seguito e tenendo conto della tabella mostrata a pagina seguente) dove ci sono righe lasciate appositamente vuote. In una riga vuota si può aggiungere **una sola** chiamata di sistema: o relativa al mutex (`lock` oppure `unlock`), o relativa a uno dei due semafori (`post` oppure `wait`); la riga (o eventualmente un intero gruppo di righe consecutive) può anche restare inutilizzata (o inutilizzato). Il numero di righe vuote **non è significativo**.

```
pthread_mutex_t boh;
sem_t mah_X, mah_Y;
int globale = 0;

void * uno (void * arg) {
   sem_wait (&mah_X) _____________________
   pthread_mutex_lock (&boh) _____________
   globale = 1;                  /* statement A */
   sem_post (&mah_Y) _____________________

   _____________________________________
   pthread_mutex_unlock _________________
   return NULL;
} /* end uno */

void * due (void * arg) {
   pthread_mutex_lock _____ * statement B */
   sem_post _____________
   _____________________________________
   globale = 2;                  /* statement C */
   pthread_mutex_unlock _________________
   sem_wait _____________
   return NULL;
} /* end due */

void main ( ) {
   pthread_t th_1, th_2;
   sem_init (&mah_X, 0, 0);
   sem_init (&mah_Y, 0, 0);
   pthread_create (&th_1, NULL, uno, NULL);
   pthread_create (&th_2, NULL, due, NULL);
   pthread_join (th_1, NULL);
   pthread_join (th_2, NULL);
return;
} /* end main */
```

Es2: Threads

Si consideri la bozza di sistema concorrente seguente (in linguaggio C), con due thread per i quali sono già previsti un mutex (`boh`), e due semafori (`mah_X` e `mah_Y`) inizializzati a zero.

I due thread vanno completati (secondo le modalità spiegate qui di seguito e tenendo conto della tabella mostrata a pagina seguente) dove ci sono righe lasciate appositamente vuote. In una riga vuota si può aggiungere **una sola** chiamata di sistema: o relativa al mutex (`lock` oppure `unlock`), o relativa a uno dei due semafori (`post` oppure `wait`); la riga (o eventualmente un intero gruppo di righe consecutive) può anche restare inutilizzata (o inutilizzato). Il numero di righe vuote **non è significativo**.

```
pthread_mutex_t boh;
sem_t mah_X, mah_Y;
int globale = 0;

void * uno (void * arg) {
   sem_wait (&mah_X) _____
   pthread_mutex_lock (&boh) _____
   globale = 1;                    /* statement A */
   sem_post (&mah_Y) _____
   _____
   pthread_mutex_unlock (&boh)
   return NULL;
} /* end uno */

void * due (void * arg) {
   pthread_mutex_lock _____ * statement B */
   sem_post _____
   _____
   globale = 2;                    /* statement C */
   pthread_mutex_unlock _____
   sem_wait _____
   return NULL;
} /* end due */

void main ( ) {
   pthread_t th_1, th_2;
   sem_init (&mah_X, 0, 0);
   sem_init (&mah_Y, 0, 0);
   pthread_create (&th_1, NULL, uno, NULL);
   pthread_create (&th_2, NULL, due, NULL);
   pthread_join (th_1, NULL);
   pthread_join (th_2, NULL);
return;
} /* end main */
```

Es2: Threads

Si consideri la bozza di sistema concorrente seguente (in linguaggio C), con due thread per i quali sono già previsti un mutex (`boh`), e due semafori (`mah_X` e `mah_Y`) inizializzati a zero.

I due thread vanno completati (secondo le modalità spiegate qui di seguito e tenendo conto della tabella mostrata a pagina seguente) dove ci sono righe lasciate appositamente vuote. In una riga vuota si può aggiungere **una sola** chiamata di sistema: o relativa al mutex (`lock` oppure `unlock`), o relativa a uno dei due semafori (`post` oppure `wait`); la riga (o eventualmente un intero gruppo di righe consecutive) può anche restare inutilizzata (o inutilizzato). Il numero di righe vuote **non è significativo**.

```
pthread_mutex_t boh;
sem_t mah_X, mah_Y;
int globale = 0;

void * uno (void * arg) {
   sem_wait (&mah_X) _____________________
   pthread_mutex_lock (&boh) _____________
   globale = 1;                  /* statement A */
   sem_post (&mah_Y) _____________________
   _____________________________________
   pthread_mutex_unlock (&boh)
   return NULL;
} /* end uno */

void * due (void * arg) {
   pthread_mutex_lock (&boh)    /* statement B */
   sem_post _____________________
   _____________________________________
   globale = 2;                  /* statement C */
   pthread_mutex_unlock _________________
   sem_wait _____________________
   return NULL;
} /* end due */

void main ( ) {
   pthread_t th_1, th_2;
   sem_init (&mah_X, 0, 0);
   sem_init (&mah_Y, 0, 0);
   pthread_create (&th_1, NULL, uno, NULL);
   pthread_create (&th_2, NULL, due, NULL);
   pthread_join (th_1, NULL);
   pthread_join (th_2, NULL);
return;
} /* end main */
```

Es2: Threads

Si consideri la bozza di sistema concorrente seguente (in linguaggio C), con due thread per i quali sono già previsti un mutex (`boh`), e due semafori (`mah_X` e `mah_Y`) inizializzati a zero.

I due thread vanno completati (secondo le modalità spiegate qui di seguito e tenendo conto della tabella mostrata a pagina seguente) dove ci sono righe lasciate appositamente vuote. In una riga vuota si può aggiungere **una sola** chiamata di sistema: o relativa al mutex (`lock` oppure `unlock`), o relativa a uno dei due semafori (`post` oppure `wait`); la riga (o eventualmente un intero gruppo di righe consecutive) può anche restare inutilizzata (o inutilizzato). Il numero di righe vuote **non è significativo**.

```
pthread_mutex_t boh;
sem_t mah_X, mah_Y;
int globale = 0;

void * uno (void * arg) {
   sem_wait (&mah_X) _____________________
   pthread_mutex_lock (&boh) _____________
   globale = 1;                  /* statement A */
   sem_post (&mah_Y) _____________________
   _____________________________________
   pthread_mutex_unlock (&boh)
   return NULL;
} /* end uno */

void * due (void * arg) {
   pthread_mutex_lock (&boh)    /* statement B */
   sem_post (&mah_X) _____________________
   _____________________________________
   globale = 2;                  /* statement C */
   pthread_mutex_unlock _________________
   sem_wait _____________________
   return NULL;
} /* end due */

void main ( ) {
   pthread_t th_1, th_2;
   sem_init (&mah_X, 0, 0);
   sem_init (&mah_Y, 0, 0);
   pthread_create (&th_1, NULL, uno, NULL);
   pthread_create (&th_2, NULL, due, NULL);
   pthread_join (th_1, NULL);
   pthread_join (th_2, NULL);
return;
} /* end main */
```

Es2: Threads

Si consideri la bozza di sistema concorrente seguente (in linguaggio C), con due thread per i quali sono già previsti un mutex (`boh`), e due semafori (`mah_X` e `mah_Y`) inizializzati a zero.

I due thread vanno completati (secondo le modalità spiegate qui di seguito e tenendo conto della tabella mostrata a pagina seguente) dove ci sono righe lasciate appositamente vuote. In una riga vuota si può aggiungere **una sola** chiamata di sistema: o relativa al mutex (`lock` oppure `unlock`), o relativa a uno dei due semafori (`post` oppure `wait`); la riga (o eventualmente un intero gruppo di righe consecutive) può anche restare inutilizzata (o inutilizzato). Il numero di righe vuote **non è significativo**.

```
pthread_mutex_t boh;
sem_t mah_X, mah_Y;
int globale = 0;

void * uno (void * arg) {
   sem_wait (&mah_X) _____________________
   pthread_mutex_lock (&boh) _____________
   globale = 1;                  /* statement A */
   sem_post (&mah_Y) _____________________
   _____________________________________
   pthread_mutex_unlock (&boh)
   return NULL;
} /* end uno */

void * due (void * arg) {
   pthread_mutex_lock (&boh)    /* statement B */
   sem_post (&mah_X) _____________________
   _____________________________________
   globale = 2;                  /* statement C */
   pthread_mutex_unlock (&boh) ___________
   sem_wait ____________________
   return NULL;
} /* end due */

void main ( ) {
   pthread_t th_1, th_2;
   sem_init (&mah_X, 0, 0);
   sem_init (&mah_Y, 0, 0);
   pthread_create (&th_1, NULL, uno, NULL);
   pthread_create (&th_2, NULL, due, NULL);
   pthread_join (th_1, NULL);
   pthread_join (th_2, NULL);
return;
} /* end main */
```

Es2: Threads

Si consideri la bozza di sistema concorrente seguente (in linguaggio C), con due thread per i quali sono già previsti un mutex (`boh`), e due semafori (`mah_X` e `mah_Y`) inizializzati a zero.

I due thread vanno completati (secondo le modalità spiegate qui di seguito e tenendo conto della tabella mostrata a pagina seguente) dove ci sono righe lasciate appositamente vuote. In una riga vuota si può aggiungere **una sola** chiamata di sistema: o relativa al mutex (`lock` oppure `unlock`), o relativa a uno dei due semafori (`post` oppure `wait`); la riga (o eventualmente un intero gruppo di righe consecutive) può anche restare inutilizzata (o inutilizzato). Il numero di righe vuote **non è significativo**.

```
pthread_mutex_t boh;
sem_t mah_X, mah_Y;
int globale = 0;

void * uno (void * arg) {
   sem_wait (&mah_X) _____________________
   pthread_mutex_lock (&boh) _____________
   globale = 1;                  /* statement A */
   sem_post (&mah_Y) _____________________
   _______________________________________
   pthread_mutex_unlock (&boh)
   return NULL;
} /* end uno */

void * due (void * arg) {
   pthread_mutex_lock (&boh)    /* statement B */
   sem_post (&mah_X) _____________________
   _______________________________________
   globale = 2;                  /* statement C */
   pthread_mutex_unlock (&boh) _____________
   sem_wait (&mah_Y) _____________________
   return NULL;
} /* end due */

void main ( ) {
   pthread_t th_1, th_2;
   sem_init (&mah_X, 0, 0);
   sem_init (&mah_Y, 0, 0);
   pthread_create (&th_1, NULL, uno, NULL);
   pthread_create (&th_2, NULL, due, NULL);
   pthread_join (th_1, NULL);
   pthread_join (th_2, NULL);
return;
} /* end main */
```

Es2: Threads - *Soluzione*

Un modo ordinato per risolvere questo esercizio consiste nell'esaminare le condizioni A, B e C, individuare i problemi di concorrenza che esse sottendono, e per ciascun problema individuato scegliere la soluzione "standard" o comunque ovvia che la programmazione concorrente mette a disposizione, compatibilmente con quanto è già scritto nel codice (dove figurano alcune chiamate di sistema). Conviene dare uno sguardo d'insieme a tutte e tre le condizioni prima di attaccarne una specifica. Si ricordi che l'ordine in cui esse figurano nella tabella dall'alto verso il basso, non implica affatto che esse siano ordinate temporalmente nello stesso modo; anzi il loro ordine temporale potrebbe non essere deterministico.

Esaminando rapidamente le tre condizioni A, B e C, si nota subito che la C implica che il thread 2 operi senza interferenza da parte del thread 1, ossia (ragionevolmente) in mutua esclusione; poiché nel codice sono già presenti chiamate di sistema per il mutex, ma lo schema mutex è incompleto, pare sensato cercare di soddisfare per prima la C (qualunque sia il suo ordine temporale rispetto ad A e B, se pure un ordine temporale deterministico esiste) completando in qualche modo lo schema abbozzato; e poi soddisfare le altre due condizioni, cominciando dalla B, la quale palesemente si riferisce a un valore iniziale, e pertanto concludendo con la A, la quale oltretutto è formulata in modo un po' diverso ossia come stato di esistenza. Ora si può passare a un esame approfondito delle tre condizioni, nell'ordine C, B e A.

Per la condizione C, si vuole tenere fissa a valore 2 la variabile globale subito dopo che il thread 2 la ha assegnata (ossia nell'intervallo di tempo tra l'assegnamento a 2 e lo statement successivo). Pertanto, dato che entrambi i thread modificano globale, gli assegnamenti a globale devono essere mutuamente esclusivi. In altri termini, ciascuno dei due thread deve avere una sequenza critica che contenga il rispettivo assegnamento a globale (o almeno questa è la soluzione più semplice e immediata, rispetto a usare i semafori, e comunque è già parzialmente data nel codice). Ciò implicherà di posizionare la lock del thread 1 a monte dell'assegnamento a globale, e la unlock del thread 2 a valle dell'assegnamento a globale. In linea di massima ciò definisce il posizionamento approssimativo della lock e della unlock evidentemente mancanti. Come esattamente posizionare tale lock e tale unlock relativamente alle post e wait che andranno (presumibilmente) collocate per soddisfare le specifiche rimanenti, è determinato dalle altre due condizioni.

Per la condizione B (la più facile), si vuole tenere fissa a valore iniziale 0 la variabile globale subito dopo che il thread 2 ha superata la lock (e ovviamente prima di avere assegnato valore 2 a globale), ossia prima che il thread 1 possa assegnare valore 1 a globale. Insomma il thread 2 deve eseguire la sequenza critica di assegnamento a globale prima del thread 1. È un problema di ordinamento e si risolve tramite un semaforo. Pertanto il thread 1 deve effettuare una wait su un semaforo, p. es. mah_X, a monte della lock da collocare. Di conseguenza alla wait su mah_X, il thread 2 deve effettuare una post su mah_X, a valle della lock già data nel codice (come posizionare tale post rispetto all'assegnamento a globale è un dettaglio minore, senza conseguenze, e viene indirettamente deciso dalla condizione A – vedi sotto).

Per la condizione A, si vuole garantire che il thread 2 esista (ancora) mentre il thread 1 è in sequenza critica. È di nuovo un problema di ordinamento e si risolve tramite un semaforo. Tuttavia non si può riusare il semaforo mah_X: per due thread che si bloccano-sbloccano ordinatamente a vicenda, occorrono due semafori indipendenti. Pertanto il thread 2 deve effettuare una wait sull'altro semaforo, mah_Y, a valle della unlock da collocare (il sistema non deve avere stalli – se la wait fosse a monte della unlock ci sarebbe stallo). Di conseguenza alla wait su mah_Y, il thread 1 deve effettuare una post su mah_Y.

Dettagli: la post su mah_Y nel thread 1 potrebbe stare a monte o a valle dell'assegnamento a globale, ma per i vincoli di spazio (numero di righe vuote) deve stare a valle, e potrebbe stare a monte o a valle della unlock già data nel codice, ma per i vincoli di spazio deve stare a monte; e indirettamente, sempre per i vincoli di spazio, la post su mah_X nel thread 2 va collocata prima dell'assegnamento a globale (ciò sistema il dettaglio lasciato irrisolto esaminando la condizione B).

Con le considerazioni svolte sopra, la struttura del sistema concorrente risulta (ragionevolmente) determinata univocamente. Ovviamente i due semafori sono interscambiabili, ma questo è un aspetto banale. Si osservi che le tre condizioni A, B e C determinano globalmente la struttura del sistema concorrente, e che nessuna di esse, presa isolatamente, determina univocamente dove esattamente collocare una certa chiamata di sistema all'interno del codice dato e relativamente alle altre chiamate che vanno pure collocate. Beninteso potrebbero esistere linee di ragionamento differenti da quella qui proposta.

Es3: File System

```
01.  int main ( ) { /* processo P          */
02.     int pid;
03.     int fd_1, fd_2;
04.     /* dichiarazioni varie            */
05.     fd_1 = open ("/cat1/file1", O_RDWR);
06.     read (fd1, buf, 4);
07.     pid = fork ( );
08.     if (pid == 0) { /* processo Q     */
09.        fd_2 = open ("/cat1/cat2/file2", O_ RDWR);
10.        read (fd_2, buf, 2);
11.        pid = fork ( );
12.        if (pid == 0) { /* processo R  */
13.           read (fd_2, buf, 2);
14.           write (fd_2, "ciao", 4);
15.           exit (0);
16.        } /* fine R                    */
17.        pid = wait (NULL);
18.        read (fd_2, buf, 2);
19.        exit (0);
20.     } /* fine Q                       */
21.     write (fd_1, "salve", 4);
22.     exit (0);
23.  } /* fine P */
```

I-Lista:	< 0, dir, 10 > < 1, dir, 100 > < 5, dir, 30 > < 10, dir, 20 > < 40, norm, 70 > < 50, norm, 80 >
Blocco 10:	… < 1, dev > … < 5, cat1> …
Blocco 20:	… < 40, file2 > …
Blocco 30:	… < 10, cat2 > … < 50, file1 > …
Blocco 70:	BUONGIORNO
Blocco 80:	BUONANOTTE

Un processo P esegue il programma seguente, creando un processo figlio Q, che crea a sua volta un figlio R:

Il pid del processo P è 500, poi il S.O. prosegue assegnando pid consecutivi ai processi via via creati. Il primo file descriptor libero è 3, e a ogni apertura il file system procede usando file descriptor in sequenza.

A un certo istante di tempo T1 il processo P è arrivato all'invocazione della funzione exit (che non è stata ancora eseguita).
A un certo istante di tempo successivo T2 T1, il processo Q è arrivato all'invocazione della funzione exit (che non è stata ancora eseguita) e il processo P non ha ancora eseguito la funzione exit (dove era già arrivato al tempo T1).
Il contenuto del volume durante è mostrato.

Nota bene: lo i-node associato al catalogo radice "/" ha 0 come i-number; la i-lista contiene terne < i-number, tipo_file, indice_blocco >; i cataloghi contengono coppie < i-number, nome_file >.

Es4: File System

Su un calcolatore dotato di sistema operativo Linux viene eseguito il programma seguente, che parte come processo **P** e crea il processo **Q**.

NOTA: si ricorda che in **lseek** (fd, offset, riferimento), riferimento = 0 indica inizio file, riferimento = 1 indica posizione corrente e riferimento = 2 indica fine file.

`int main ( ) { /* processo P */`	**sequenza**
`...`	
`fd1 = open (“/acso/esame/file1”, O_RDWR);`	**1**
`read (fd1, buf1, 10);`	**2**
`write (fd1, buf2, 600);`	**3**
`pid1 = fork ()`	**4**
`if (pid1 == 0) { /* processo Q */`	
`fd2 = open (“/acso/esame/file2”, O_RDONLY);`	**5**
`lseek (fd2, 1024, 1);`	**6**
`read (fd1, buf1, 100);`	**9**
`close (fd1)`	**10**
`exit (0);`	**11**
`} else { /* processo P */`	
`read (fd1, buf1, 10);`	**7**
`close (fd1);`	**8**
`} /* if */`	
`exit (0);`	**12**
`} /* main */`	

Durante l'esecuzione è stata osservata la **sequenza di chiamate di sistema indicata nella colonna di destra**.

Si completi il contenuto delle tabelle seguenti, indicando la sequenza dei valori assunti fino alla conclusione della chiamata di sistema numero **12** (si scriva **L** oppure **NE** per indicare che una cella si è liberata o non esiste più). Per la determinazione degli I-node, si veda il contenuto del volume a pagina seguente:

Al momento dell'esecuzione, il contenuto del volume è il seguente:

I-Lista: < 0, dir, 4 > < 6, dir, 20 > < 7, dir, 31 > < 98, norm, { 800, 801, 802, 803, 804, … } >
< 110, norm, { 1718, 1719, … } > < 232, norm, { 2733, 2734, … } >
blocco 4: … < 6, acso > …
blocco 20: … < 7, esame > …
blocco 31: … < 98, file1 > < 110, file2 > < 232, file3 > …

Es4: File System *Soluzione*

tab. file aperti del proc. **P**	
file des.	rif. riga
0	X
1	X
2	X
3	*2, L, NE*
4	
5	
6	

tab. file aperti del proc. **Q**	
file des.	rif. riga
0	X
1	X
2	X
3	*2, L, NE*
4	*3, NE*
5	
6	

tabella globale dei file aperti			
riga	posizione corrente	n di processi	i-nodo
0	X	X	X
1	X	X	X
2	*0, 10, 610, 620, 720, L*	*1, 2, 1, L*	*98*
3	*0, 1024, L*	*1, L*	*110*
4			
5			
6			

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A				
condizione	**variabili locali al thread th_A**		**variabili globali**	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A				
subito dopo STATEMENT S2 in th_A				
subito dopo STATEMENT S3 in th_B				
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A				
condizione	**variabili locali al thread th_A**		**variabili globali**	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A				
subito dopo STATEMENT S2 in th_A				
subito dopo STATEMENT S3 in th_B				
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A				
condizione	**variabili locali al thread th_A**		**variabili globali**	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A				
subito dopo STATEMENT S2 in th_A				
subito dopo STATEMENT S3 in th_B				
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*			
subito dopo STATEMENT S2 in th_A				
subito dopo STATEMENT S3 in th_B				
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*	*X*		
subito dopo STATEMENT S2 in th_A				
subito dopo STATEMENT S3 in th_B				
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*	*X*	*4*	
subito dopo STATEMENT S2 in th_A				
subito dopo STATEMENT S3 in th_B				
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*	*X*	*4*	*0*
subito dopo STATEMENT S2 in th_A				
subito dopo STATEMENT S3 in th_B				
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*	*X*	*4*	*0*
subito dopo STATEMENT S2 in th_A	*2*			
subito dopo STATEMENT S3 in th_B				
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*	*X*	*4*	*0*
subito dopo STATEMENT S2 in th_A	*2*	*2*		
subito dopo STATEMENT S3 in th_B				
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*	*X*	*4*	*0*
subito dopo STATEMENT S2 in th_A	*2*	*2*	*4 / 6*	
subito dopo STATEMENT S3 in th_B				
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*	*X*	*4*	*0*
subito dopo STATEMENT S2 in th_A	*2*	*2*	*4 / 6*	*0*
subito dopo STATEMENT S3 in th_B				
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*	*X*	*4*	*0*
subito dopo STATEMENT S2 in th_A	*2*	*2*	*4 / 6*	*0*
subito dopo STATEMENT S3 in th_B	*U (NE / X / 2)*			
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*	*X*	*4*	*0*
subito dopo STATEMENT S2 in th_A	*2*	*2*	*4 / 6*	*0*
subito dopo STATEMENT S3 in th_B	*U (NE / X / 2)*	*U (NE / X / 2)*		
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*	*X*	*4*	*0*
subito dopo STATEMENT S2 in th_A	*2*	*2*	*4 / 6*	*0*
subito dopo STATEMENT S3 in th_B	*U (NE / X / 2)*	*U (NE / X / 2)*	*3 / 6*	
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*	*X*	*4*	*0*
subito dopo STATEMENT S2 in th_A	*2*	*2*	*4 / 6*	*0*
subito dopo STATEMENT S3 in th_B	*U (NE / X / 2)*	*U (NE / X / 2)*	*3 / 6*	*0 / 1*
subito dopo STATEMENT S4 in main				

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*	*X*	*4*	*0*
subito dopo STATEMENT S2 in th_A	*2*	*2*	*4 / 6*	*0*
subito dopo STATEMENT S3 in th_B	*U (NE / X / 2)*	*U (NE / X / 2)*	*3 / 6*	*0 / 1*
subito dopo STATEMENT S4 in main	*NE*			

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella A

condizione	variabili locali al thread th_A		variabili globali	
	Aa	**Ab**	**Ga**	**sem_A**
subito dopo STATEMENT S1 in th_A	*2*	*X*	*4*	*0*
subito dopo STATEMENT S2 in th_A	*2*	*2*	*4 / 6*	*0*
subito dopo STATEMENT S3 in th_B	*U (NE / X / 2)*	*U (NE / X / 2)*	*3 / 6*	*0 / 1*
subito dopo STATEMENT S4 in main	*NE*	*NE*		

Es5: Parallelismo

esercizio n. 1 – thread e parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella B

condizione	variabili locali al thread th_B
	Ba
subito dopo STATEMENT S1 in th_A	
subito dopo STATEMENT S2 in th_A	
subito dopo STATEMENT S3 in th_B	
subito dopo STATEMENT S4 in main	

Es5: Parallelismo

esercizio n. 1 – thread e parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella B

condizione	variabili locali al thread th_B
	Ba
subito dopo STATEMENT S1 in th_A	
subito dopo STATEMENT S2 in th_A	
subito dopo STATEMENT S3 in th_B	
subito dopo STATEMENT S4 in main	

Es5: Parallelismo

esercizio n. 1 – thread e parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella B

condizione	variabili locali al thread th_B
	Ba
subito dopo STATEMENT S1 in th_A	
subito dopo STATEMENT S2 in th_A	
subito dopo STATEMENT S3 in th_B	
subito dopo STATEMENT S4 in main	

Es5: Parallelismo

esercizio n. 1 – thread e parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella B

condizione	variabili locali al thread th_B
	Ba
subito dopo STATEMENT S1 in th_A	*U (NE / X / 3 / 9)*
subito dopo STATEMENT S2 in th_A	
subito dopo STATEMENT S3 in th_B	
subito dopo STATEMENT S4 in main	

Es5: Parallelismo

esercizio n. 1 – thread e parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella B

condizione	variabili locali al thread th_B
	Ba
subito dopo STATEMENT S1 in th_A	*U (NE / X / 3 / 9)*
subito dopo STATEMENT S2 in th_A	*U (NE / X / 3 / 9 / 18)*
subito dopo STATEMENT S3 in th_B	
subito dopo STATEMENT S4 in main	

Es5: Parallelismo

esercizio n. 1 – thread e parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella B

condizione	variabili locali al thread th_B
	Ba
subito dopo STATEMENT S1 in th_A	*U (NE / X / 3 / 9)*
subito dopo STATEMENT S2 in th_A	*U (NE / X / 3 / 9 / 18)*
subito dopo STATEMENT S3 in th_B	*9 / 18*
subito dopo STATEMENT S4 in main	

Es5: Parallelismo

esercizio n. 1 – thread e parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella B

condizione	variabili locali al thread th_B
	Ba
subito dopo STATEMENT S1 in th_A	*U (NE / X / 3 / 9)*
subito dopo STATEMENT S2 in th_A	*U (NE / X / 3 / 9 / 18)*
subito dopo STATEMENT S3 in th_B	*9 / 18*
subito dopo STATEMENT S4 in main	*U (NE / X / 3 / 9 / 18)*

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella C

condizione	variabili locali al thread main	variabili globali	
	Ma	Ga	sem_A
subito dopo STATEMENT S4 in main			

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella C

condizione	variabili locali al thread main	variabili globali	
	Ma	Ga	sem_A
subito dopo STATEMENT S4 in main			

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella C

condizione	variabili locali al thread main	variabili globali	
	Ma	Ga	sem_A
subito dopo STATEMENT S4 in main			

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella C

condizione	variabili locali al thread main	variabili globali	
	Ma	Ga	sem_A
subito dopo STATEMENT S4 in main			

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella C

condizione	variabili locali al thread main	variabili globali	
	Ma	Ga	sem_A
subito dopo STATEMENT S4 in main	*2*		

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella C

condizione	variabili locali al thread main	variabili globali	
	Ma	Ga	sem_A
subito dopo STATEMENT S4 in main	*2*	*U (3 / 4 / 6)*	

Es5: Parallelismo

Si completino le tabelle, indicando gli state delle variabili globali e locali. Lo stato di una variabile può essere:

- intero, carattere, stringa (quando il valore è definito)
- X, quando la variabile non è ancora stata inizializzata
- NE, se la variabile può non esistere
- se la variabile si può trovare in due o più stati, li si riporti tutti quanti
- U quando la variable può trovarsi in tre o più stati o in qualsiasi combinazione

Si presti attenzione alla colonna "condizione". In particolare, con "subito dopo statement X" si intendono richiedere i valori che le variabili possono assumere tra lo statement X e lo statement immediatamente successivo del thread indicato nella condizione stessa.

```
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
/* PTHREAD_MUTEX_INITIALIZER è equivalente a pthread_mutex_init */
sem_t sem_A;
int Ga = 1;
void * A (void * arg) {
     int Aa = (int) arg;
     int Ab;
     pthread_mutex_lock (&mutex_A);
     Ga = Aa * 2;                          /* STATEMENT S1 */
     Ab = Ga - 2;
     pthread_mutex_unlock (&mutex_A);      /* STATEMENT S2 */
     return (void *) Ab;
} /* thread A */
```

```
void * B (void * arg) {
     int Ba = (int) arg;
     pthread_mutex_lock (&mutex_A);
     Ga = Ga + 2;
     Ba = Ba * Ga;                         /* STATEMENT S3 */
     pthread_mutex_unlock (&mutex_A);
     sem_wait (&sem_A);
     Ga = 3;
     return NULL;
} /* thread B */
```

```
int main ( ) {
     pthread_t th_A, th_B;
     int Ma;
     sem_init (&sem_A, 0, 0);
     pthread_create (&th_A, NULL, A, (void *) 2);
     pthread_create (&th_B, NULL, B, (void *) 3);
     pthread_join (th_A, (void *) &Ma);
     sem_post (&sem_A);                    /* STATEMENT S4 */
     pthread_join (th_B, NULL);
     return Ma;
} /* thread principale */
```

Tabella C

condizione	variabili locali al thread main	variabili globali	
	Ma	Ga	sem_A
subito dopo STATEMENT S4 in main	*2*	*U (3 / 4 / 6)*	*0 / 1*

Es6: Parallelismo

Si completi la tabella predisposta qui sotto indicando lo stato di esistenza della variabile locale nell'istante di tempo specificato da ciascuna condizione, così: se la variabile certamente esiste, si scriva ESISTE; se certamente 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 la variabile assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**variabile locale**	
	gone in **TR1**	*gone* in **TR2**
subito dopo stat. **A**		
subito dopo stat. **C** in **TR1**		
subito dopo stat. **D**		

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Si completi la tabella predisposta qui sotto indicando lo stato di esistenza della variabile locale nell'istante di tempo specificato da ciascuna condizione, così: se la variabile certamente esiste, si scriva ESISTE; se certamente 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 la variabile assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**variabile locale**	
	gone in **TR1**	*gone* in **TR2**
subito dopo stat. **A**	*PUÒ ESISTERE*	
subito dopo stat. **C** in **TR1**		
subito dopo stat. **D**		

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Si completi la tabella predisposta qui sotto indicando lo stato di esistenza della variabile locale nell'istante di tempo specificato da ciascuna condizione, così: se la variabile certamente esiste, si scriva ESISTE; se certamente 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 la variabile assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**variabile locale**	
	gone in **TR1**	*gone* in **TR2**
subito dopo stat. **A**	*PUÒ ESISTERE*	*PUÒ ESISTERE*
subito dopo stat. **C** in **TR1**		
subito dopo stat. **D**		

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Si completi la tabella predisposta qui sotto indicando lo stato di esistenza della variabile locale nell'istante di tempo specificato da ciascuna condizione, così: se la variabile certamente esiste, si scriva ESISTE; se certamente 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 la variabile assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**variabile locale**	
	gone in **TR1**	*gone* in **TR2**
subito dopo stat. **A**	*PUÒ ESISTERE*	*PUÒ ESISTERE*
subito dopo stat. **C** in **TR1**	*ESISTE*	
subito dopo stat. **D**		

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Si completi la tabella predisposta qui sotto indicando lo stato di esistenza della variabile locale nell'istante di tempo specificato da ciascuna condizione, così: se la variabile certamente esiste, si scriva ESISTE; se certamente 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 la variabile assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**variabile locale**	
	gone in **TR1**	*gone* in **TR2**
subito dopo stat. **A**	*PUÒ ESISTERE*	*PUÒ ESISTERE*
subito dopo stat. **C** in **TR1**	*ESISTE*	*PUÒ ESISTERE*
subito dopo stat. **D**		

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Si completi la tabella predisposta qui sotto indicando lo stato di esistenza della variabile locale nell'istante di tempo specificato da ciascuna condizione, così: se la variabile certamente esiste, si scriva ESISTE; se certamente 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 la variabile assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**variabile locale**	
	gone in **TR1**	*gone* in **TR2**
subito dopo stat. **A**	*PUÒ ESISTERE*	*PUÒ ESISTERE*
subito dopo stat. **C** in **TR1**	*ESISTE*	*PUÒ ESISTERE*
subito dopo stat. **D**	*PUÒ ESISTERE*	

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Si completi la tabella predisposta qui sotto indicando lo stato di esistenza della variabile locale nell'istante di tempo specificato da ciascuna condizione, così: se la variabile certamente esiste, si scriva ESISTE; se certamente 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 la variabile assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**variabile locale**	
	gone in **TR1**	*gone* in **TR2**
subito dopo stat. **A**	*PUÒ ESISTERE*	*PUÒ ESISTERE*
subito dopo stat. **C** in **TR1**	*ESISTE*	*PUÒ ESISTERE*
subito dopo stat. **D**	*PUÒ ESISTERE*	*NON ESISTE*

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Si completi la tabella predisposta qui sotto, indicando i valori delle variabili globali (sempre esistenti)
Inserire in tabella la struttura in pagine della memoria virtuale dei due programmi X e Y (notazione: CX0, CX1, DX0, PX0,... CY0, ..., COND0, ...).
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**	
	pass	*posted*
subito dopo stat. **A**		
subito dopo stat. **B** in **TR1**		
subito dopo stat. **C** in **TR2**		

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Si completi la tabella predisposta qui sotto, indicando i valori delle variabili globali (sempre esistenti)
Inserire in tabella la struttura in pagine della memoria virtuale dei due programmi X e Y (notazione: CX0, CX1, DX0, PX0,... CY0, ..., COND0, ...).
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**	
	pass	*posted*
subito dopo stat. **A**	*1*	
subito dopo stat. **B** in **TR1**		
subito dopo stat. **C** in **TR2**		

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Si completi la tabella predisposta qui sotto, indicando i valori delle variabili globali (sempre esistenti)
Inserire in tabella la struttura in pagine della memoria virtuale dei due programmi X e Y (notazione: CX0, CX1, DX0, PX0,... CY0, ..., COND0, ...).
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**	
	pass	*posted*
subito dopo stat. **A**	*1*	*2*
subito dopo stat. **B** in **TR1**		
subito dopo stat. **C** in **TR2**		

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Si completi la tabella predisposta qui sotto, indicando i valori delle variabili globali (sempre esistenti)
Inserire in tabella la struttura in pagine della memoria virtuale dei due programmi X e Y (notazione: CX0, CX1, DX0, PX0,... CY0, ..., COND0, ...).
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**	
	pass	*posted*
subito dopo stat. **A**	*1*	*2*
subito dopo stat. **B** in **TR1**	*0 / 1*	
subito dopo stat. **C** in **TR2**		

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Si completi la tabella predisposta qui sotto, indicando i valori delle variabili globali (sempre esistenti)
Inserire in tabella la struttura in pagine della memoria virtuale dei due programmi X e Y (notazione: CX0, CX1, DX0, PX0,... CY0, ..., COND0, ...).
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**	
	pass	*posted*
subito dopo stat. **A**	*1*	*2*
subito dopo stat. **B** in **TR1**	*0 / 1*	*1 / 2 / 4*
subito dopo stat. **C** in **TR2**		

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Si completi la tabella predisposta qui sotto, indicando i valori delle variabili globali (sempre esistenti)
Inserire in tabella la struttura in pagine della memoria virtuale dei due programmi X e Y (notazione: CX0, CX1, DX0, PX0,... CY0, ..., COND0, ...).
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**	
	pass	*posted*
subito dopo stat. **A**	*1*	*2*
subito dopo stat. **B** in **TR1**	*0 / 1*	*1 / 2 / 4*
subito dopo stat. **C** in **TR2**	*0*	

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Si completi la tabella predisposta qui sotto, indicando i valori delle variabili globali (sempre esistenti)
Inserire in tabella la struttura in pagine della memoria virtuale dei due programmi X e Y (notazione: CX0, CX1, DX0, PX0,... CY0, ..., COND0, ...).
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**	
	pass	*posted*
subito dopo stat. **A**	*1*	*2*
subito dopo stat. **B** in **TR1**	*0 / 1*	*1 / 2 / 4*
subito dopo stat. **C** in **TR2**	*0*	*2 / 3*

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Il sistema può andare in stallo (deadlock). Qui si indichino gli statement dove si bloccano i thread incorsi in stallo (non necessariamente tutti):

HR	TR1	TR2

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Il sistema può andare in stallo (deadlock). Qui si indichino gli statement dove si bloccano i thread incorsi in stallo (non necessariamente tutti):

HR	TR1	TR2
pthread_mutex_lock *(&gate)*		

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Il sistema può andare in stallo (deadlock). Qui si indichino gli statement dove si bloccano i thread incorsi in stallo (non necessariamente tutti):

HR	TR1	TR2
pthread_mutex_lock *(&gate)*	***sem_wait*** *(&pass)*	

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Il sistema può andare in stallo (deadlock). Qui si indichino gli statement dove si bloccano i thread incorsi in stallo (non necessariamente tutti):

HR	TR1	TR2
pthread_mutex_lock *(&gate)*	***sem_wait*** *(&pass)*	*si trova a monte di* ***pthread_mutex_lock*** *o è anch'esso fermo su* ***pthread_mutex_lock***

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Es6: Parallelismo

Il sistema può andare in stallo (deadlock). Qui si indichino gli statement dove si bloccano i thread incorsi in stallo (non necessariamente tutti):

HR	TR1	TR2
pthread_mutex_lock *(&gate)*	***sem_wait*** *(&pass)*	*si trova a monte di* ***pthread_mutex_lock*** *o è anch'esso fermo su* ***pthread_mutex_lock***

Si possono scambiare i ruoli di TR1 e TR2.

```
/* variabili globali                                                    */
pthread_mutex_t gate = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int posted = 1;

void * header (void * arg) {                          /* funzione di thread  */

    printf ("Started.\n");
    pthread_mutex_lock (&gate);
    sem_post (&pass);
    posted = posted + 1;                              /* statement A */
    pthread_mutex_unlock (&gate);
    printf ("Finished.\n");
    return NULL;

} /* header */

void * trailer (void * id) {                          /* funzione di thread  */

    int gone = 0;                                     /* variabile locale    */

    pthread_mutex_lock (&gate);                       /* statement B */
    sem_wait (&pass);
    pthread_mutex_unlock (&gate);
    gone = 1;                                         /* statement C */
    sem_post (&pass);
    posted = posted + (int) id;
    return NULL;

} /* trailer */

int main (int argc, char * argv [ ]) {                /* thread principale   */

    pthread_t hr, tr1, tr2;
    sem_init (&pass, 0, 0);
    pthread_create (&hr, NULL, &header, NULL);
    pthread_create (&tr1, NULL, &trailer, (void *) 1);
    pthread_create (&tr2, NULL, &trailer, (void *) 2);
    pthread_join (hr);
    pthread_join (tr2);                               /* statement D */
    pthread_join (tr1);

} /* main */
```

Alla prossima lezione

alessandro.nacci@polimi.it