# RACCOLTA DI ESERCIZI
# SU THREAD E PROGRAMMAZIONE CONCORRENTE

## I docenti di AXO

## A.A. 2015-16

## esercizio su thread e parallelismo – mercoledì 21 novembre 2012

Si consideri il programma C seguente (gli "#include" sono omessi):

```
pthread_mutex_t law   = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t order = PTHREAD_MUTEX_INITIALIZER;
sem_t mess;
int global = 0;
```

```
void * one (void * arg) {
   int local = 0;
   pthread_mutex_lock (&law);
      sem_post (&mess);
      local = 1;                                          /* statement A */
   pthread_mutex_unlock (&law);
   global = 1;
   pthread_mutex_lock (&order);
      local = 3;                                          /* statement B */
   pthread_mutex_unlock (&order);
   return (void *) global;
} /* end one */
```

```
void * two (void * arg) {
   int local = 0;
   pthread_mutex_lock (&order);
      global = 2;
      pthread_mutex_lock (&law);
         sem_wait (&mess);                                /* statement C */
         local = (int) arg;
      pthread_mutex_unlock (&law);
   pthread_mutex_unlock (&order);
   return NULL;
} /* end two */
```

```
void main ( ) {
   pthread_t th_1, th_2;
   sem_init (&mess, 0, 0);
   pthread_create (&th_2, NULL, two,  (void *) 2);
   pthread_create (&th_1, NULL, one, NULL);
   pthread_join (th_1, &global);                          /* statement D */
   pthread_join (th_2, NULL);
return;
} /* end main */
```

**Si completi** la tabella qui sotto **indicando lo stato di esistenza della variabile locale** nell'istante di tempo specificato da ciascuna condizione, così: se la variabile **esiste**, si scriva ESISTE; se **non esiste**, si scriva NON ESISTE; e se può essere **esistente o inesistente**, si scriva PUÒ ESISTERE. Ogni casella della tabella va riempita in uno dei tre modi (non va lasciata vuota).

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede lo stato che la variabile o il parametro assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **variabile locale** | |
|---|---|---|
| | *local* in **th_1** | *local* in **th_2** |
| subito dopo stat. **A** | *ESISTE* | *ESISTE* |
| subito dopo stat. **C** | *PUÒ ESISTERE* | *ESISTE* |
| subito dopo stat. **D** | *NON ESISTE* | *PUÒ ESISTERE* |

**Si completi** la tabella qui sotto, **indicando i valori delle variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha un valore definito; oppure X quando è indefinita
- se la variabile può avere due o più valori, li si riporti tutti quanti
- il semaforo può avere valore positivo o nullo (non valore negativo)

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede il valore (o i valori) che la variabile ha tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **variabili globali** | |
|---|---|---|
| | *mess* | *global* |
| subito dopo stat. **A** | *1* | *0 / 2* |
| subito dopo stat. **B** | *0 / 1* | *1 / 2* |
| subito dopo stat. **C** | *0* | *1 / 2* |
| subito dopo stat. **D** | *0 / 1* | *1 / 2* |

**Il sistema può andare in stallo** (deadlock). Qui **si indichino** gli statement dove si bloccano i due thread, precisando il valore (o i valori) della variabile `global`:

| **th_1** | **th_2** | *global* |
|---|---|---|
| `mutex_lock (law)` | `sem_wait (mess)` | *2* |

## esercizio su thread e parallelismo – venerdì 20 settembre 2013

Si consideri il programma C seguente (gli "#include" sono omessi):

```
pthread_mutex_t ONE = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t TWO = PTHREAD_MUTEX_INITIALIZER;
sem_t GATE;
int global = 1;

void * first (void * arg) {
   int local = 0;
   pthread_mutex_lock (&ONE);                          /* statement A */
   global = 2;
   sem_wait (&GATE);                                   /* statement B */
   pthread_mutex_lock (&TWO);
   global = 5;
   pthread_mutex_unlock (&ONE);
   pthread_mutex_unlock (&TWO);
   return NULL;
} /* end first */

void * last (void * arg) {
   int local = 0;
   pthread_mutex_lock (&TWO);
   global = 3;
   pthread_mutex_lock (&ONE);
   global = 4;
   sem_post (&GATE);                                   /* statement C */
   pthread_mutex_unlock (&ONE);
   global = 6;
   pthread_mutex_unlock (&TWO);
   return NULL;
} /* end last */

void main ( ) {
   pthread_t TH_1, TH_2;
   sem_init (&GATE, 0, 0);
   pthread_create (&TH_1, NULL, first, NULL);
   pthread_create (&TH_2, NULL, last, NULL);
   pthread_join (TH_1, NULL);                          /* statement D */
   pthread_join (TH_2, NULL);
   return;
} /* end main */
```

**Si completi** la tabella qui sotto **indicando lo stato di esistenza della variabile locale** nell'istante di tempo specificato da ciascuna condizione, così: se la variabile **esiste**, si scriva ESISTE; se **non esiste**, si scriva NON ESISTE; e se può essere **esistente o inesistente**, si scriva PUÒ ESISTERE. Ogni casella della tabella va riempita in uno dei tre modi (non va lasciata vuota).

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede lo stato che la variabile o il parametro assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **variabile locale** | |
|---|---|---|
| | *local* in **TH_1** | *local* in **TH_2** |
| subito dopo stat. **A** | *ESISTE* | *PUÒ ESISTERE* |
| subito dopo stat. **C** | *ESISTE* | *ESISTE* |
| subito dopo stat. **D** | *NON ESISTE* | *PUÒ ESISTERE* |

**Si completi** la tabella qui sotto, **indicando i valori delle variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha un valore definito; oppure X quando è indefinita
- se la variabile può avere due o più valori, li si riporti tutti quanti
- il semaforo può avere valore positivo o nullo (non valore negativo)

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede il valore (o i valori) che la variabile ha tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **variabili globali** | |
|---|---|---|
| | *GATE* | *global* |
| subito dopo stat. **A** | *0 / 1* | *1 / 3 / 4 / 6* |
| subito dopo stat. **B** | *0* | *2 / 6* |
| subito dopo stat. **C** | *1* | *4* |
| subito dopo stat. **D** | *0* | *5* |

**Il sistema può andare in stallo** (deadlock). Qui **si indichino** gli statement dove si bloccano i due thread, precisando il valore (o i valori) della variabile `global`:

| **TH_1** | **TH_2** | *global* |
|---|---|---|
| ***sem_wait*** *(GATE)* | ***mutex_lock*** *(ONE)* | *2 / 3* |

## esercizio su thread e parallelismo – mercoledì 19 settembre 2010

Si consideri il programma C seguente (gli "#include" sono omessi):

```
/* variabili globali                                              */
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
sem_t pass;
int glob = 2;
```

```
void * begin ( ) {                                  /* funzione di thread  */
     int loc;                                       /* variabile locale    */
     if (glob == 2) {
          pthread_mutex_lock (&lock);               /* statement A */
          sem_post (&pass);                         /* statement B */
          pthread_mutex_unlock (&lock);
          sem_post (&pass);
     } /* if */
     return NULL;
} /* begin */
```

```
void * end ( ) {                                    /* funzione di thread  */
     int loc;                                       /* variabile locale    */
     sem_wait (&pass);                              /* statement C */
     glob = 3;
     pthread_mutex_lock (&lock);
     sem_wait (&pass);                              /* statement D */
     pthread_mutex_unlock (&lock);
     sem_wait (&pass);
     return NULL;
} /* end */
```

```
void main ( ) {                                     /* thread principale   */
     pthread_t T1, T2;
     sem_init (&pass, 0, 1);                        /* inizializzato a 1   */
     pthread_create (&T1, NULL, begin, NULL);
     pthread_create (&T2, NULL, end, NULL);
     pthread_join (T2);                             /* statement E */
     pthread_join (T1);
} /* main */
```

**Si completi** la tabella qui sotto **i**ndicando lo **stato di esistenza** della **variabile locale** e del **parametro** nell'istante di tempo specificato da ciascuna condizione, così: se la variabile o il parametro **esiste**, si scriva ESISTE; se **non esiste**, si scriva NON ESISTE; e se può essere **esistente** o **inesistente**, si scriva PUÒ ESISTERE. Ogni casella della tabella va riempita in uno dei tre modi (non va lasciata vuota).

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede lo stato che la variabile o il parametro assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **variabile locale** | |
|---|---|---|
| | *loc* in **T1** | *loc* in **T2** |
| subito dopo stat. **A** | *ESISTE* | *PUÒ ESISTERE* |
| subito dopo stat. **C** | *PUÒ ESISTERE* | *ESISTE* |
| subito dopo stat. **E** | *PUÒ ESISTERE* | *NON ESISTE* |

**Si completi** la tabella qui sotto, indicando i **valori** delle **variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha 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* | *glob* |
| subito dopo stat. **A** | *0 / 1* | *2 / 3* |
| subito dopo stat. **B** | *1 / 2* | *2 / 3* |
| subito dopo stat. **C** | *0 / 1 / 2* | *2* |
| subito dopo stat. **D** | *0 / 1* | *3* |
| subito dopo stat. **E** | *0* | *3* |

Il sistema può andare in **stallo** (deadlock) in **due** situazioni. Qui sotto **si scriva** lo statement dove il thread si blocca oppure se esso termina (per avere uno stallo almeno un thread si deve bloccare):

| situazione | **`T1`** | **`T2`** |
|---|---|---|
| 1 | *`terminato`* | *`seconda sem_wait`* |
| 2 | *`pthread_mutex_lock`* | *`seconda sem_wait`* |

*Nota: un thread termina e l'altro si blocca sul semaforo, oppure un thread si blocca sul mutex e l'altro sul semaforo.*

### esercizio su thread e parallelismo – lunedì 4 luglio 2011

Si consideri il programma C seguente (gli "#include" sono omessi):

```
/* 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 */
```

**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* |

**Si completi** la tabella predisposta qui sotto, **indicando i valori delle variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha un valore definito; oppure X quando è indefinita
- se la variabile può avere due o più valori, li si riporti tutti quanti
- il semaforo può avere valore positivo o nullo (non valore negativo)

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede il valore (o i valori) che la variabile ha tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **variabili globali** | |
|---|---|---|
| | *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* |

**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.*

### esercizio su thread e parallelismo – venerdì 2 marzo 2012

Si consideri il programma C seguente (gli "#include" sono omessi):

```
/* variabili globali                                                    */
pthread_mutex_t elem = PTHREAD_MUTEX_INITIALIZER;
sem_t check;
int counter = 0;
```

---

```
void * sequence (void * arg) {                    /* funzione di thread  */

    int num;                                      /* variabile locale    */

    printf ("Sequence started.\n");               /* statement A */
    for (num = 1; num <= 3; num++) {
        pthread_mutex_lock (&elem);
        sem_post (&check);
        counter++;                                /* statement B */
        pthread_mutex_unlock (&elem);
    } /* for */
    printf ("Sequence ended.\n");
    return NULL;

} /* sequence */
```

---

```
void * single (void * arg) {                      /* funzione di thread  */

    sem_wait (&check);                            /* statement C */
    pthread_mutex_lock (&elem);
    sem_wait (&check);
    counter--;                                    /* statement D */
    pthread_mutex_unlock (&elem);
    return NULL;

} /* single */
```

---

```
void main ( ) {                                   /* thread principale   */

    pthread_t T1, T2;
    sem_init (&check, 0, 0);
    pthread_create (&T2, NULL, single, (void *) 1);
    pthread_create (&T1, NULL, sequence, NULL);
    pthread_join (T1);                            /* statement E */
    pthread_join (T2);

} /* main */
```

---

**Si completi** la tabella qui sotto **indicando lo stato di esistenza della variabile locale e del parametro** nell'istante di tempo specificato da ciascuna condizione, così: se la variabile o il parametro **esiste**, si scriva ESISTE; se **non esiste**, si scriva NON ESISTE; e se può essere **esistente o inesistente**, si scriva PUÒ ESISTERE. Ogni casella della tabella va riempita in uno dei tre modi (non va lasciata vuota).

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede lo stato che la variabile o il parametro assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **variabile locale o parametro** | |
|---|---|---|
| | *num* in **T1** | *arg* in **T2** |
| subito dopo stat. **A** | *ESISTE* | *ESISTE* |
| subito dopo stat. **C** | *PUÒ ESISTERE* | *ESISTE* |
| subito dopo stat. **E** | *NON ESISTE* | *PUÒ ESISTERE* |

**Si completi** la tabella qui sotto, **indicando i valori delle variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha un valore definito; oppure X quando è indefinita
- se la variabile può avere due o più valori, li si riporti tutti quanti
- il semaforo può avere valore positivo o nullo (non valore negativo)

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede il valore (o i valori) che la variabile ha tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **variabili globali** | |
|---|---|---|
| | *check* | *counter* |
| subito dopo stat. **B** (`num` = 1) | *1 / 0* | *1* |
| subito dopo stat. **B** (`num` = 2) | *2 / 1* | *2* |
| subito dopo stat. **B** (`num` = 3) | *3 / 2 / 1* | *3 / 2* |
| subito dopo stat. **C** | *2 / 1 / 0* | *3 / 2 / 1 / 0* |
| subito dopo stat. **D** | *1 / 0* | *2 / 1* |

**Il sistema può andare in stallo** (deadlock). Qui **si indichino** gli statement dove si bloccano i due thread, precisando anche l'iterazione di **T1** (specificando il valore di `num`) e il/i valore/i di *`counter`*:

| **T1** | **T2** | *counter* |
|---|---|---|
| ***pthread_mutex_lock*** *num =* ***2*** | *seconda* ***sem_wait*** | *1* |

*A un primo sguardo si riconosce subito che il thread T1 può iterare la sequenza critica fino a tre volte, se non si blocca in stallo sul* `lock` *a guardia della sequenza critica, e che thread T2 può eseguire la sequenza critica una sola volta, se non si blocca in stallo sulla* `wait` *interna alla sequenza critica (la seconda* `wait`*).*

*Il thread T2 non raggiunge e dunque non entra in sequenza critica se il thread T1 non ha passata almeno una* `post` *entrando in sequenza critica e uscendone. Se però il thread T2 entra in sequenza critica quando il thread T1 ha iterata la sequenza critica una sola volta e dunque ha passata una sola* `post`*, il thread T2 si sospende sulla* `wait` *dentro la sequenza critica (in attesa di una* `post` *che non arriverà mai) e prima o poi il thread T1 si sospende sul* `lock` *a guardia della sequenza critica, nella seconda iterazione; così il sistema è bloccato in stallo. Se infine il thread T1 ha iterata la sequenza critica almeno due volte, il thread T1 può passare entrambe le* `wait`*. I due thread possono terminare secondo vari ordini possibli.*

*Riassumendo con ordine, si ha questo comportamento:*

- *se il thread T1 non ha eseguita la sequenza critica, il thread T2 non può entrare in sequenza critica*
- *se il thread T1 ha iterata la sequenza critica una sola volta e il thread T2 entra in sequenza critica, il sistema va in stallo (deadlock) così:*
  - *T1 si sospende sul* `lock` *a guardia della sequenza critica, con indice* `num` *= 2*
  - *T2 si sospende sulla* `wait` *dentro la sequenza critica*
- *se il thread T1 ha iterata la sequenza critica due o tre volte e il thread T2 entra in sequenza critica, il sistema non va in stallo ed entrambi i thread T1 e T2 terminano (con vari ordini di terminazione)*

*Detto altrimenti la sequenza critica di T2 è eseguibile per intero, ossia senza sospendersi sulla* `wait` *interna, non prima della seconda iterazione della sequenza critica di T1. Ciò dipende appunto dal fatto che il thread T1 esegue tre* `post` *sul semaforo, una per ciascuna iterazione della sequenza critica, e dal fatto che il thread T2 esegue due* `wait`*, la prima davanti alla sequenza critica e la seconda dentro.*

*La variabile* `num` *e il parametro* `arg` *sono locali ai thread T1 e T2, rispettivamente. Un parametro si comporta come una variabile locale: esiste nel contesto della funzione che lo dichiara. Ecco gli stati delle variabili locali:*

- *in A, che è nel contesto di T1, si ha che ovviamente* `num` *esiste, e che* `arg` *esiste poiché T2 è creato prima di T1 e T2 non può essere terminato giacché T1 non ha passata nessuna* `post`
- *in C, che è nel contesto di T2, si ha che* `num` *può esistere poiché se T2 ha passata la prima* `wait` *allora T1 deve essere iniziato per avere passata almeno una* `post` *o essere terminato prima ancora che T2 abbia raggiunto C, e che ovviamente* `arg` *esiste*
- *in E, che è nel contesto del thread principale main, si ha che* `num` *non esiste poiché T1 è terminato, e che* `arg` *può esistere poiché T2 è stato creato e può essere terminato*

*Le variabili globali* `check` *e* `counter` *sono visibili a tutti i thread. La variabile* `check` *è un semaforo inizializzato a zero e bilancia tutte le esecuzioni di* `post` *e* `wait`*. La variabile* `counter` *è inizializzata a zero e viene incrementata e decrementata a valle della* `post` *e a valle della* `wait` *interne alle sequenze critiche di T1 e T2, rispettivamente, e dunque bilancia le esecuzioni di queste* `post` *e* `wait` *ma ignora le esecuzioni della prima* `wait` *di T2. Ecco i valori delle variabili globali:*

- *in B, che è nel contesto di T1, si ha:*
  - *con* `num` *= 1 e dunque una* `post`*, che* `check` *vale 1 o 0 poiché T2 ha passate nessuna o la prima* `wait`*, e che* `counter` *vale 1 poiché T2 non ha passata la seconda* `wait`
  - *con* `num` *= 2 e dunque due* `post`*, che* `check` *vale 2 o 1 poiché T2 ha passate nessuna o la prima* `wait` *ma non la seconda, e che* `counter` *vale 2 poiché T2 non ha passata la seconda* `wait`
  - *con* `num` *= 3 e dunque tre* `post`*, che* `check` *vale 3 2 o 1 poiché T2 ha passate nessuna una o entrambe le* `wait`*, e che* `counter` *vale 3 o 2 poiché T2 può (ma non necessariamente deve) avere passata la seconda* `wait`
- *in C, che è nel contesto di T2, si ha che* `check` *vale 2 1 o 0 poiché T1 ha passate tre due o una* `post` *e T2 ha passata la prima* `wait` *ma non la seconda, e che* `counter` *vale 3 2 o 1 poiché T1 ha passate tre due o una* `post` *e T2 non ha passata la seconda* `wait`
- *in D, che è nel contesto di T2, si ha che* `check` *vale 1 o 0 poiché T1 ha passate tre o due* `post` *e T2 ha passate entrambe le* `wait`*, e che* `counter` *vale 2 o 1 poiché T1 ha passate tre o due* `post` *e T2 ha passata la seconda* `wait`

*Dello stallo si è già detto: thread T1 e T2 sospesi sulla seconda iterazione del* `lock` *a guardia della sequenza critica e sulla seconda* `wait`*, rispettivamente; la variabile globale* `counter` *vale 1 poiché T1 ha passata una*

*`post` e T2 non ha passata la seconda `wait`. Questo è l'unico stallo poiché T1 esegue più `post` di quante `wait` siano eseguite da T2 e la prima `wait` di T2 si sblocca senz'altro pur di attendere una `post` di T1.*

## esercizio su thread e parallelismo – giovedì 21 febbraio 2013

Si consideri il programma C seguente (gli "#include" sono omessi):

```
pthread_mutex_t OUTER = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t INNER = PTHREAD_MUTEX_INITIALIZER;
sem_t MIDDLE;
int global = 0;

void * one (void * arg) {
   int local = 0;
   pthread_mutex_lock (&OUTER);                          /* statement A */
   pthread_mutex_lock (&INNER);
   global = 1;
   sem_wait (&MIDDLE);                                   /* statement B */
   pthread_mutex_unlock (&OUTER);
   global = 2;
   pthread_mutex_unlock (&INNER);
   return NULL;
} /* end one */

void * nine (void * arg) {
   int local = 0;
   pthread_mutex_lock (&INNER);
   global = 3;
   pthread_mutex_unlock (&INNER);
   sem_post (&MIDDLE);                                   /* statement C */
   pthread_mutex_lock (&OUTER);
   global = 4;
   pthread_mutex_unlock (&OUTER);
   local = (int) arg;
   return (void *) local;
} /* end nine */

void main ( ) {
   pthread_t TH_1, TH_9;
   sem_init (&MIDDLE, 0, 0);
   pthread_create (&TH_1, NULL, one, NULL);
   pthread_create (&TH_9, NULL, nine, (void *) 9);
   pthread_join (TH_9, &global);
   printf ("concluding ...");                            /* statement D */
   pthread_join (TH_1, NULL);
   return;
} /* end main */
```

**Si completi** la tabella qui sotto **indicando lo stato di esistenza della variabile locale** nell'istante di tempo specificato da ciascuna condizione, così: se la variabile **esiste**, si scriva ESISTE; se **non esiste**, si scriva NON ESISTE; e se può essere **esistente o inesistente**, si scriva PUÒ ESISTERE. Ogni casella della tabella va riempita in uno dei tre modi (non va lasciata vuota).

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede lo stato che la variabile o il parametro assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **variabile locale** | |
|---|---|---|
| | *local* in **TH_1** | *local* in **TH_9** |
| subito dopo stat. **A** | *ESISTE* | *PUÒ ESISTERE* |
| subito dopo stat. **C** | *PUÒ ESISTERE* | *ESISTE* |
| subito dopo stat. **D** | *PUÒ ESISTERE* | *NON ESISTE* |

**Si completi** la tabella qui sotto, **indicando i valori delle variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha un valore definito; oppure X quando è indefinita
- se la variabile può avere due o più valori, li si riporti tutti quanti
- il semaforo può avere valore positivo o nullo (non valore negativo)

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede il valore (o i valori) che la variabile ha tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **variabili globali** | |
|---|---|---|
| | *MIDDLE* | *global* |
| subito dopo stat. **A** | *0 / 1* | *0 / 3 / 4 / 9* |
| subito dopo stat. **B** | *0* | *1 / 9* |
| subito dopo stat. **C** | *0 / 1* | *1 / 2 / 3* |
| subito dopo stat. **D** | *0 / 1* | *1 / 2 / 9* |

**Il sistema può andare in stallo** (deadlock). Qui **si indichino** gli statement dove si bloccano i due thread, precisando il valore (o i valori) della variabile `global`:

| **TH_1** | **TH_9** | *global* |
|---|---|---|
| ***sem_wait*** *(MIDDLE)* | ***mutex_lock*** *(INNER)* | *1* |

## esercizio su thread e parallelismo – giovedì 27 febbraio 2014

Si consideri il programma C seguente (gli "#include" sono omessi):

```
pthread_mutex_t OMEGA = PTHREAD_MUTEX_INITIALIZER;
sem_t ONE, TWO;
int global = 0;
void * A (void * arg) {
   int local = 0;
   sem_wait (&ONE);                                   /* istruzione A */
   pthread_mutex_lock (&OMEGA);
      sem_wait (&TWO);                                /* istruzione B */
   pthread_mutex_unlock (&OMEGA);
   return NULL;
} /* end A */
void * B (void * arg) {
   int local = 0;
   pthread_mutex_lock (&OMEGA);
      sem_post (&ONE);
      sem_post (&TWO);
   pthread_mutex_unlock (&OMEGA);                     /* istruzione C */
   return NULL;
} /* end B */
void * C (void * arg) {
   sem_wait (&TWO);
   sem_wait (&TWO);                                   /* istruzione D */
   global = 1;
   return NULL;
} /* end C */
void main ( ) {
   pthread_t TH_1, TH_2, TH_3;
   sem_init (&ONE, 0, 0);
   sem_init (&TWO, 0, 1);
   pthread_create (&TH_1, NULL, A, NULL);
   pthread_create (&TH_2, NULL, B, NULL);
   pthread_create (&TH_3, NULL, C, NULL);
   pthread_join (TH_2, NULL);
   pthread_join (TH_3, NULL);                         /* istruzione E */
   pthread_join (TH_1, NULL);
   return;
} /* end main */
```

**Si completi** la tabella qui sotto **indicando lo stato di esistenza della variabile locale** nell'istante di tempo specificato da ciascuna condizione, così: se la variabile **esiste**, si scriva ESISTE; se **non esiste**, si scriva NON ESISTE; e se può essere **esistente o inesistente**, si scriva PUÒ ESISTERE. Ogni casella della tabella va riempita in uno dei tre modi (non va lasciata vuota).

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede lo stato che la variabile o il parametro assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **variabile locale** | |
|---|---|---|
| | *local* in **TH_1** | *local* in **TH_2** |
| subito dopo istr. **A** | *ESISTE* | *PUÒ ESISTERE* |
| subito dopo istr. **C** | *PUÒ ESISTERE* | *ESISTE* |
| subito dopo istr. **D** | *ESISTE* | *PUÒ ESISTERE* |
| subito dopo istr. **E** | *ESISTE* | *NON ESISTE* |

**Si completi** la tabella qui sotto, **indicando i valori delle variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha un valore definito; oppure X quando è indefinita
- se la variabile può avere due o più valori, li si riporti tutti quanti
- il semaforo può avere valore positivo o nullo (non valore negativo)

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede il valore (o i valori) che la variabile ha tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **variabile globale** | | |
|---|---|---|---|
| | *ONE* | *TWO* | *global* |
| subito dopo istr. **A** | *0* | *0 / 1 / 2* | *0 / 1* |
| subito dopo istr. **B** | *0* | *0 / 1* | *0* |
| subito dopo istr. **C** | *0 / 1* | *0 / 1 / 2* | *0 / 1* |
| subito dopo istr. **D** | *0 / 1* | *0* | *0* |

**Il sistema va sempre in stallo** (deadlock), in due casi diversi. Qui **si indichino** le primitive dove si bloccano i thread (uno solo o più di uno), precisando il valore (o i valori) della variabile `global`:

| **caso** | **TH_1** | **TH_2** | **TH_3** | *global* |
|---|---|---|---|---|
| **1** | `wait TWO` | – | – | *0 / 1* |
| **2** | – | – | `seconda wait TWO` | *0* |

## esercizio su thread e parallelismo – giovedì 26 febbraio 2015

Si consideri il programma C seguente (gli "#include" sono omessi):

```
pthread_mutex_t GATE = PTHREAD_MUTEX_INITIALIZER;
sem_t GO;
int global = 0;

void * FIRST (void * arg) {
   sem_wait (&GO);                                   /* istruzione A */
   global = 1;
   pthread_mutex_lock (&GATE);
   sem_post (&GO);
   pthread_mutex_unlock (&GATE);
   return NULL;
} /* end FIRST */

void * LAST (void * arg) {
   if (global == 0) {
      global = 2;
      pthread_mutex_lock (&GATE);                    /* istruzione B */
      sem_wait (&GO);
      pthread_mutex_unlock (&GATE);
      sem_post (&GO);
   } else {
      global = 3;
      sem_wait (&GO);                                /* istruzione C */
   } /* if */
   return NULL;
} /* end LAST */

void main ( ) {
   pthread_t TH_1, TH_2, TH_3;
   sem_init (&GO, 0, 1);
   pthread_create (&TH_1, NULL, FIRST, NULL);
   pthread_create (&TH_2, NULL, LAST, NULL);
   pthread_join (TH_1, NULL);
   pthread_create (&TH_3, NULL, FIRST, NULL);
   pthread_join (TH_3, NULL);                        /* istruzione D */
   pthread_join (TH_2, NULL);
   return;
} /* end main */
```

**Si completi** la tabella qui sotto **indicando lo stato di esistenza del thread** nell'istante di tempo specificato da ciascuna condizione, così: se il thread **esiste**, si scriva ESISTE; se **non esiste**, si scriva NON ESISTE; e se può essere **esistente o inesistente**, si scriva PUÒ ESISTERE. Ogni casella della tabella va riempita in uno dei tre modi (non va lasciata vuota).

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede lo stato che la variabile o il parametro assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **stato del thread** | | |
|---|---|---|---|
| | **TH_1** | **TH_2** | **TH_3** |
| subito dopo istr. **A** in **TH_1** | *ESISTE* | *PUÒ ESISTERE* | *NON ESISTE* |
| subito dopo istr. **B** | *PUÒ ESISTERE* | *ESISTE* | *PUÒ ESISTERE* |
| subito dopo istr. **D** | *NON ESISTE* | *PUÒ ESISTERE* | *NON ESISTE* |

**Si completi** la tabella qui sotto, **indicando i valori delle variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha un valore definito; oppure X quando è indefinita
- se la variabile può avere due o più valori, li si riporti tutti quanti
- il semaforo può avere valore positivo o nullo (non valore negativo)

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede il valore (o i valori) che la variabile ha tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **variabile globale** | |
|---|---|---|
| | *GO* | *global* |
| subito dopo istr. **A** in **TH_1** | *0* | *0 / 2* |
| subito dopo istr. **B** | *0 / 1* | *1 / 2* |
| subito dopo istr. **C** | *0* | *1 / 3* |
| subito dopo istr. **D** | *0 / 1* | *1 / 2 / 3* |

**Il sistema va in stallo** (deadlock), in **uno o più casi**. Qui **si indichino** le primitive dove si bloccano i thread (uno solo o più di uno), precisando il valore (o i valori) della variabile `global` (il numero di righe nella tabella non è significativo):

| **caso** | **TH_1** | **TH_2** | **TH_3** | *global* |
|---|---|---|---|---|
| **1** | `lock` | `prima wait` | `non creato` | *1 / 2* |
| **2** | `terminato` | `prima wait` | `lock` | *1 / 2* |
| **3** | `terminato` | `copo C` | `wait` | *3* |
| **4** | | | | |

## esercizio su thread e programmazione concorrente – martedì 25 novembre 2014

Si consideri la bozza di sistema concorrente seguente (in linguaggio C), con un thread principale e due thread secondari, per i quali sono già previsti un mutex (`MUX`) e due semafori (`ABC` e `XYZ`) inizializzati a zero. Il thread principale e quelli secondari vanno completati – secondo le modalità spiegate qui di seguito e tenendo conto della tabella mostrata a pagina seguente – laddove nel codice ci sono righe lasciate appositamente vuote (la numerazione delle righe vuote serve solo per distinguerle l'una dall'altra). In una riga vuota si può aggiungere **una sola** chiamata di sistema: o relativa al mutex `MUX` (`lock` o `unlock`) o relativa ai semafori `ABC` e `XYZ` (`post` o `wait`). **Alcune righe possono rimanere eventualmente vuote**.

```
pthread_mutex_t MUX
sem_t ABC, XYZ
int globale

void * alfa (...) {

    sem_wait (&ABC) _____________________________ (1)

    pthread_mutex_lock (&MUX) ___________________ (2)

    _____________________________________________ (3)

    globale = 1                                   /* statement A */
    pthread_mutex_unlock (&MUX)

    sem_post (&XYZ) _____________________________ (4)

    _____________________________________________ (5)

    return NULL
} /* end alfa */

void * beta (...) {
    pthread_mutex_lock (&MUX)                     /* statement B */

    sem_post (&ABC) _____________________________ (6)

    _____________________________________________ (7)

    globale = 2
    pthread_mutex_unlock (&MUX)
    return (void * 3)
} /* end beta */

void main ( ) {
    pthread_t th_1, th_2
    sem_init (&ABC, 0, 0)
    sem_init (&XYZ, 0, 0)
    pthread_create (&th_1, NULL, alfa, NULL)
    pthread_create (&th_2, NULL, beta, NULL)

    sem_wait (&XYZ) _____________________________ (8)
    pthread_join (th_2, &globale)
    printf ("%d", globale)                        /* statement C */
    pthread_join (th_1, NULL)
    return
} /* end main */
```

**Si consideri** la tabella seguente, la quale specifica parzialmente lo **stato** che il sistema concorrente dovrebbe assumere in tre **condizioni** differenti (A, B e C, indicate nel codice del sistema).

**Si completi** il codice dato a pagina precedente, secondo le modalità spiegate all'inizio, così da ottenere un sistema concorrente che **si comporti** precisamente **come specificato dalla tabella**, e **non abbia stalli**.

Si badi bene alla colonna "condizione": con "subito dopo statement X" si chiede lo stato che il sistema assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **stato del sistema concorrente** | | |
|---|---|---|---|
| | *th_1* | *globale* | *stdout* |
| subito dopo stat. **A** | | vietato vedere il valore 2 | |
| subito dopo stat. **B** | ESISTE | | |
| subito dopo stat. **C** | | | scritto il valore 3 |

*Preliminarmente si può osservare che il codice contiene un meccanismo mutex evidentemente incompleto, il quale ragionevolmente andrà completato in modo opportuno. Comunque in definitiva sono le condizioni A, B e C, quelle che determinano come concludere il progetto del sistema. Dapprima è opportuno esaminarle singolarmente, ma poi bisogna anche tenere conto delle loro eventuali interazioni.*

*Qui si sceglie di cominciare con le condizioni più semplici da soddisfare. Procedendo dapprima in modo intuitivo: la condizione A coinvolge solo alfa e beta in due punti precisi ed è formulata come divieto di coesistenza tra due osservabili (i valori 1 e 2), ossia come un tipico caso di mutua esclusione, nel codice c'è già un mutex da completare, dunque è ragionevole soddisfare prima A; tra le condizioni B e C, la prima è formulata come esistenza ed è facilmente ottenibile con un semaforo (ma badando a evitare stalli poiché c'è un mutex coinvolto), mentre la seconda è formulata come obbligo (potenzialmente complesso poiché vanno esclusi tutti gli osservabili indesiderati), dunque è ragionevole soddisfare prima B; resta solo la condizione C, che con un poco più di riflessione (qui ci sono due valori da escludere, 1 e 2, ma se ne scarta subito uno) è pure risolubile con un semaforo. Pertanto è ragionevole esaminare le condizioni nell'ordine A, B e C.*

*Per la condizione A: beta non deve eseguire l'assegnamento* `globale = 2`*, il quale è in sequenza critica, subito dopo che alfa ha eseguito lo statement A; basta mettere anche* `globale = 1` *in sequenza critica:* `lock` *in riga 1, 2 o 3; in quale delle tre righe metterlo è deciso dalla condizione B, come si vede qui sotto.*

*Per la condizione B: alfa viene senz'altro creato prima di beta, ma non deve essere già terminato quando beta esegue lo statement B; basta usare il semaforo* `ABC`*: si colloca* `wait` *a monte di* `lock` *– se stesse a valle dentro la sequenza critica potrebbe dare stallo – e dunque in riga 1, e si colloca* `post` *in riga 6 (alla fine la riga 7 resterà vuota); ora la sequenza critica di* `alfa` *è senz'altro eseguita dopo quella di* `beta`*; e così il posizionamento di* `lock` *in riga 2 è determinato (alla fine la riga 3 resterà vuota).*

*Per la condizione C: con* `printf` *main deve scrivere il valore 3, il quale proviene senz'altro dal meccanismo return-join di beta; ciò permette di scartare subito il valore 2, pure assegnato a* `globale` *da parte di beta ma prima di terminare con* `return`*; dunque basta garantire che con* `printf` *main non scriva il valore 1, il quale però potrebbe essere stato assegnato a* `globale` *da parte di alfa; al momento tale evento di scrittura risulta possibile poiché la sequenza critica di alfa è serializzata dopo quella di beta; pertanto bisogna impedire che l'assegnamento* `globale = 1` *di alfa si inserisca tra la prima* `join` *e la* `printf`*; basta usare il semaforo* `XYZ`*: si colloca* `post` *a valle di* `globale = 1` *e dunque in riga 4 (la riga 5 resta vuota), e si colloca* `wait` *a monte di* `join` *– e a maggior ragione a monte di* `printf` *– e dunque in riga 8.*

*Pertanto in totale vengono aggiunte cinque chiamate di sistema e tre righe restano vuote. Ovviamente i semafori* `ABC` *e* `XYZ` *sono interscambiabili, ma non è una modifica strutturale al codice. I gruppi di linee (1-2-3), (4-5) e (6-7) sono riempiti a partire dalla loro prima linea onde evitare varianti puramente grafiche.*

*VARIANTE ALLA SOLUZIONE*

*La presenza di righe vuote in eccedenza permette di riposizionare diversamente le chiamate di sistema aggiunte,* ***mantenendo sostanzialmente inalterati l'uso delle risorse e la logica di funzionamento.*** *Infatti la* `wait ABC` *in riga 1 è riposizionabile in riga 5, poiché per la condizione B basta impedire che alfa sia già terminato prima che beta esegua lo statement B o a maggior ragione prima che beta abbia passata la* `post ABC` *in riga 6. Ecco la soluzione con la variante introdotta (il resto è immodificato):*

```
pthread_mutex_t MUX
sem_t ABC, XYZ
int globale
void * alfa (...) {
    _____________________________________________ (1)
    pthread_mutex_lock (&MUX) _____________________ (2)
    _____________________________________________ (3)
    globale = 1                                   /* statement A */
    pthread_mutex_unlock (&MUX)
    sem_post (&XYZ) _______________________________ (4)
    sem_wait (&ABC) _______________________________ (5)
    return NULL
} /* end alfa */
void * beta (...) {
    pthread_mutex_lock (&MUX)                     /* statement B */
    sem_post (&ABC) _______________________________ (6)
    _____________________________________________ (7)
    globale = 2
    pthread_mutex_unlock (&MUX)
    return (void * 3)
} /* end beta */
void main ( ) {
    pthread_t th_1, th_2
    sem_init (&ABC, 0, 0)
    sem_init (&XYZ, 0, 0)
    pthread_create (&th_1, NULL, alfa, NULL)
    pthread_create (&th_2, NULL, beta, NULL)
    sem_wait (&XYZ) _______________________________ (8)
    pthread_join (th_2, &globale)
    printf ("%d", globale)                        /* statement C */
    pthread_join (th_1, NULL)
    return
} /* end main */
```

*Le righe 4 e 5 sono commutabili, ma di nuovo non è una modifica strutturale sostanziale. Invece la* `wait XYZ` *in riga 8 non è riposizionabile in riga 7: è vero che ciò soddisferebbe comunque la condizione C imponendo a beta di terminare solo dopo che alfa avesse eseguito* `globale = 1`*, ma potrebbe dare stallo. Infine la* `post XYZ` *in riga 6 non è riposizionabile in riga 1 o 3 poiché violerebbe la condizione C.*

## esercizio su thread e programmazione concorrente – martedì 26 novembre 2013

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 */
```

**Si consideri** la tabella seguente, la quale specifica parzialmente lo **stato** che il sistema concorrente dovrebbe assumere in tre **condizioni** differenti (A, B e C, indicate nel codice del sistema).

**Si completi** il codice dato a pagina precedente, secondo le modalità spiegate all'inizio, così da ottenere un sistema concorrente che **si comporti** precisamente **come specificato dalla tabella**, e **non abbia stalli**.

Si badi bene alla colonna "condizione": con "subito dopo statement X" si chiede lo stato che il sistema assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

| **condizione** | **stato del sistema concorrente** | |
|---|---|---|
| | *th_2* | *globale* |
| subito dopo stat. **A** | ESISTE | |
| subito dopo stat. **B** | | 0 |
| subito dopo stat. **C** | | 2 |

*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 per lo meno questa è una soluzione immediata (rispetto a usare i semafori) e oltretutto 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. 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 dove esattamente collocare una certa chiamata di sistema all'interno del codice dato o relativamente alle altre chiamate che vanno pure collocate.*

*Beninteso potrebbero esistere ragionamenti differenti da quello qui proposto e in parte preimpostato nel testo, conducenti a soluzioni diverse. Uno per esempio, consiste nell'osservare che il mutex non è indispensabile per realizzare il sistema, ma bastano i due semafori. Tuttavia le primitive del mutex sono date in parte e il mutex va completato. Un modo per farlo funzionare "a vuoto" e senza creare stalli, è questo:*

```
void * uno (void * arg) {

    sem_wait (&mah_X) ____________________________

    ____________________________________________

    globale = 1;                                        /* statement A */

    sem_post (&mah_Y) ____________________________

    pthread_mutex_lock (&boh) _____________________
    pthread_mutex_unlock (&boh)
    return NULL;
} /* end uno */

void * due (void * arg) {

    pthread_mutex_lock (&boh)                           /* statement B */

    sem_post (&mah_X) ____________________________

    sem_wait (&mah_Y) ____________________________

    globale = 2;                                        /* statement C */

    pthread_mutex_unlock (&boh) _____________

    ____________________________________________
    return NULL;
} /* end due */
```

*Si noti che la* `post` *di* `uno` *deve stare fuori dal mutex, altrimenti si potrebbe avere stallo. Si vede che il corpo del mutex di* `uno` *è vuoto, dunque il mutex non ha alcun impatto sul sistema; pertanto si potrebbe eliminare il mutex in entrambe le funzioni* `uno` *e* `due`*. I semafori basterebbero a garantire le proprietà richieste (si lascia la verifica al lettore). Si può considerare questa soluzione come un po' forzata rispetto al testo (nel senso di cambiare le risorse disponibili per la concorrenza rispetto alla soluzione precedente), ma corretta.*