Sincronizzazione dei processi

Programma – Sistemi Operativi

- Introduzione ai sistemi operativi
- Gestione dei processi
- Sincronizzazione dei processi
- Gestione della memoria centrale
- Gestione della memoria di massa
- File system
- Sicurezza e protezione

Processo cooperante

Un **processo cooperante** è un processo che:

- può influenzare un altro processo in esecuzione nel sistema
- può subire l'influenza di un altro processo in esecuzione nel sistema

I processi cooperanti possono condividere:

- uno spazio logico di indirizzi (codice e dati)
- solo dati attraverso file o messaggi

Race condition

- Una **race condition** si verifica quando i processi hanno accesso concorrente ai dati condivisi e il risultato finale dipende dal particolare ordine in cui si verificano gli accessi.

- Le **race condition** possono portare a valori corrotti dei dati condivisi.

Race condition

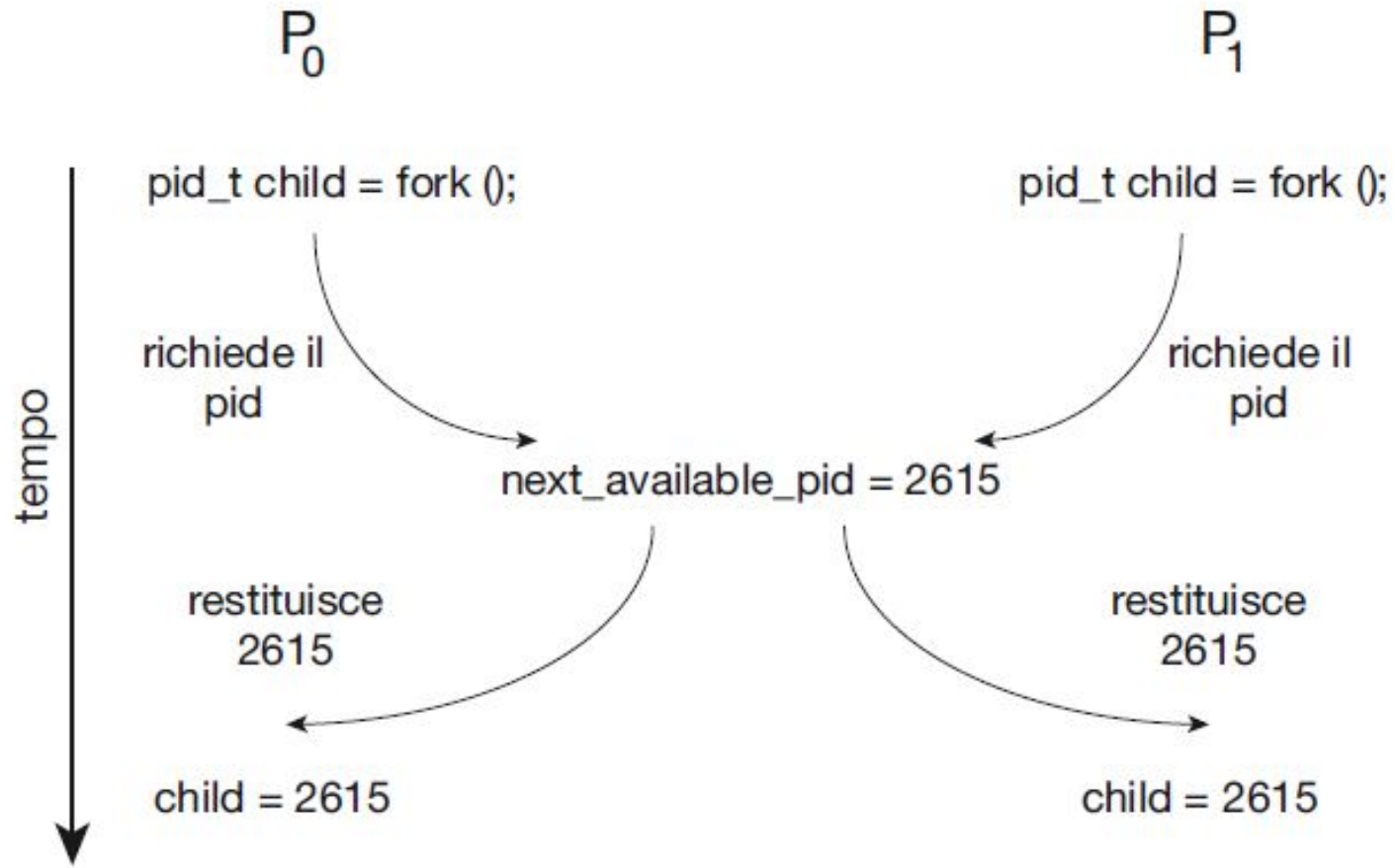


Figura 6.2 Race condition durante l'assegnamento del pid.

Race condition

Supponiamo di avere la seguente situazione:

Variabile condivisa `counter` che vale 5

Processo 1 (produttore)

```
counter++
```

Processo 2 (consumatore)

```
counter--
```

Se Processo 1 e Processo 2 sono in esecuzione concorrente, quale sarà il valore di `counter` al termine dell'esecuzione dei due processi?

Race condition

- **counter++** could be implemented as
  register1 = counter
  register1 = register1 + 1
  counter = register1
- **counter--** could be implemented as
  register2 = counter
  register2 = register2 - 1
  counter = register2
- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute **register1 = counter** {register1 = 5}
  S1: producer execute **register1 = register1 + 1** {register1 = 6}
  S2: consumer execute **register2 = counter** {register2 = 5}
  S3: consumer execute **register2 = register2 – 1** {register2 = 4}
  S4: producer execute **counter = register1** {counter = 6 }
  S5: consumer execute **counter = register2** {counter = 4}

Race condition

- How do we solve the race condition?
- We need to make sure that:
  - The execution of

    counter++

    is done as an "atomic" action. That is, while it is being executed, no other instruction can be executed concurrently.

    - actually no other instruction can access **counter**
  - Similarly for

    counter--
- The ability to execute an instruction, or a number of instructions, atomically is crucial for being able to solve many of the synchronization problems.

Sezione critica

Una **sezione critica (CS)** è una porzione di codice in cui i dati condivisi possono essere manipolati

Quando un processo è in esecuzione nella propria sezione critica, non si consente ad alcun altro processo di essere in esecuzione nella propria sezione critica

Figura 6.1 Struttura generale di un tipico processo.

Problema della sezione critica

- Il problema della **sezione critica** consiste nel progettare un protocollo che i processi possano usare per cooperare.

- Ogni processo deve chiedere il permesso per entrare nella propria **sezione critica.**

Soluzione al problema della sezione critica

Una soluzione del problema della sezione critica deve soddisfare i tre seguenti requisiti.

- Mutua esclusione
- Progresso
- Attesa limitata

Mutua esclusione

Se un processo P_i è in esecuzione nella propria sezione critica, nessun altro processo P_j può essere in esecuzione nella propria sezione critica

Progresso

Se nessun processo è in esecuzione nella propria sezione critica e qualche processo desidera entrare nella propria sezione critica, solo i processi che si trovano fuori dalle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica

Questa scelta non si può rimandare indefinitamente

Progresso – analisi passo per passo

Progress: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

If no process is executing in its critical section

If there is a process executing in its critical section (even though not stated explicitly, this includes the leave section as well), then this means that some work is getting done. So we are making progress. Otherwise, if this was not the case...

https://stackoverflow.com/questions/33143779/what-is-progress-and-bounded-waiting-in-critical-section

Progresso

and some processes wish to enter their critical sections

If no process wants to enter their critical sections, then there is no more work to do. Otherwise, if there is at least one process that wishes to enter its critical section...

then only those processes that are not executing in their remainder section

This means we are talking about those processes that are executing in either of the first two sections (remember, no process is executing in its critical section or the leave section)...

https://stackoverflow.com/questions/33143779/what-is-progress-and-bounded-waiting-in-critical-section

Progresso

can participate in deciding which will enter its critical section next,
Since there is at least one process that wishes to enter its CS, somehow we must choose one of them to enter its CS. But who's going to make this decision? Those process who already requested permission to enter their critical sections have the right to participate in making this decision. In addition, those processes that may wish to enter their CSs but have not yet requested the permission to do so (this means that they are in executing in the first section) also have the right to participate in making this decision.

https://stackoverflow.com/questions/33143779/what-is-progress-and-bounded-waiting-in-critical-section

Progresso

and this selection cannot be postponed indefinitely.

This states that it will take a limited amount of time to select a process to enter its CS. In particular, no deadlock or livelock will occur. So after this limited amount of time, a process will enter its CS and do some work, thereby making progress.

https://stackoverflow.com/questions/33143779/what-is-progress-and-bounded-waiting-in-critical-section

Attesa limitata

Se un processo P_i ha già richiesto l'ingresso nella propria sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accolga la richiesta di P_i

Gestione delle sezioni critiche

La due strategie principali per la gestione delle sezioni critiche nei sistemi operativi sono:

1. **kernel con diritto di prelazione**

2. **kernel senza diritto di prelazione**

Kernel con diritto di prelazione

kernel con diritto di prelazione

Consente che un processo funzionante in modalità di sistema sia sottoposto a prelazione, rinviandone in tal modo l'esecuzione. NON è immune da race condition

Kernel senza diritto di prelazione

kernel senza diritto di prelazione → non consente di applicare la prelazione a un processo attivo in modalità di sistema. È immune da race condition

Soluzione di Peterson

- La soluzione di Peterson è una soluzione software al problema della sezione critica

- È limitata a due processi, P_0 e P_1, ognuno dei quali esegue alternativamente la propria sezione critica e la sezione non critica.

- P_0 e P_1 condividono
  `int turn;`
  `boolean flag[2];`

```
while (true) {

    flag[i]= true;
    turn = j;
    while (flag[j] && turn == j);

       /*sezione critica*/

    flag[i] = false;

       /*sezione non critica*/

}
```

Figura 6.3 Struttura del processo P_i nella soluzione di Peterson.

indica se un processo è pronto a entrare in CS

Riordino delle istruzioni

Variabili globali

```
boolean flag = false;
int x = 0;
```

Non ci sono dipendenze tra le variabili `flag` e `x`

Thread 1

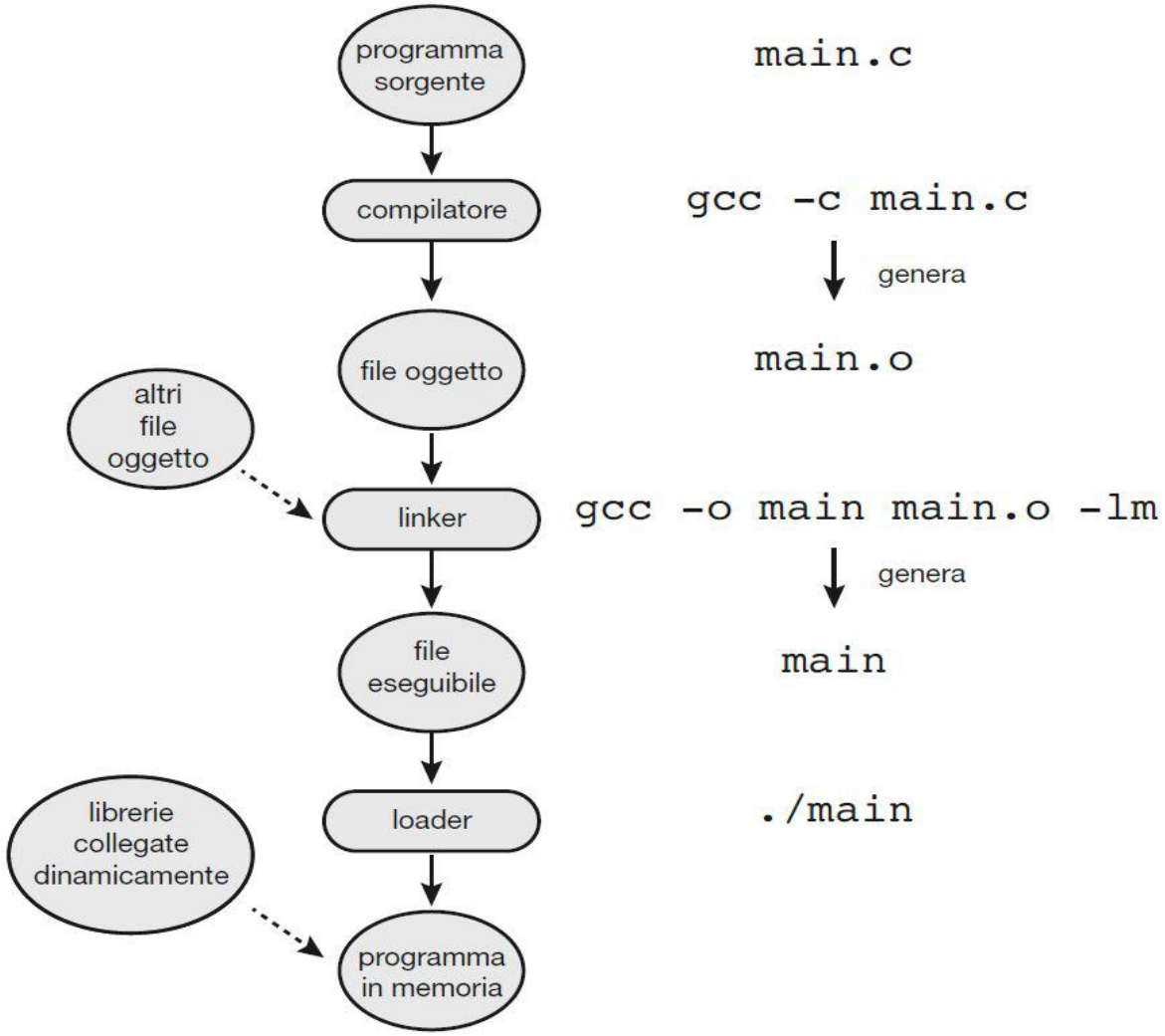
```
while(!flag)
    ;
print x;
```

Thread 2

```
x = 100;
flag = true;
```

Cosa viene stampato da Thread 1?

Riordino delle istruzioni



Soluzione di Peterson

Si consideri ciò che accadrebbe se gli assegnamenti che compaiono nella sezione d'ingresso della soluzione di Peterson della Figura 6.3 venissero riordinati.
In questo caso sarebbe possibile avere entrambi i thread attivi nelle loro sezioni critiche contemporaneamente, come mostrato nella Figura 6.4.

```
while (true) {

    flag[i]= true;
    turn = j;
    while (flag[j] && turn == j);

      /*sezione critica*/

    flag[i] = false;

      /*sezione non critica*/

}
```

Figura 6.3 Struttura del processo P_i nella soluzione di Peterson.

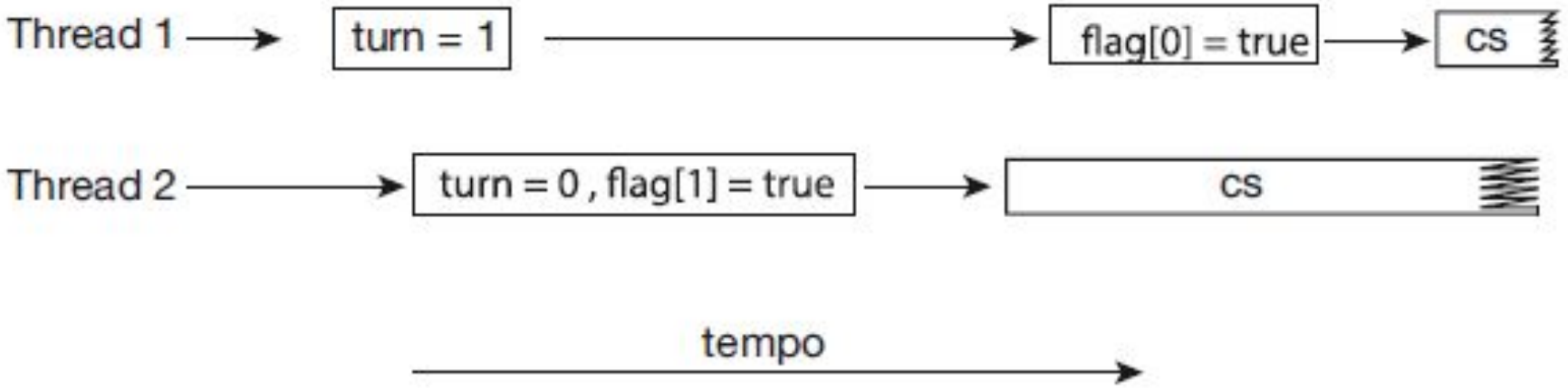


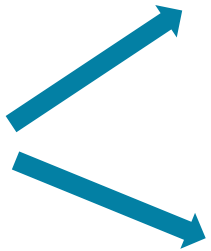
Figura 6.4 Effetti del riordino delle istruzioni nella soluzione di Peterson.

Supporto hardware per la sincronizzazione

Soluzioni basate sul software (come quella di Peterson) non garantiscono il loro funzionamento su architetture elaborative moderne

Modello di memoria

un modello di memoria rientra in una delle due categorie



Fortemente ordinato una modifica alla memoria su un processore è immediatamente visibile a tutti gli altri processori

Debolmente ordinato le modifiche alla memoria su un processore potrebbero non essere immediatamente visibili agli altri processori

Barriere di memoria

barriere di memoria (*memory barrier)* o **recinzioni di memoria *(memory fence)***

➡ assicurano che le operazioni di `store` siano completate e visibili ad altri processori prima che vengano eseguite le operazioni di `load` e `store` future

Barriere di memoria

Variabili globali

```
boolean flag = false;
int x = 0;
```

Non ci sono dipendenze tra le variabili `flag` e `x`

Thread 1

```
while(!flag)
    memory barrier();
print x;
```

Thread 2

```
x = 100;
memory barrier();
flag = true;
```

Cosa viene stampato da Thread 1?

Istruzioni hardware

Molte delle moderne architetture offrono particolari istruzioni che permettono di controllare e modificare il contenuto di una parola di memoria, oppure di scambiare il contenuto di due parole di memoria, in modo **atomico** – cioè come un'unità non interrompibile.

test and set

```
boolean test_and_set(boolean *obiettivo){
    boolean valore = *obiettivo;
    *obiettivo = true;

    return valore;
}
```

Figura 6.5 Definizione dell'istruzione atomica `test_and_set()`.

Istruzioni hardware per CS

Le istruzioni hardware sono utilizzabili per risolvere il problema della sezione critica in modo relativamente semplice

`lock` iniz
`false`

```
do {
  while (test_and_set(&lock))
    ; /*non fa niente*/

  /*sezione critica*/

  lock = false;

  /*sezione non critica*/
} while (true);
```

Figura 6.6 Realizzazione di mutua esclusione con `test_and_set()`.

compare and swap()

```
int compare_and_swap(int *value, int expected, int new_value) {
  int temp = *value;

  if (*value == expected)
    *value = new_value;

  return temp;
}
```

Figura 6.7 Definizione dell'istruzione atomica `compare_and_swap()`.

CAS viene eseguita atomicamente. Se due istruzioni CAS vengono schedulate simultaneamente (ciascuna su un core diverso), verranno eseguite sequenzialmente in ordine arbitrario.

compare and swap()

`lock` iniz

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
          ; /* non fa niente */

   /* sezione critica */

   lock = 0;

   /* sezione non critica */
}
```

Figura 6.8 Realizzazione di mutua esclusione con `compare_and_swap()`.

Questo algoritmo soddisfa il requisito della mutua esclusione, ma non quello dell'attesa limitata

`compare and swap()` con attesa limitata

Variabili globali

```
boolean waiting[n];
boolean lock;
```

`lock` inizializzata a `0`

tutti gli elementi di `waiting` inizializzati a `false`

```
while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key ==1)
       key = compare_and_swap(&lock, 0, 1);
    waiting[i] = false;

       /* sezione critica */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
       j = (j + 1) % n;

    if (j == i)
       lock = 0;
    else
       waiting[j] = false;

       /* sezione non critica */
}
```

Figura 6.9 Mutua esclusione con attesa limitata con `compare_and_swap()`.

Variabile atomica

variabile atomica → fornisce operazioni atomiche su tipi di dati di base come interi e booleani.

Le variabili atomiche possono essere utilizzate per garantire la mutua esclusione in situazioni in cui potrebbe esserci una race condition su una singola variabile durante il suo aggiornamento

Lock mutex

strumenti più robusti che risolvono le race condition

lock mutex

protegge le regioni critiche e quindi prevenire le race condition

Il termine mutex deriva da **mut**ual **ex**clusion

```
while (true) {

    acquisisci lock

        sezione critica

    rilascia lock

        sezione non critica

}
```

Figura 6.10 Soluzione al problema della sezione critica con lock mutex.

Lock mutex

```
acquisisci_lock() {
   while(!disponibile)
           ; /* busy wait */
   disponibile = false;
}


rilascia_lock() {
    disponibile = true;
}
```

I lock mutex generano busy waiting, per questo sono anche detti spinlock

Busy waiting

Si verifica una situazione di attesa attiva (“busy waiting”) quando un processo che vorrebbe entrare nella propria sezione critica è costretto a ciclare continuamente richiedendo
“posso accedere alla mia sezione critica?”

Il busy waiting comporta uno speco di cicli di CPU

Vantaggio degli spinlock

Gli spinlock hanno il vantaggio di non rendere necessario alcun context switch (che richiede un considerevole aggravio in termini di costi di esecuzione) quando un processo deve attendere un lock.

Sono quindi molto utili quando si prevede che i lock saranno trattenuti per tempi brevi.

Semafori

semafori 

uno strumento più robusto in grado di comportarsi in modo simile a un lock mutex,
ma capace anche di fornire metodi più complessi per la sincronizzazione delle attività dei processi

Semafori

Un **semaforo S** è una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: wait() e signal().

```
wait(S) {
     while(S <= 0)
          ; /* busy wait */
     S--;
}
```

```
signal(S) {
     S++;
}
```

Uso dei semafori

Can solve various synchronization problems

- A solution to the CS problem.
  - Create a semaphore "**synch**" initialized to 1

```
wait(synch)
    CS
signal(synch);
```

- Consider P_1 and P_2 that require code segment S_1 to happen before code segment S_2
  - Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Semafori contatore

semafori
contatore

trovano applicazione nel controllo dell’accesso a una data risorsa presente in un numero finito di esemplari.

Semafori binari

semafori binari

sono simili ai **lock mutex** e vengono utilizzati al loro posto per la mutua esclusione nei sistemi dove i lock mutex non sono disponibili

Semafori

Benché i *semafori* costituiscano un meccanismo pratico ed efficace per la sincronizzazione dei processi, il loro uso scorretto può generare errori difficili da individuare

Monitor

Una strategia per gestire possibili errori nell'uso dei semafori consiste nell'incorporare semplici strumenti di sincronizzazione in costrutti linguistici di alto livello: i **monitor**

```
monitor monitor name
{
      /* dichiarazione di variabili condivise */

      function P1 ( . . . ) {
            . . .
      }
      function P2 ( . . . ) {
            . . .
      }
                  .
                  .
                  .
      function Pn ( . . . ) {
            . . .
      }
      initialization_code ( . . . ) {
            . . .
      }
}
```

Figura 6.11 Sintassi di un monitor in pseudocodice.

Monitor

Il **costrutto monitor** assicura che all'interno di un monitor possa essere attivo un solo processo alla volta, in modo tale che non si debba codificare esplicitamente il vincolo di mutua esclusione (Figura 6.12)

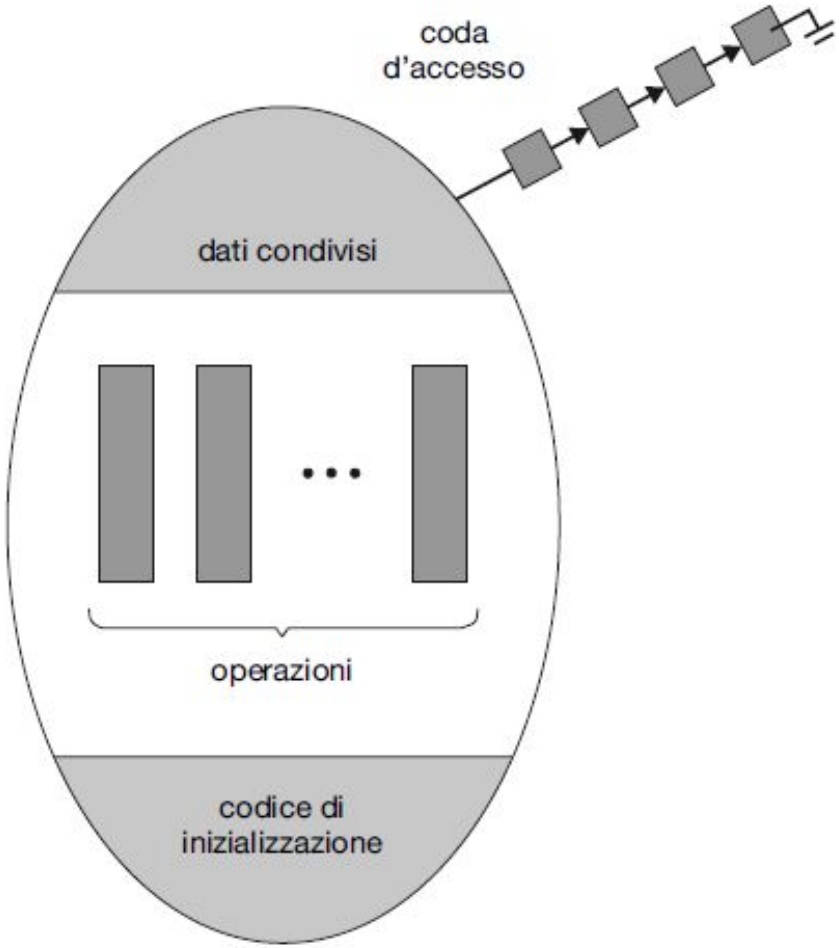


Figura 6.12 Schema di un monitor.

Monitor

Un programmatore che necessiti di implementare un proprio particolare schema di sincronizzazione può definire una o più **variabili di tipo *condition* *(condizionali):***

```
condition x, y;
```

Un monitor utilizza variabili condizionali per consentire ai processi di attendere che si verifichino determinate condizioni e di segnalarsi reciprocamente quando ciò avviene

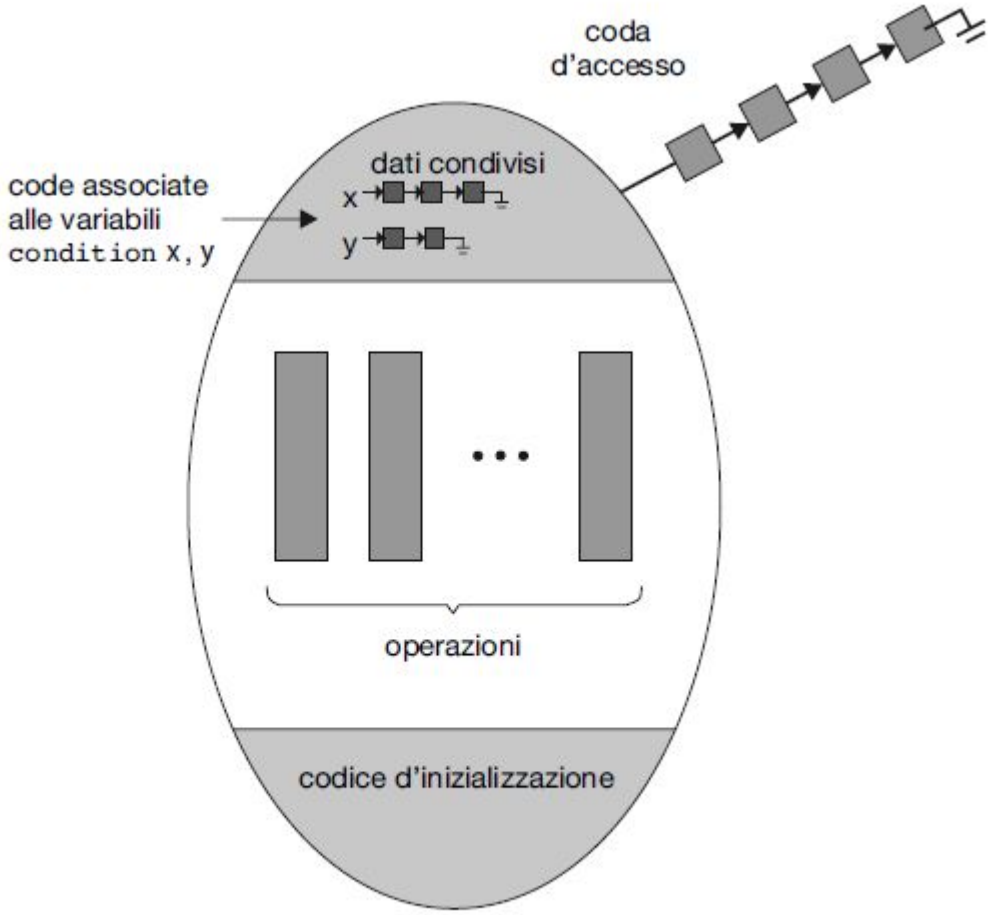


Figura 6.13 Monitor con variabili condition.

Liveness

Il termine **liveness** fa riferimento a un insieme di proprietà che un sistema deve soddisfare per garantire che i processi facciano progressi durante il ciclo di vita della loro esecuzione.

Un processo che attende *indefinitamente* è un esempio di “**mancanza di liveness**” (*liveness failure*).

Liveness

Due situazioni possono portare a mancanza di *liveness*:

- stallo o deadlock
- inversione di priorità

Deadlock

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Starvation

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended

La starvation può essere causata per esempio dall'uso di una lista LIFO associata ad un semaforo

Inversione di priorità

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Mars Pathfinder

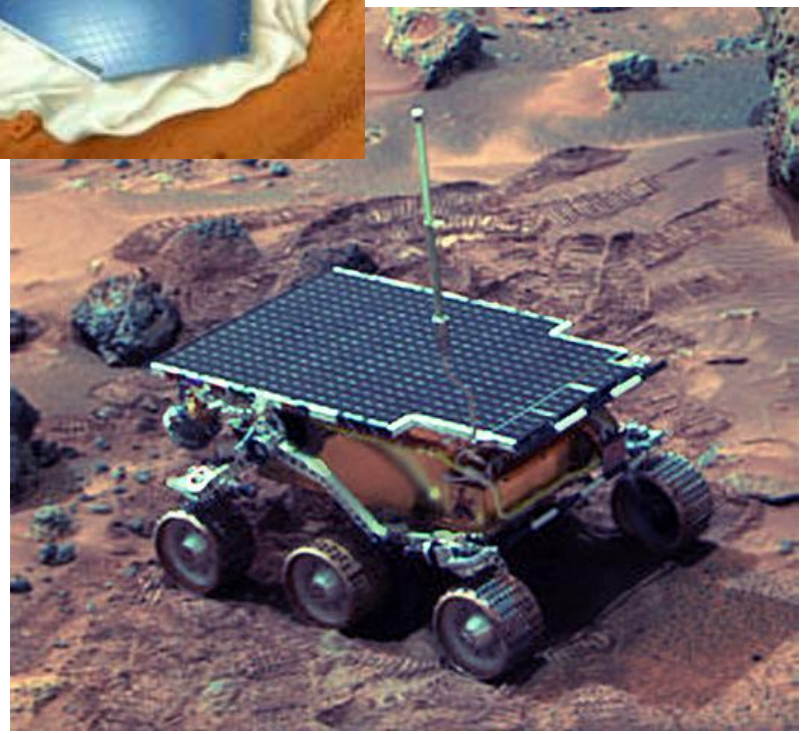
Mars Pathfinder è stata una missione scientifica della NASA per l'esplorazione di Marte

- prima missione ad aver trasportato un rover, Sojourner, sul pianeta.

- Lancio: 4 dicembre 1996
- Arrivo: 4 luglio 1997 (7 mesi dopo)

- Il lander operò come una stazione meteorologica

- Il rover analizzò il suolo e le rocce del sito di atterraggio ed effettuò diversi esperimenti sulla superficie.

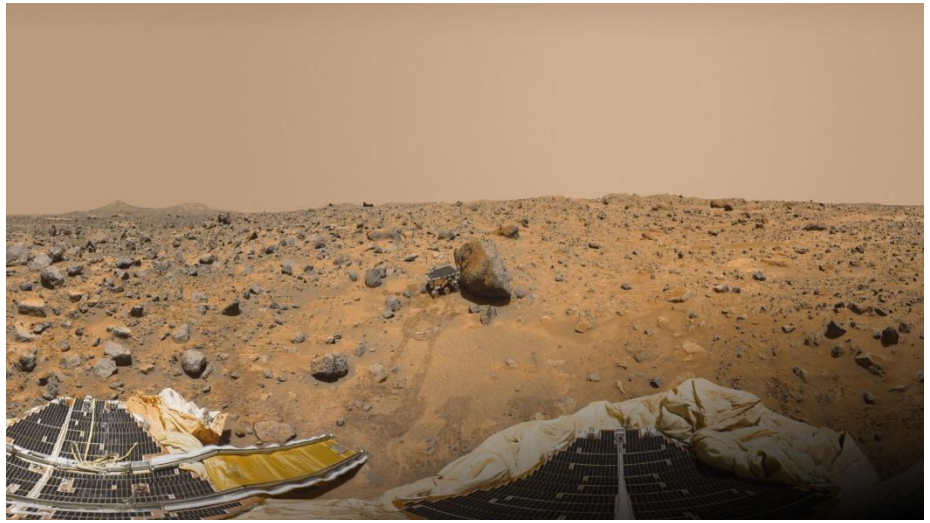


https://space.skyrocket.de/doc_sdat/mars_pathfinder.htm

https://mrbit.me/Pathfinder-Priority-Inversion/

er

- Iniziano i test scientifici, ma subito c'e` un problema:
  - **Il software del Sojourner compie un reset !! Piu` volte di seguito !!**
  - Il reset fa ripartire correttamente il software, ma nessuno riesce a capire come mai, e come far smettere i continui reset
  - I dati non vanno perduti nel reset, ma bisogna rimandare gli esperimenti al "sol" (giorno marziano) successivo – per sempre?



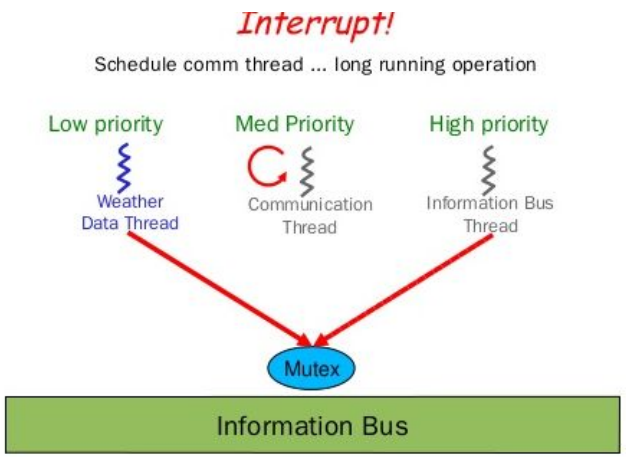
- La raccolta dati dal bus era affidata a due task:
  - Il gestore delle transizioni del bus (bc_sched)
  - Il responsabile di acquisizione dati (bc_dist)
- Una sequenza di operazioni corrette richiedeva che i due task si alternassero in esecuzione, durante ogni ciclo di 8Hz
  - Quando un task iniziava, per prima cosa controllava che l'altro task avesse finito; altrimenti, generava un **reset del sistema**.

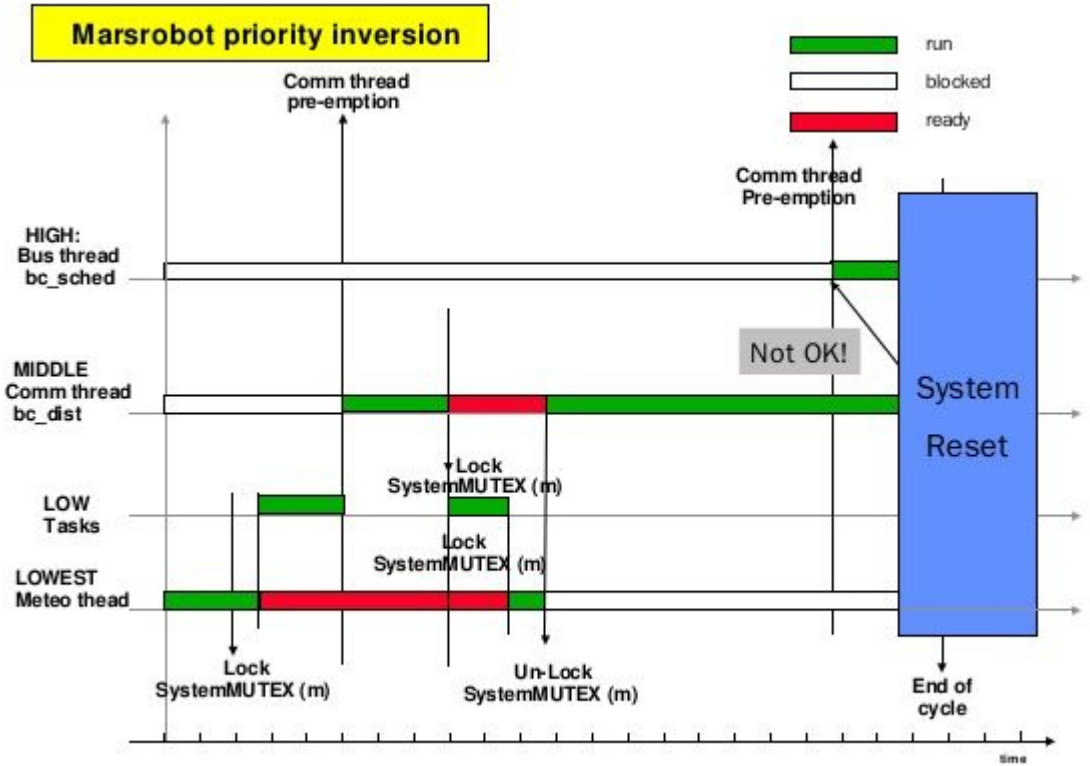
Il problema riguardava tre task:

- bc_dist (priorita` 3),
- Un qualsiasi task relativo alle operazioni della stazione spaziale (priorita` 4), e
- Il task della stazione meteo (priorita` 5)

Il task della stazione meteo riceveva i dati da bc_sched, per fare questo, doveva prendere in uso esclusivo (lock) una risorsa, cioe` il bus.

- La sua esecuzione veniva sospesa dal task con priorita` 4, prima che rilasciasse il lock,
- Successivamente, bc_dist non riusciva ad acquisire quella risorsa, e forzava il reset.





https://www.slideshare.net/gabrielladodero/quando-i-computer-non-funzionano-su-marte

Livello di contesa

I vari strumenti che possono essere utilizzati per risolvere il problema della sezione critica e per sincronizzare l'attività dei processi possono essere valutati a seconda del livello di contesa.

Alcuni strumenti funzionano meglio con un certo livello di contesa rispetto ad altri.

Livello di contesa

Le seguenti linee guida identificano delle regole generali per distinguere le prestazioni della sincronizzazione basata su `compare and swap()` (CAS) e della sincronizzazione tradizionale (come i lock mutex e i semafori) al variare del livello di contesa:

Nessuna contesa

Contesa moderata

Alta contesa

Valutazione delle soluzioni

- **Nessuna contesa.** Sebbene entrambe le opzioni siano generalmente veloci, la protezione CAS sarà leggermente più veloce della sincronizzazione tradizionale.

- **Contesa moderata.** La protezione CAS sarà più veloce e in alcuni casi molto più veloce rispetto alla sincronizzazione tradizionale.

- **Alta contesa.** Con carichi molto elevati, la sincronizzazione tradizionale sarà in definitiva più veloce della sincronizzazione basata su CAS.

Risorse

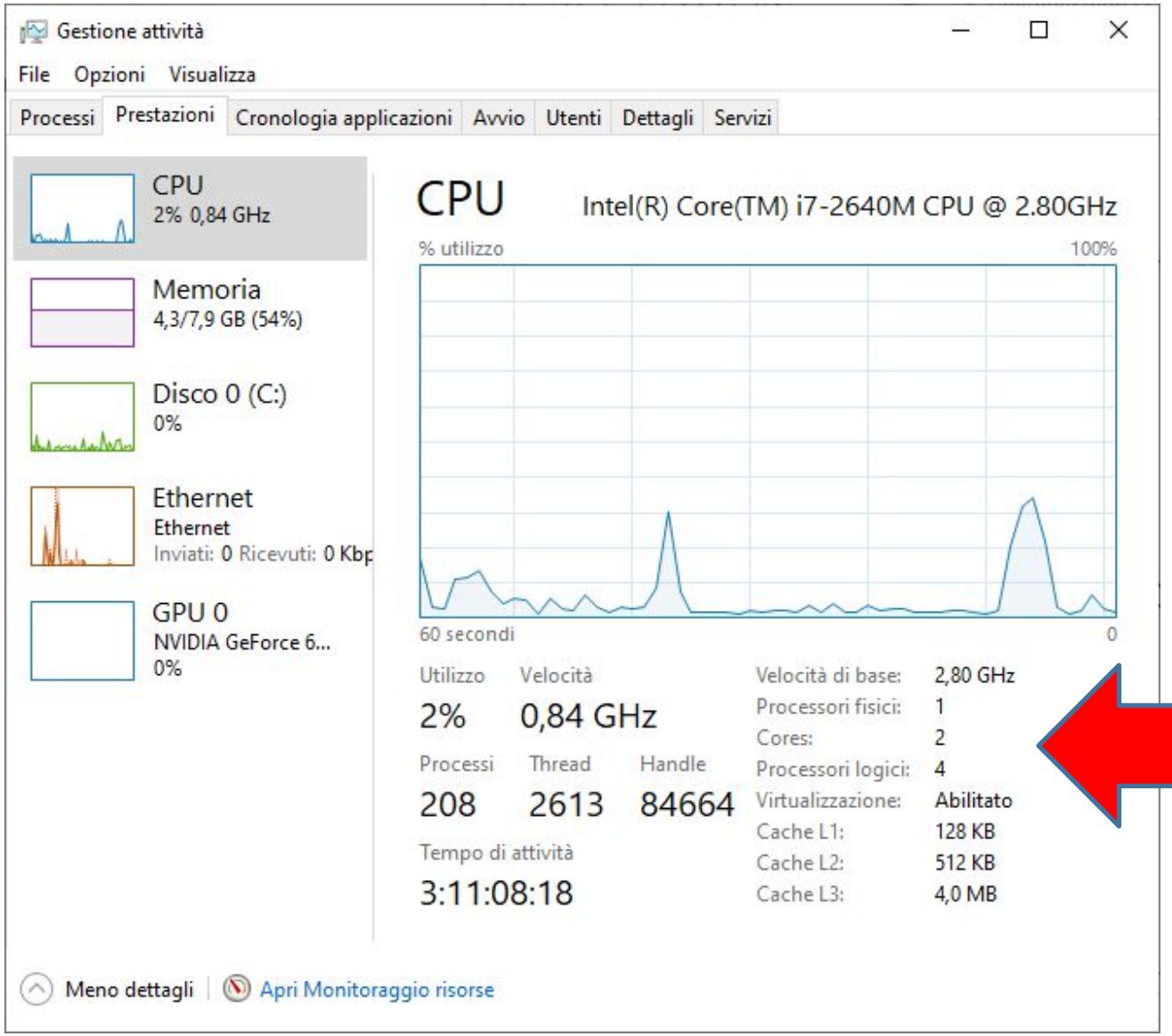
Un **sistema** è composto da un numero finito di risorse.

Le risorse possono essere raggruppate in **classi** differenti, ciascuna formata da un numero *n* di **istanze** identiche.

Per esempio, se il sistema ha 4 CPU allora la classe di risorse CPU avrà 4 istanze.

Se un thread richiede l'assegnazione di una risorsa di classe *C*, allora qualunque istanza di *C* dovrebbe soddisfare la richiesta.

Risorse - Esempio



Risorse

In un ambiente con multiprogrammazione più thread possono competere per ottenere una risorsa

Un thread può servirsi di una risorsa soltanto se rispetta la seguente sequenza:

Richiesta - Uso - Rilascio

Richiesta: il thread richiede la risorsa. Se la richiesta non si può soddisfare, allora il thread attende fino a chè non sarà possibile acquisire tale risorsa.

Uso: il thread opera sulla risorsa.

Rilascio: il thread rilascia la risorsa.

Richiesta - Uso - Rilascio

Esempi di chiamate di sistema per richiesta/rilascio di una risorsa:

- `request()/release()` per una periferica
- `open()/close()` per un file
- `allocate()/free()` per una porzione di memoria
- `wait()/signal()` per i semafori
- `acquire()/release()` per i lock mutex

anche lock mutex e semafori sono risorse di sistema!

Stallo

Se le risorse richieste da un thread *T* sono trattenute da altri thread, a loro volta nello stato di attesa, il thread *T* potrebbe non cambiare più il suo stato.

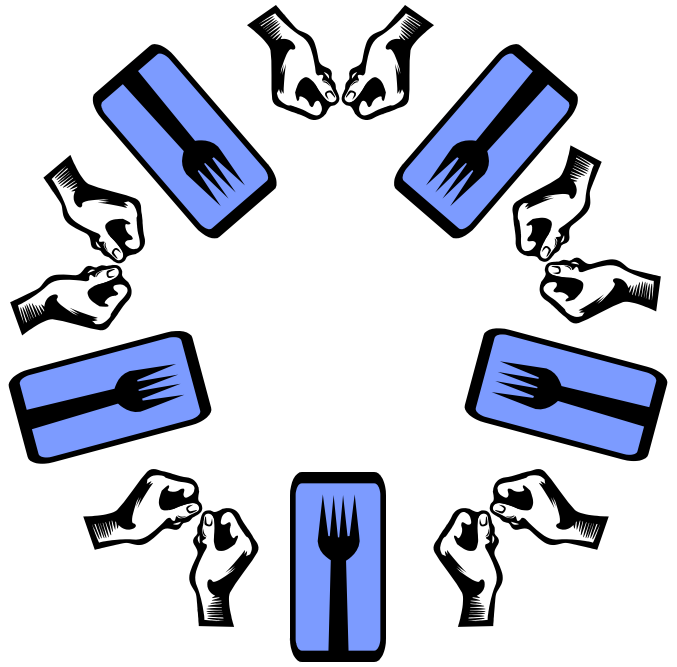
Situazioni di questo tipo sono chiamate di stallo (deadlock)

Esempio di stallo

Implementazione non corretta del problema dei filosofi a cena

```
while (true) {
     wait(chopstick[i]);
     wait(chopstick[(i+1) % 5]);
          . . .
     /* mangia */
          . . .
     signal(chopstick[i]);
     signal(chopstick[(i+1) % 5]);
          . . .
     /* pensa */
          . . .
}
```

Figura 7.6 Struttura del filosofo *i*.

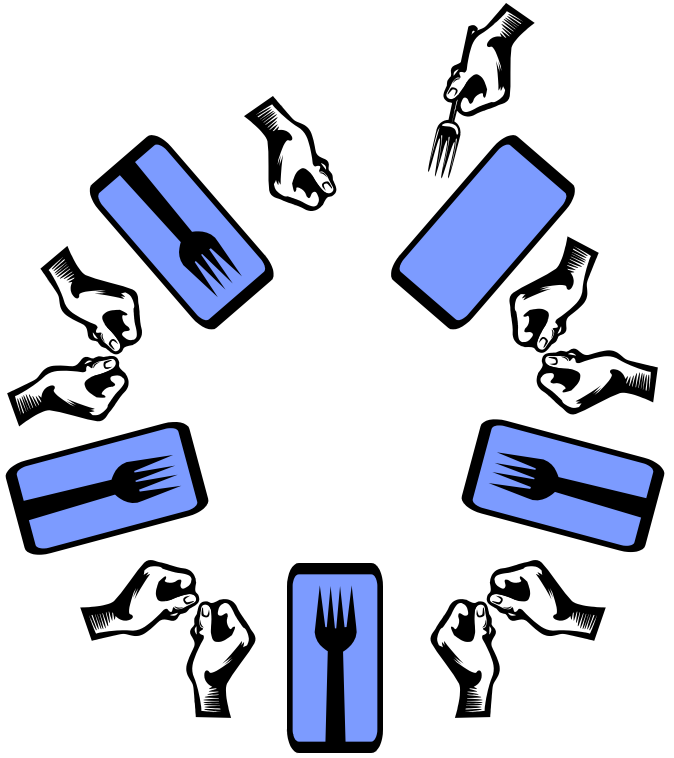


https://courses.engr.illinois.edu/cs241/fa2011/lectures/final/17-ClassicSynch.ppt

Esecuzione

```
while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        . . .
    /* mangia */
        . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        . . .
    /* pensa */
        . . .
}
```

Figura 7.6 Struttura del filosofo *i*.

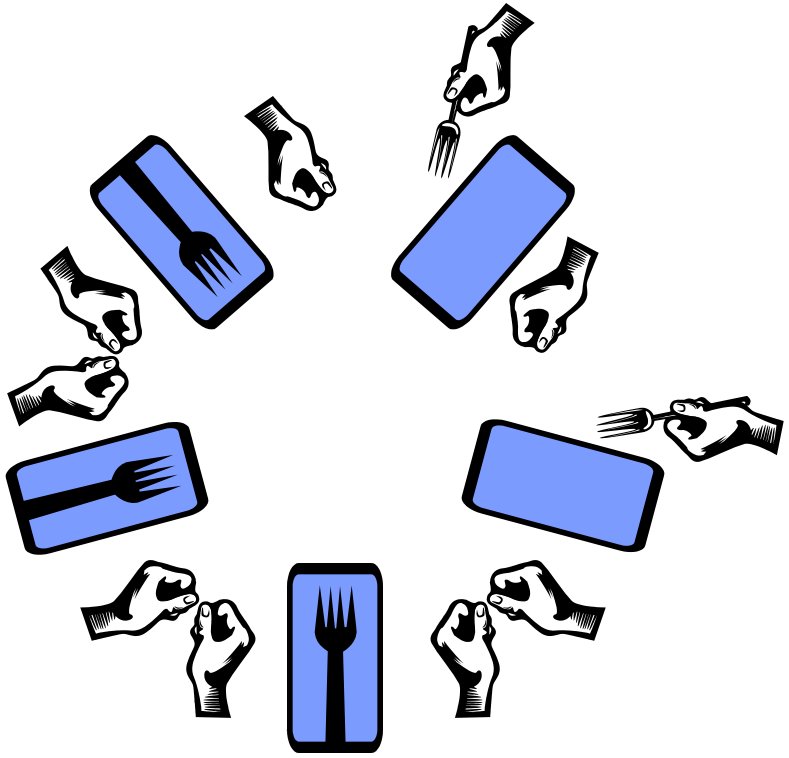


https://courses.engr.illinois.edu/cs241/fa2011/lectures/final/17-ClassicSynch.ppt

Esecuzione

```
while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        . . .
    /* mangia */
        . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        . . .
    /* pensa */
        . . .
}
```

Figura 7.6 Struttura del filosofo *i*.

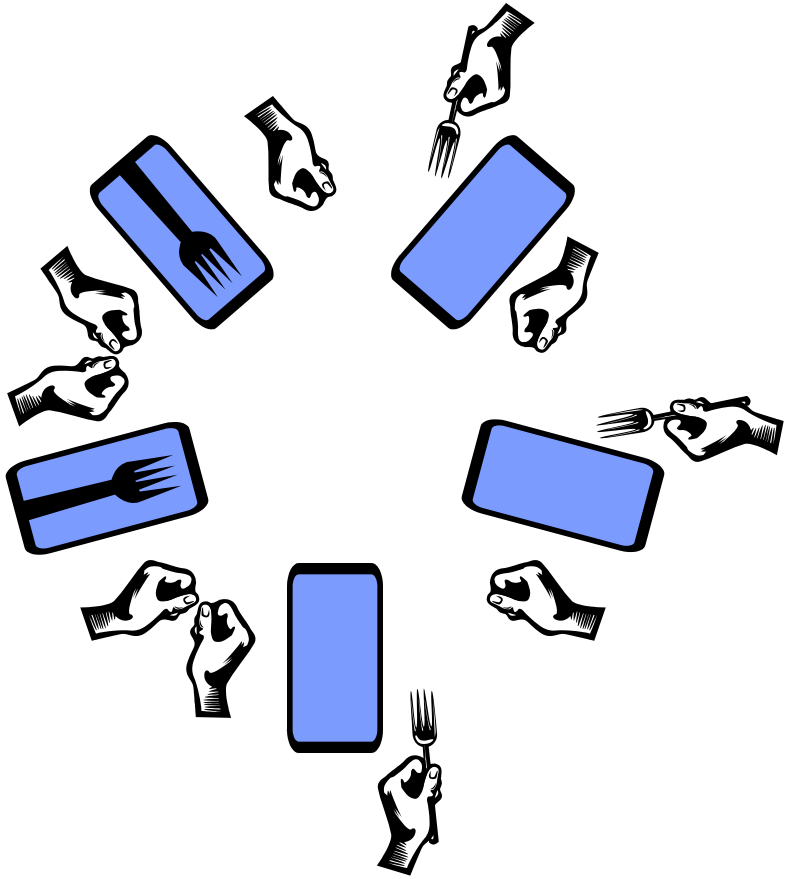


https://courses.engr.illinois.edu/cs241/fa2011/lectures/final/17-ClassicSynch.ppt

Esecuzione

```
while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        . . .
    /* mangia */
        . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        . . .
    /* pensa */
        . . .
}
```

Figura 7.6 Struttura del filosofo *i*.

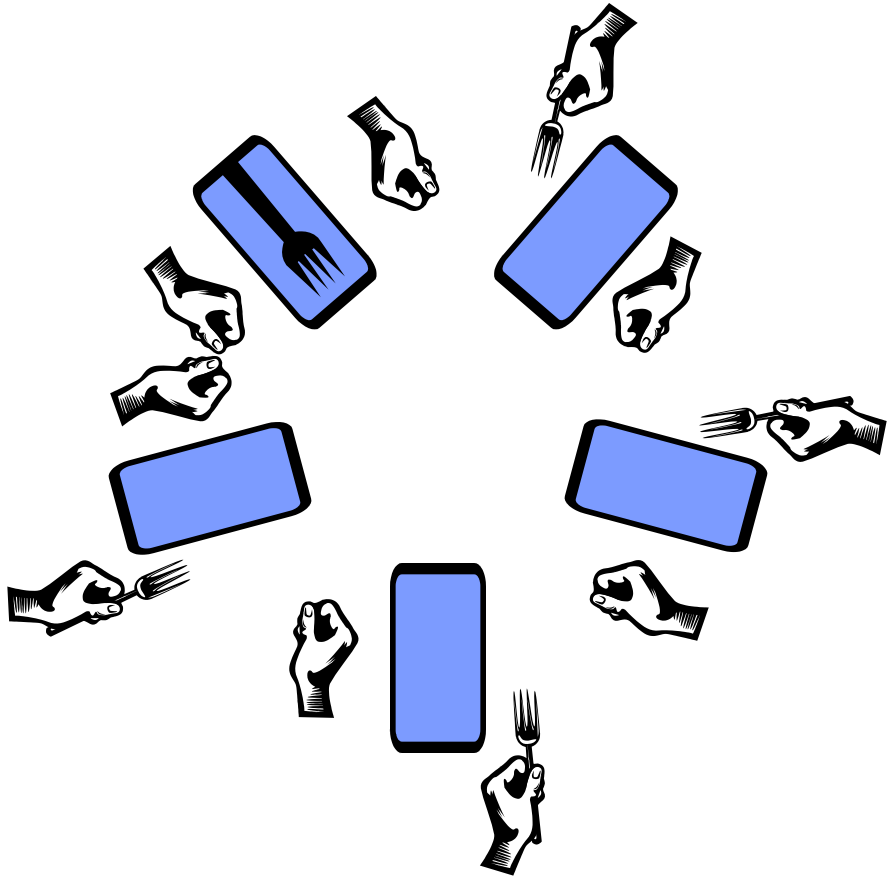


https://courses.engr.illinois.edu/cs241/fa2011/lectures/final/17-ClassicSynch.ppt

Esecuzione

```
while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        . . .
    /* mangia */
        . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        . . .
    /* pensa */
        . . .
}
```

Figura 7.6 Struttura del filosofo *i*.

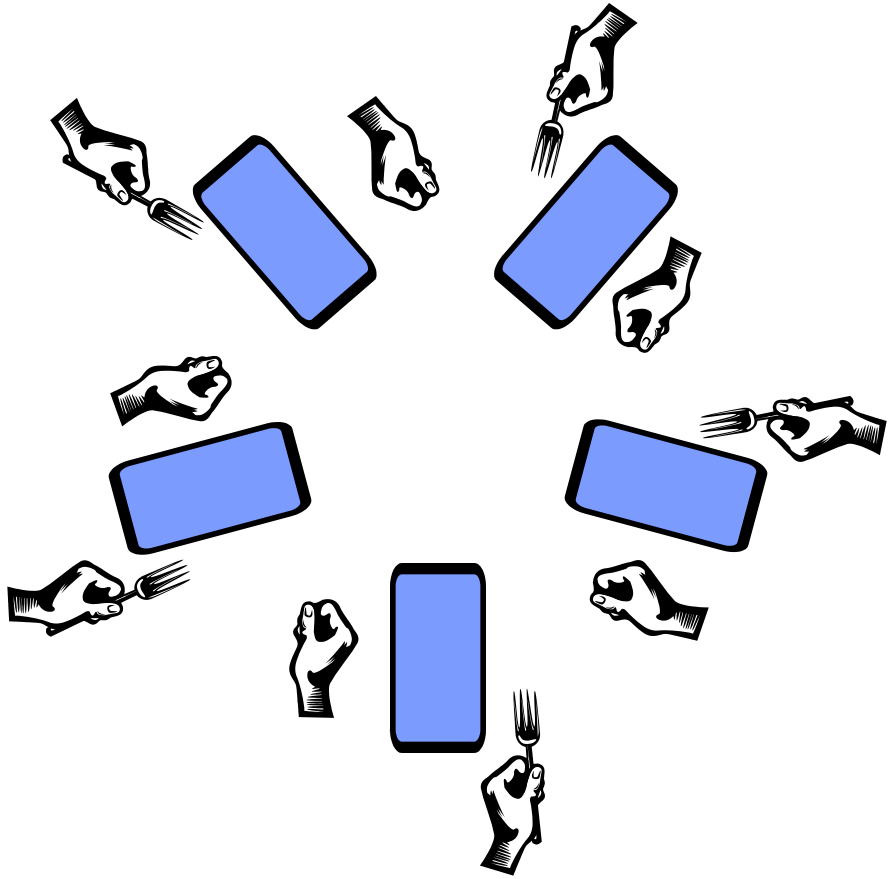


https://courses.engr.illinois.edu/cs241/fa2011/lectures/final/17-ClassicSynch.ppt

Esecuzione

```
while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        . . .
    /* mangia */
        . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        . . .
    /* pensa */
        . . .
}
```

Figura 7.6 Struttura del filosofo *i*.

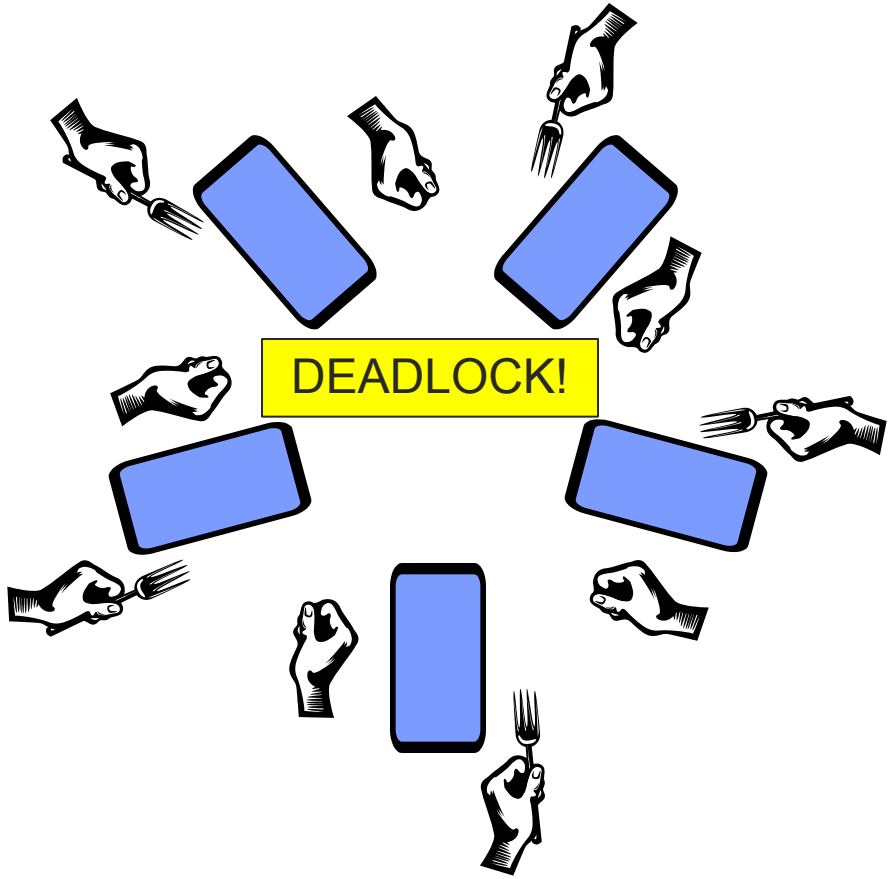


https://courses.engr.illinois.edu/cs241/fa2011/lectures/final/17-ClassicSynch.ppt

Esecuzione

```
while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        . . .
    /* mangia */
        . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        . . .
    /* pensa */
        . . .
}
```

Figura 7.6 Struttura del filosofo *i*.



https://courses.engr.illinois.edu/cs241/fa2011/lectures/final/17-ClassicSynch.ppt

Esempio di stallo in applicazioni multithread

Inizializzazione:

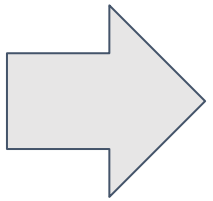
```
pthread_mutex t first_mutex;
pthread_mutex t second mutex;

phtread_mutex init(&first_mutex,NULL);
phtread_mutex init(&second_mutex,NULL);
```

```
/* thread_one esegue in questa funzione */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
      * Fa qualcosa
      */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two esegue in questa funzione */
void *do_work_two(void *param)
{
   pthread_mutex_lock (&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
      * Fa qualcosa
      */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

Figura 8.1 Esempio di stallo.

Codice completo

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex t first_mutex;
pthread_mutex t second_mutex;

void *do work one(void *param)
{
  pthread_mutex lock(&first_mutex);
  pthread_mutex lock(&second_mutex);
  printf("doing work one\n");
  pthread_mutex unlock(&second_mutex);
  pthread_mutex unlock(&first_mutex);
  pthread exit(0);
}

void *do work two(void *param)
{
  pthread_mutex lock(&second_mutex);
  pthread_mutex lock(&first_mutex);
  printf("doing work two\n");
  pthread_mutex unlock(&first_mutex);
  pthread_mutex unlock(&second_mutex);
  pthread exit(0);
}
```



https://github.com/dbloisi/3.4-stallo-dei-processi

Codice completo

```
nt main()

 pthread t thread one, thread two;

 pthread_mutex init(&first_mutex, NULL);
 pthread_mutex init(&second_mutex, NULL);

 if(pthread create(&thread one, NULL, do work one, NULL) < 0)
 {
    printf("errore creazione thread one\n");
    exit(1);
 }

 if(pthread create(&thread two, NULL, do work two, NULL) < 0)
 {
    printf("errore creazione thread two\n");
    exit(1);
 }

 pthread join (thread one, NULL);
 pthread join (thread two, NULL);
 return 0;
}
```

https://github.com/dbloisi/3.4-stallo-dei-processi

Esecuzione

https://github.com/dbloisi/3.4-stallo-dei-processi

Debug

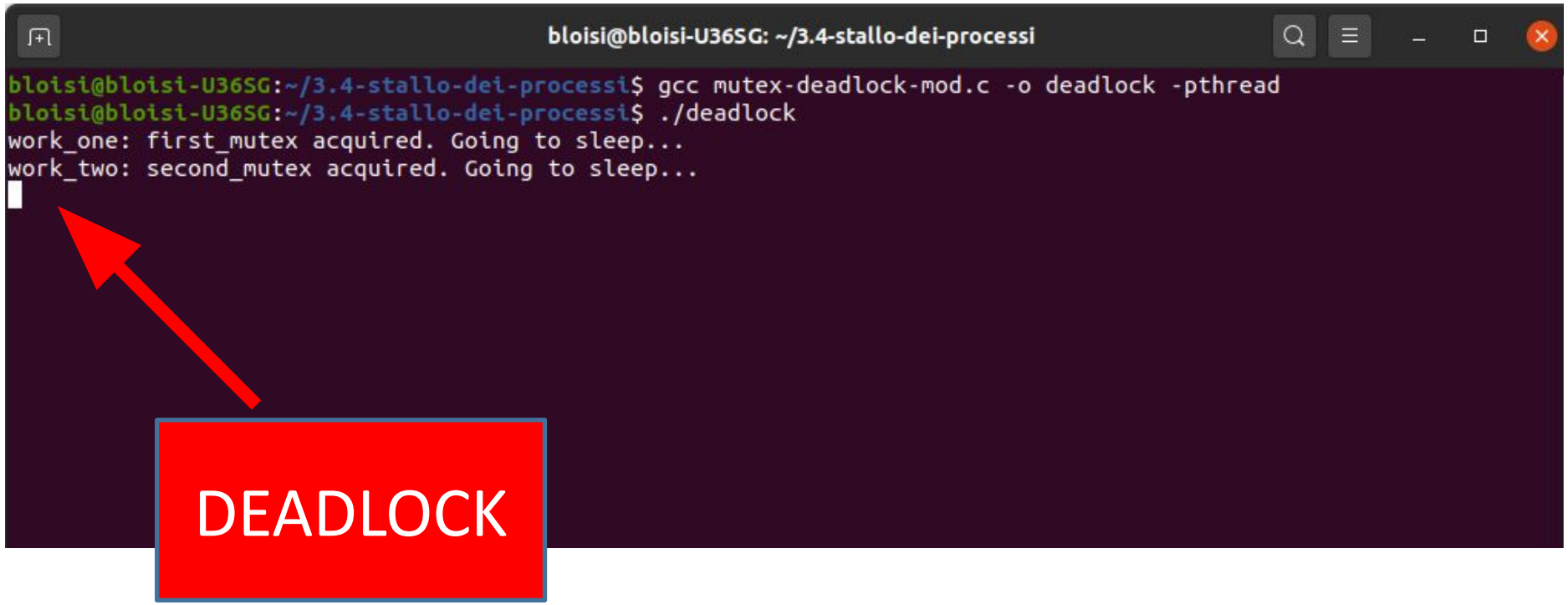
E’ difficile identificare e sottoporre a test gli stalli che si verificano solo in determinate condizioni di scheduling

Modifichiam l'esempio d stallo in applicazioni multithread

```
void *do_work_one(void *param)
{
  pthread_mutex_lock(&first_mutex);
  printf("work_one: first_mutex acquired. Going to sleep...\n");
  sleep(1);
  pthread_mutex_lock(&second_mutex);
  printf("doing work one\n");
  pthread_mutex_unlock(&second_mutex);
  pthread_mutex_unlock(&first_mutex);
  pthread_exit(0);
}

void *do_work_two(void *param)
{
  pthread_mutex_lock(&second_mutex);
  printf("work_two: second_mutex acquired. Going to sleep...\n");
  sleep(1);
  pthread_mutex_lock(&first_mutex);
  printf("doing work two\n");
  pthread_mutex_unlock(&first_mutex);
  pthread_mutex_unlock(&second_mutex);
  pthread_exit(0);
}
```

https://github.com/dbloisi/3.4-stallo-dei-processi/blob/master/mutex-deadlock-mod.c

Esecuzione Esempio Modificato



https://github.com/dbloisi/3.4-stallo-dei-processi/blob/master/mutex-deadlock-mod.c

Stallo attivo (livelock)

Lo stallo attivo o livelock si verifica quando un thread tenta continuamente un'azione che non ha successo.

Il livelock è meno comune del deadlock, ma è comunque un problema complesso nella progettazione di applicazioni concorrenti e, come il deadlock, può verificarsi solo in determinate condizioni di scheduling.

Stallo attivo (livelock)

```
void *do work one(void *param)
{
  int done = 0;

  while(!done) {
    pthread_mutex lock(&first_mutex);
    if(pthread_mutex trylock(&second_mutex))
    {
      printf("doing work one\n");
      pthread_mutex unlock(&second_mutex);
      pthread_mutex unlock(&first_mutex);
      done = 1;
    }
    else
      pthread_mutex unlock(&first_mutex);
  }
  pthread_exit(0);
}
```

```
void *do work two(void *param)
{
  int done = 0;

  while(!done) {
    pthread_mutex lock(&second_mutex);
    if(pthread_mutex trylock(&first_mutex))
    {
      printf("doing work two\n");
      pthread_mutex unlock(&first_mutex);
      pthread_mutex unlock(&second_mutex);
      done = 1;
    }
    else
      pthread_mutex unlock(&second_mutex);
  }
  pthread_exit(0);
}
```

https://github.com/dbloisi/3.4-stallo-dei-processi

Stallo attivo (livelock)

```
int main()
{
  pthread t thread one, thread two;

  pthread_mutex init(&first_mutex, NULL);
  pthread_mutex init(&second_mutex, NULL);

  if(pthread_create(&thread_one, NULL, do work one, NULL) < 0)
  {
      printf("errore creazione thread_one\n");
      exit(1);
  }

  if(pthread_create(&thread two, NULL, do work two, NULL) < 0)
  {
      printf("errore creazione thread two\n");
      exit(1);
  }

  pthread join (thread_one, NULL);
  pthread join (thread two, NULL);
  return 0;
}
```

Se thread_one acquisisce first_mutex e, in seguito, thread_two acquisisce second_mutex si può avere uno stallo attivo

https://github.com/dbloisi/3.4-stallo-dei-processi

Stallo attivo (livelock)

thread_one acquisisce first_mutex
thread_two acquisisce second_mutex
thread_one prova ad acquisire second_mutex INSUCCESSO
thread_two prova ad acquisire first_mutex INSUCCESSO
thread_one rilascia first_mutex
thread_two rilascia second_mutex

thread_one acquisisce first_mutex
thread_two acquisisce second_mutex
thread_one prova ad acquisire second_mutex INSUCCESSO
thread_two prova ad acquisire first_mutex INSUCCESSO
thread_one rilascia first_mutex
thread_two rilascia second_mutex

thread_one acquisisce first_mutex
thread_two acquisisce second_mutex
thread_one prova ad acquisire second_mutex INSUCCESSO
thread_two prova ad acquisire first_mutex INSUCCESSO
thread_one rilascia first_mutex
thread_two rilascia second_mutex

…

Lo stallo attivo si verifica quando un thread tenta continuamente una azione che non ha successo

https://github.com/dbloisi/3.4-stallo-dei-processi

Situazioni di stallo

Le condizioni necessarie per generare una situazione di stallo sono:

- Mutua esclusione
- Possesso e attesa
- Assenza di prelazione
- Attesa circolare

Condizioni necessarie allo stallo

Mutua esclusione

Deve esistere almeno una risorsa non condivisibile, cioè utilizzabile da un solo thread alla volta. Se un altro thread richiede tale risorsa, esso viene ritardato fino al rilascio della risorsa.

Condizioni necessarie allo stallo

Possesso e attesa

Un thread deve possedere almeno una risorsa ed essere in attesa di acquisire risorse che siano in possesso di altri thread.

Condizioni necessarie allo stallo

Assenza di prelazione

Le risorse non possono essere prelazionate. Questo significa che una risorsa può essere rilasciata solo volontariamente dal thread che la possiede, una volta terminato il proprio task.

Condizioni necessarie allo stallo

Attesa circolare

Deve esistere un insieme di thread $\{T_0, T_1, \ldots, T_n\}$ tale che T_0 sia in attesa di una risorsa posseduta da T_1, T_1 sia in attesa di una risorsa posseduta da T_2, …, T_{n-1} sia in attesa di una risorsa posseduta da T_n, T_n sia in attesa di una risorsa posseduta da T_0

Grafo di assegnazione delle risorse

Le situazioni di stallo si possono descrivere con maggior precisione avvalendosi di una rappresentazione detta **grafo di assegnazione delle risorse**

arco di assegnazione da risorsa a thread

si usa un rettangolo per le classi di risorse (**un puntino per ogni istanza**)

si usa un cerchio per i thread

arco di richiesta da thread a risorsa

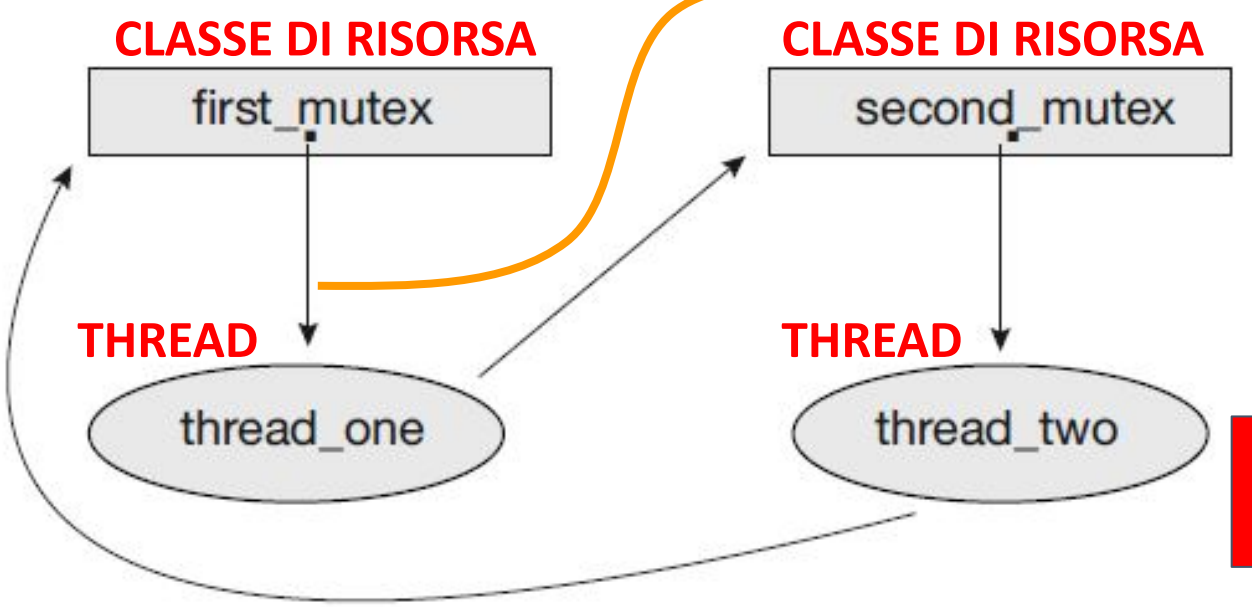


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.

Esempio Grafo di assegnazione delle risorse

```
/* thread_one esegue in questa funzione */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
      * Fa qualcosa
      */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two esegue in questa funzione */
void *do_work_two(void *param)
{
   pthread_mutex_lock (&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
      * Fa qualcosa
      */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

Figura 8.1 Esempio di stallo.

thread_one ha la CPU

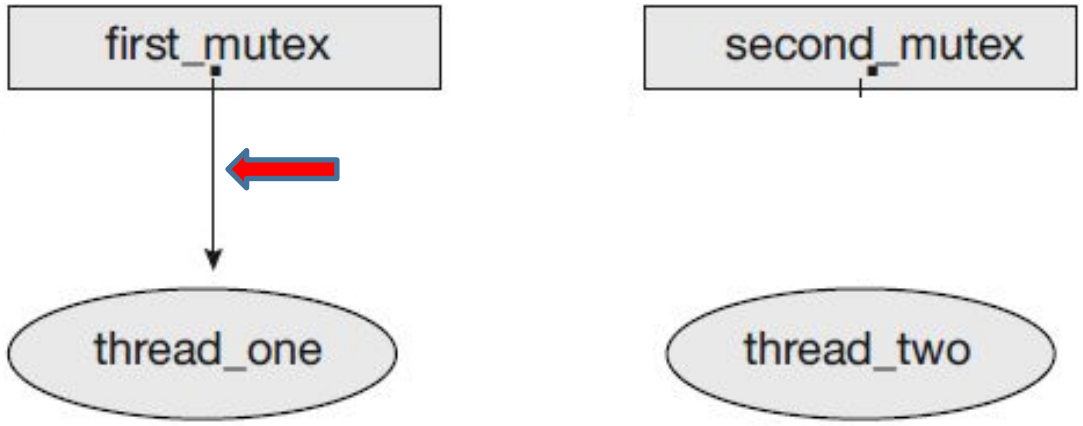


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.

Esempio Grafo di assegnazione delle risorse

```
/* thread_one esegue in questa funzione */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
        * Fa qualcosa
        */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two esegue in questa funzione */
void *do_work_two(void *param)
{
    pthread_mutex_lock (&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
        * Fa qualcosa
        */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

Figura 8.1 Esempio di stallo.

thread_two ha la CPU

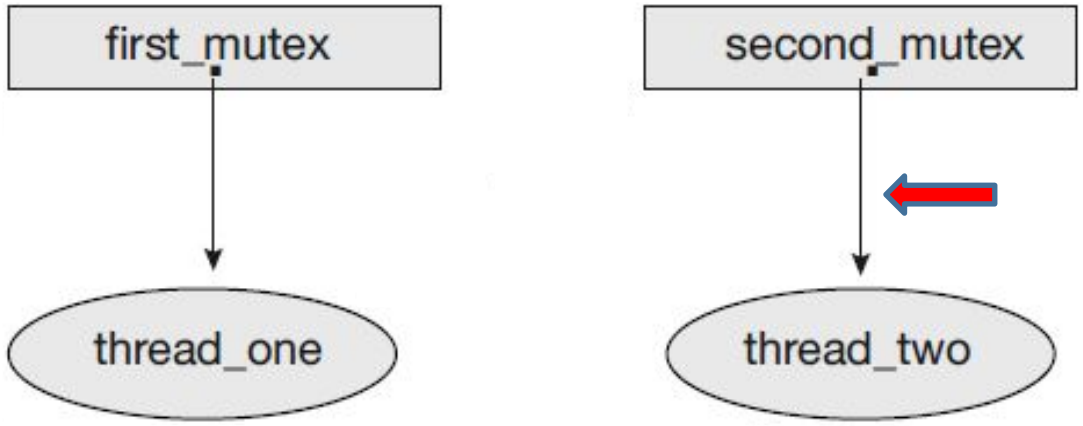


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.

Esempio Grafo di assegnazione delle risorse

```
/* thread_one esegue in questa funzione */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
      * Fa qualcosa
      */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two esegue in questa funzione */
void *do_work_two(void *param)
{
   pthread_mutex_lock (&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
      * Fa qualcosa
      */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

Figura 8.1 Esempio di stallo.

thread_one ha la CPU

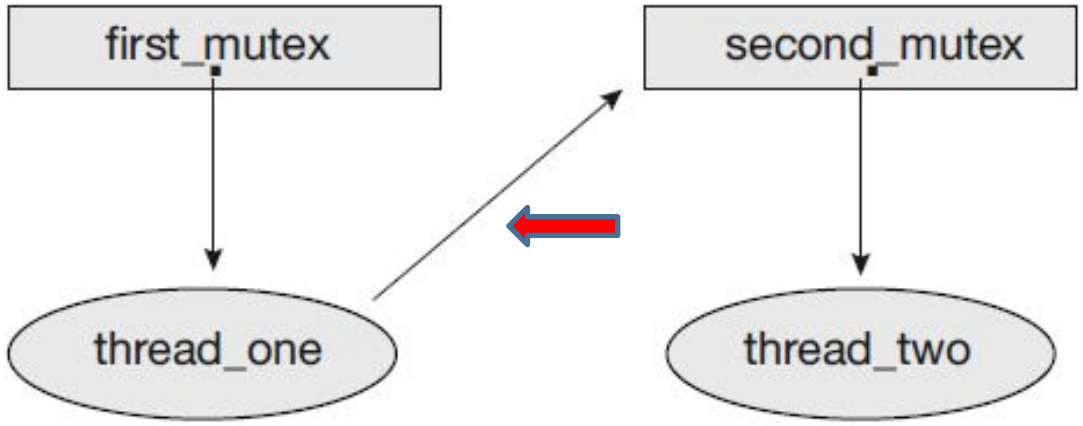


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.

Esempio Grafo di assegnazione delle risorse

```
/* thread_one esegue in questa funzione */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
        * Fa qualcosa
        */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two esegue in questa funzione */
void *do_work_two(void *param)
{
    pthread_mutex_lock (&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
        * Fa qualcosa
        */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

Figura 8.1 Esempio di stallo.

thread_two ha la CPU

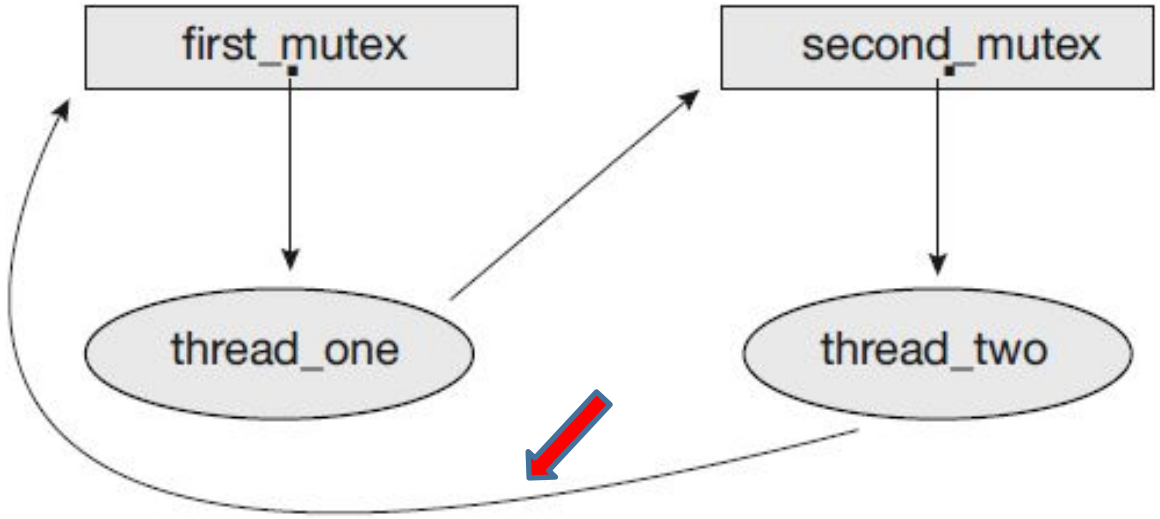


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.

Esempio Grafo di assegnazione delle risorse

```
/* thread_one esegue in questa funzione */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
        * Fa qualcosa
        */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two esegue in questa funzione */
void *do_work_two(void *param)
{
    pthread_mutex_lock (&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
        * Fa qualcosa
        */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

Figura 8.1 Esempio di stallo.

STALLO!

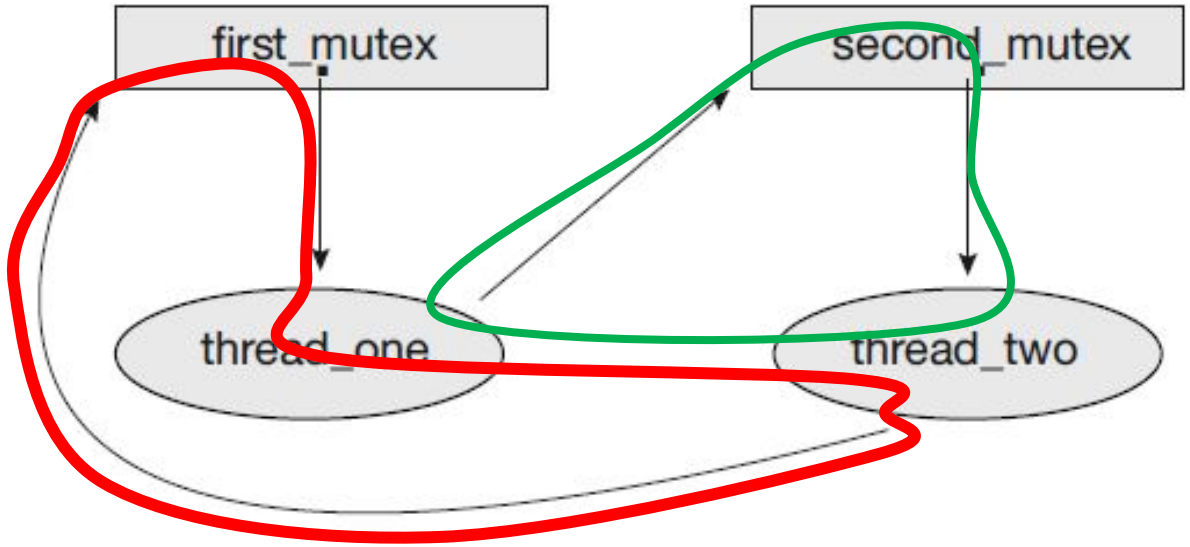


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.

Formalismo - Grafo di assegnazione delle risorse

- Insiemi T, R ed E:
  - $T = \{ T_1, T_2, T_3 \}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{ T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3 \}$
- Istanze delle risorse:
  - un'istanza del tipo di risorsa R_1
  - due istanze del tipo di risorsa R_2
  - un'istanza del tipo di risorsa R_3
  - tre istanze del tipo di risorsa R_4
- Stati dei thread:
  - il thread T_1 possiede un'istanza del tipo di risorsa R_2 e attende un'istanza del tipo di risorsa R_1
  - il thread T_2 possiede un'istanza dei tipi di risorsa R_1 ed R_2 e attende un'istanza del tipo di risorsa R_3
  - il thread T_3 possiede un'istanza del tipo di risorsa R_3

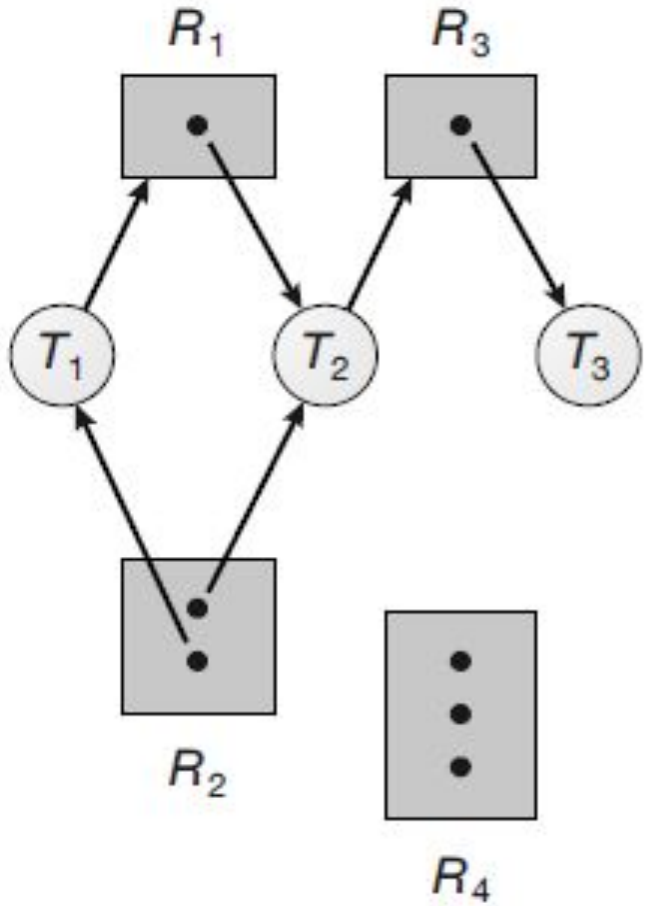
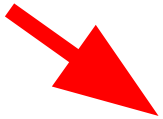


Figura 8.4 Grafo di assegnazione delle risorse.

Deadlock e grafo di assegnazione delle risorse

- **Se il grafo di assegnazione delle risorse non contiene cicli, allora nessun thread del sistema subisce uno stallo.**
- **Se il grafo contiene un ciclo, allora può sopraggiungere uno stallo.**



Dipende dal numero di istanze delle risorse contese

Grafo di assegnazione delle risorse con stallo

Se viene aggiunto un arco di richiesta $T_3 \rightarrow R_2$ al grafo della Figura 8.4 si viene a creare una situazione di stallo

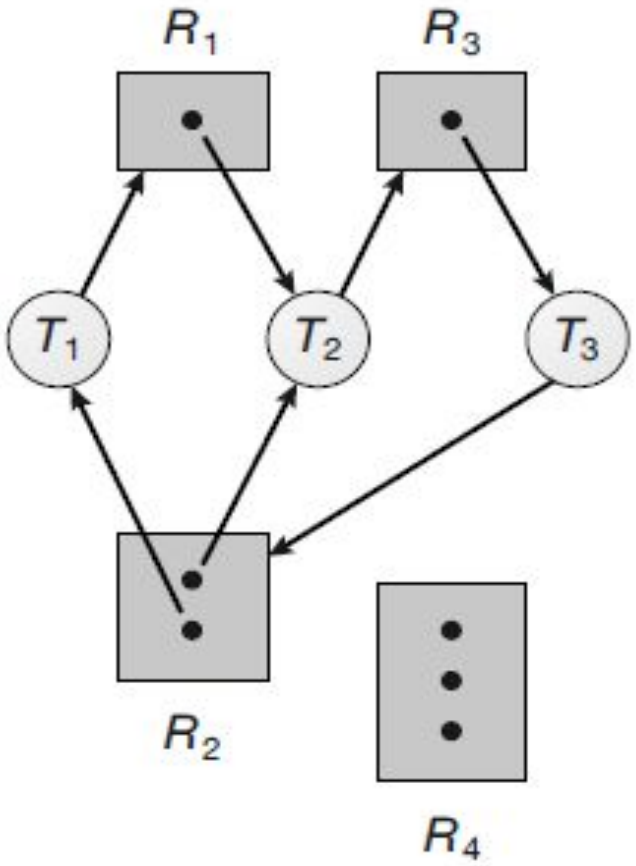


Figura 8.5 Grafo di assegnazione delle risorse con uno stallo.

Grafo di assegnazione delle risorse con ciclo senza stallo

Anche in questo esempio c'è un ciclo:

$$T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$$

In questo caso, però, non si ha alcuno stallo: il thread T_4 può rilasciare la propria istanza del tipo di risorsa R_2, che si può assegnare al thread T_3, rompendo il ciclo.

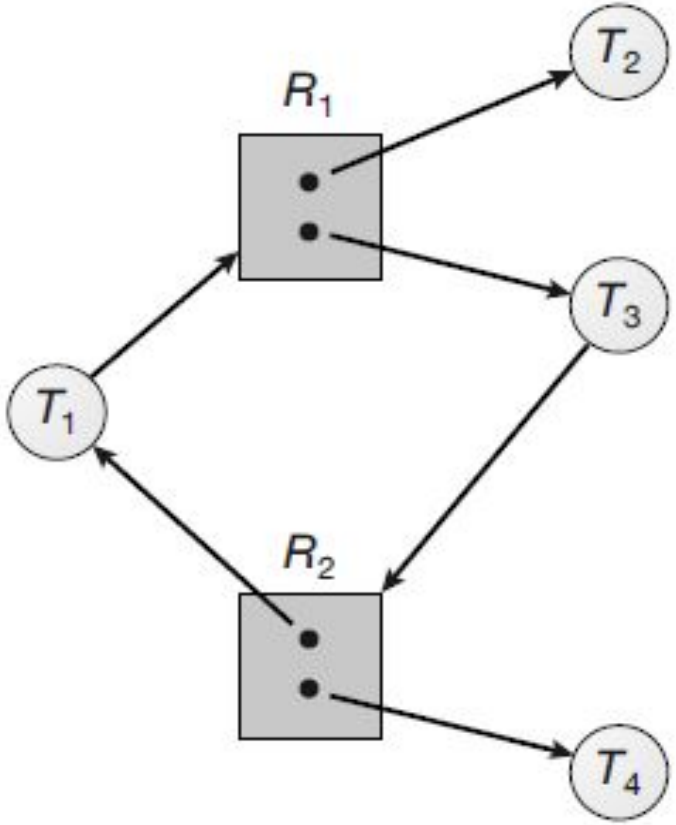


Figura 8.6 Grafo di assegnazione delle risorse con un ciclo, ma senza stallo.

Gestione delle situazioni di stallo

Il problema delle situazioni di stallo si può affrontare in tre modi:

1. ignorare del tutto il problema, *fingendo* che le situazioni di stallo non possano mai verificarsi nel sistema

soluzione adottata da Linux e Windows

2. usare un protocollo per prevenire o evitare le situazioni di stallo, assicurando che il sistema non entri *mai* in stallo

soluzione che necessita del contributo del programmatore

3. permettere al sistema di entrare in stallo, individuarlo e, quindi, eseguire il ripristino

soluzione adottata nei database

Prevenire le situazioni di stallo

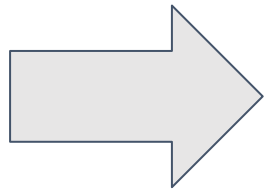
Prevenire le situazioni di stallo significa far uso di metodi atti ad assicurare che non si verifichi almeno una delle condizioni necessarie

- Mutua esclusione
- Possesso e attesa
- Assenza di prelazione
- Attesa circolare

Prevenire le situazioni di stallo

Affinché si abbia uno stallo si devono verificare quattro condizioni necessarie; perciò si può *prevenire il verificarsi di uno stallo* assicurando che almeno una di queste condizioni non possa capitare.

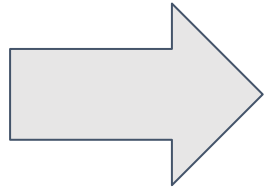
Mutua esclusione → Tutte le risorse devono essere condivisibili.
Impossibile in pratica, per esempio un lock mutex non può essere condiviso



Prevenire le situazioni di stallo

Possesso e attesa → Occorre garantire che un thread che richiede una risorsa non ne possegga altre. Per esempio, possiamo assegnare tutte le risorse necessarie a un thread prima che vada in esecuzione.

Poco efficiente e può provocare attesa indefinita



Prevenire le situazioni di stallo

Assenza di prelazione → Se un thread T possiede una o più risorse e ne richiede un'altra, che però è impegnata, allora si esercita la prelazione su tutte le risorse in possesso di T (rilascio implicito).
Difficile da applicare a lock mutex e semafori

Attesa circolare → impo
i tipi di risorse e imporre che ciascun thread richieda le risorse in ordine crescente.
Soluzione pratica

Ordinamento

Imporre un ordinamento sui lock non garantisce l'assenza di situazioni di stallo quando i lock possono essere acquisiti dinamicamente

Cosa succede se si invocano contemporaneamente

`transaction(account1, account2, 25.)`
e
`transaction(account2, account1, 50.)`
?

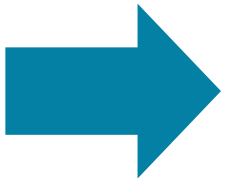
```
void transaction(Account from, Account to, double amount)
{
  mutex lock1, lock2;
  lock1 = get_lock(from);
  lock2 = get_lock(to);

  acquire(lock1);
    acquire(lock2);

      withdraw(from, amount);
      deposit(to, amount);

    release(lock2);
  release(lock1);
}
```

Ogni conto ha un lock_mutex associato

Figura 8.7 Esempio di stallo con ordinamento dei lock.

Evitare le situazioni di stallo

Il metodo per prevenire le situazioni di stallo illustrato in precedenza può causare effetti collaterali negativi



In alternativa:
per **evitare situazioni di stallo** occorre che il sistema operativo abbia in anticipo informazioni aggiuntive riguardanti le risorse che un thread richiederà e userà durante le sue attività.

Evitare le situazioni di stallo

L’algoritmo per **evitare lo stallo** deve esaminare dinamicamente lo stato di assegnazione delle risorse per garantire che non possa esistere una condizione di attesa circolare

- Algoritmo con grafo di assegnazione delle risorse
- Algoritmo del banchiere
- Algoritmo di verifica della sicurezza
- Algoritmo di richiesta delle risorse

Stato sicuro

Uno **stato** si dice **sicuro** se il sistema è in grado di assegnare risorse a ciascun thread (fino al suo massimo) in un certo ordine e impedire il verificarsi di uno stallo.

Stato sicuro

Un sistema si trova in stato sicuro solo se esiste una **sequenza sicura**

Uno **stato sicuro** non è di stallo. Viceversa, uno stato di stallo è uno stato non sicuro; tuttavia *non* tutti gli stati non sicuri sono stati di stallo

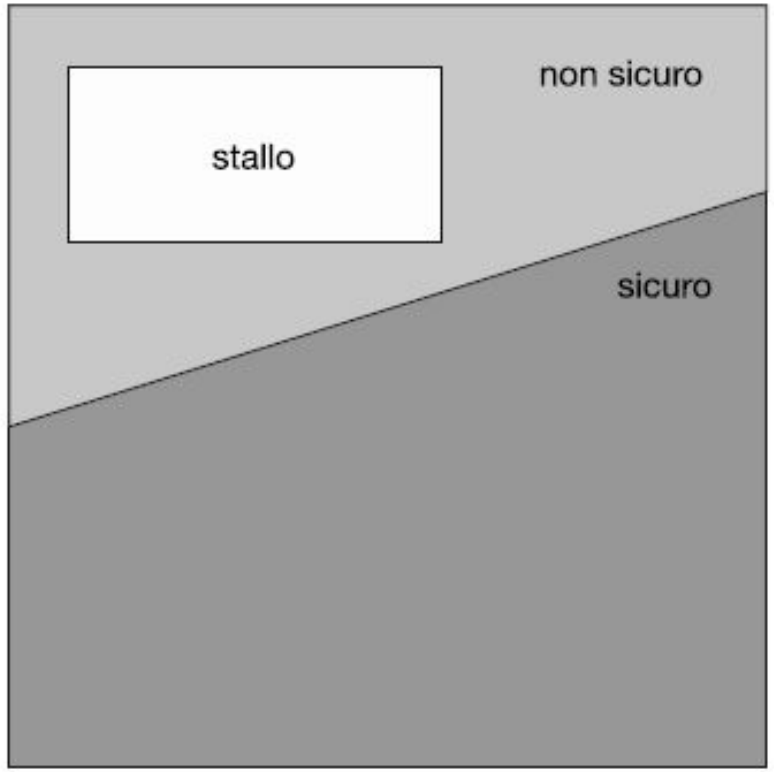


Figura 8.8 Spazi degli stati sicuri, non sicuri e di stallo.

Esempio: Stato sicuro

Risorse totali a disposizione: 12

Thread in esecuzione: 3

Situazione al tempo t_0:

	este massime	Unità possedute
		5
		2
		2

La sequenza <T1,T0,T2> è sicura?

Esempio: Stato sicuro

Risorse totali a disposizione: 12

Thread in esecuzione: 3

Situazione al tempo t_0:

	este massime	Unità possedute
		5
		2
		2

La sequenza <T1,T0,T2> è sicura?

Si. Abbiamo 3 risorse libere (12 – (5+2+2)). A T1 possiamo assegnare subito 2 risorse, che saranno poi restituite insieme alle altre 2 al termine di T1. Abbiamo quindi 5 risorse libere. A T0 possiamo assegnare 5 risorse, che saranno poi restituite insieme alle altre 5 al termine di T0. Abbiamo quindi 10 risorse libere. A T2 possiamo assegnare 7 risorse, che saranno poi restituite insieme alle altre 2 al termine di T2. Abbiamo quindi 12 risorse libere

Esempio: Stato non sicuro

Risorse totali a disposizione: 12

Thread in esecuzione: 3

Situazione al tempo t_1:

	este massime	Unità possedute
		5
		2
		3

La sequenza <T1,T0,T2> è sicura?

Esempio: Stato non sicuro

Risorse totali a disposizione: 12

Thread in esecuzione: 3

Situazione al tempo t_1:

	este massime	Unità possedute
		5
		2
		3

La sequenza <T1,T0,T2> è sicura?

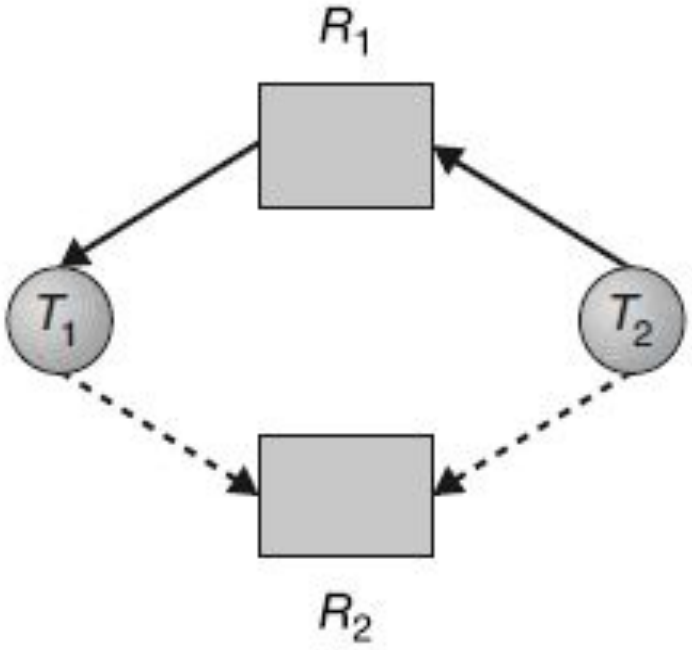
No. Abbiamo 2 risorse libere (12 – (5+2+3)). A T1 possiamo assegnare subito 2 risorse, che saranno poi restituite insieme alle altre 2 al termine di T1. Abbiamo quindi 4 risorse libere. A T0 non possiamo assegnare le 5 risorse necessarie, quindi T0 deve attendere. A T2 non possiamo assegnare 6 risorse necessarie, quindi T2 deve attendere. Siamo in stallo.

Algoritmo con grafo di assegnazione delle risorse

Si può applicare se ogni classe di risorse ha una sola istanza

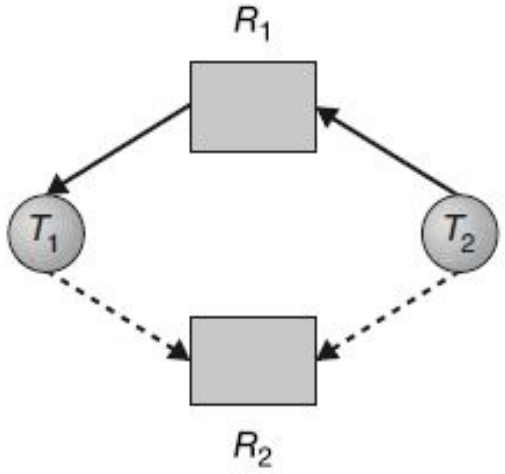
- Un arco di rivendicazione
  $T_i \longrightarrow R_j$ indica che
  un thread T_i *può richiedere*
  la risorsa R_j

- Viene indicato con una
  freccia tratteggiata

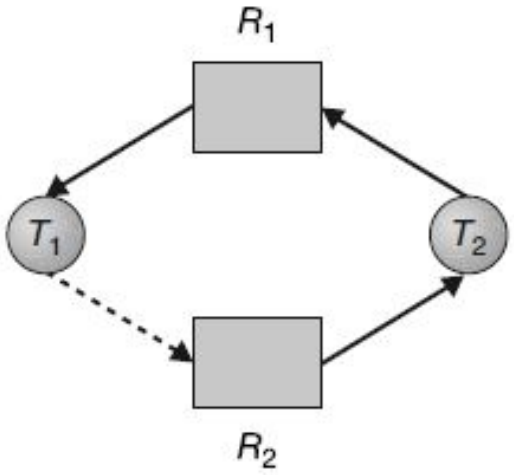


Algoritmo con grafo di assegnazione delle risorse

Si può applicare se ogni classe di risorse ha una sola istanza



$_1$ che T_2 *richiederanno in futuro* R_2



pponga che T_2 *richieda* R_2

- Sebbene sia attualmente libera, R_2 *non* può essere assegnata a T_2, poiché questa operazione creerebbe un ciclo nel grafo e un ciclo indica che il sistema è in uno stato non sicuro
- Se, a questo punto, T_1 *richiedesse* R_2, si avrebbe uno stallo

Algoritmo del banchiere

Algoritmo del banchiere → Questo nome è stato scelto perché l'algoritmo si potrebbe impiegare in un sistema bancario per assicurare che la banca non assegni mai tutto il denaro disponibile, in modo da non poter più soddisfare le richieste di tutti i suoi clienti

Algoritmo del banchiere

La realizzazione dell'algoritmo del banchiere richiede la gestione di alcune strutture dati che codificano lo stato di assegnazione delle risorse del sistema.

Disponibili	*Massimo*	*Assegnate*	*Necessità*

Algoritmo del banchiere

Si supponga di avere ***n*** thread e ***m*** classi di risorsa

Disponibili

- Array mono-dimensionale di lunghezza ***m*** che indica il numero di risorse disponbili per ogni classe ***j***
- ***Available[j] = k*** indica che ci sono ***k*** istanze libere della risorsa R_j

Massimo

- Matrice di dimensione ***n***x***m*** che definisce la massima richiesta di risorse per ogni thread nel sistema
- ***Max[i, j] = k*** indica che un thread T_i può richiedere al massimo ***k*** istanze della risorsa R_j

Algoritmo del banchiere

Assegnate

- Matrice di dimensione ***n*x*m*** che definisce il numero di risorse di ogni classe allocate al momento a ogni thread.
- ***Allocation[i, j] = k*** indica che al thread T_i sono allocate al momento ***k*** istanze della risorsa R_j

Necessità

- Matrice di dimensione ***n*x*m*** che tiene traccia delle risorse residue di cui ha bisogno ogni thread.
- ***Need [i, j] = k*** indica che il thread T_i ha bisogno al momento di ***k*** istanze della risorsa R_j per la sua esecuzione.
- ***Need [i, j] = Max [i, j] – Allocation [i, j]***

Rilevamento delle situazioni di stallo

Istanza singola di ciascun tipo di risorsa

grafo di attesa

variante del grafo di assegnazione delle risorse

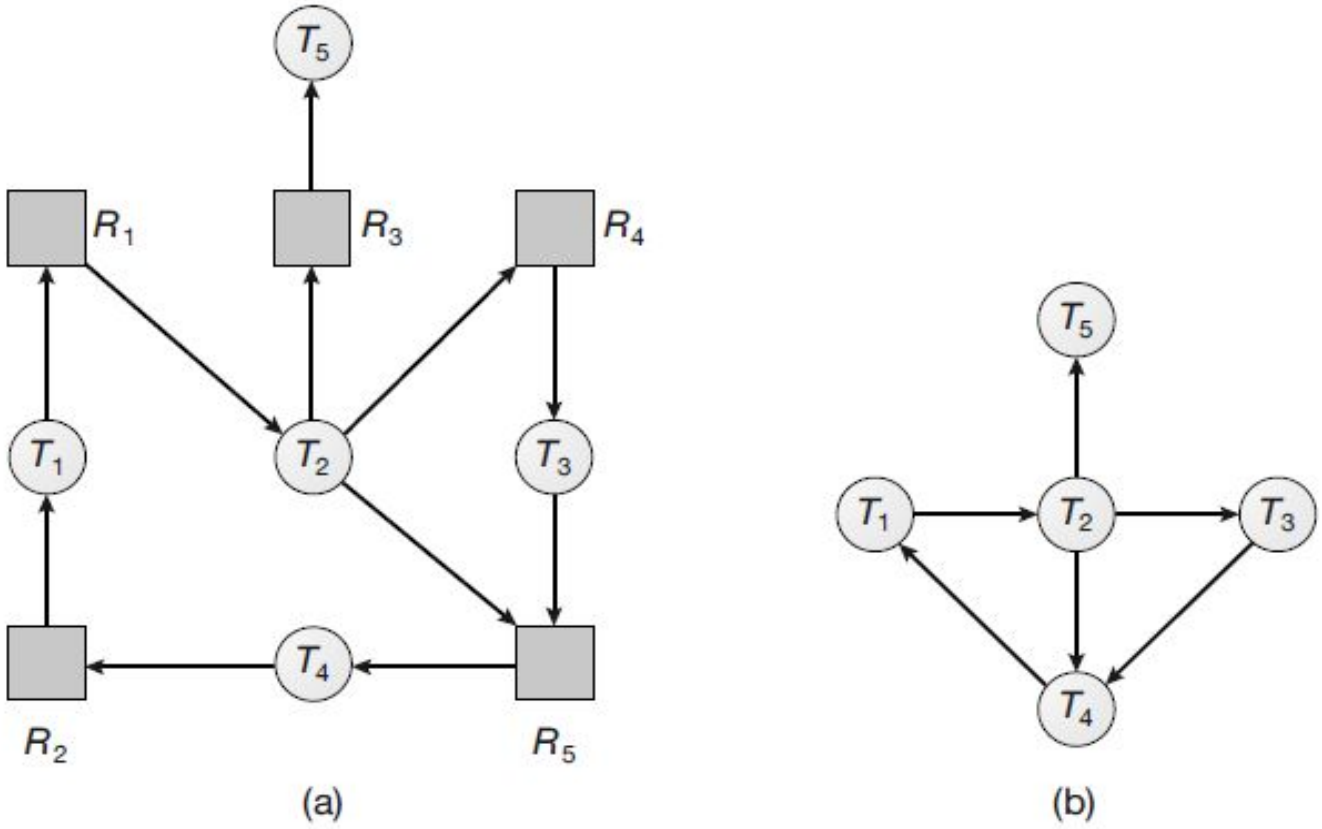


Figura 8.11 (a) Grafo di assegnazione delle risorse; (b) Grafo d'attesa corrispondente.

Rilevamento delle situazioni di stallo

Più istanze di ciascun tipo di risorsa

Lo schema con grafo di attesa non si può applicare ai sistemi di assegnazione delle risorse con più istanze di ciascun tipo di risorsa.

Rilevamento delle situazioni di stallo

Più istanze di ciascun tipo di risorsa

Esiste un algoritmo di rilevamento di situazioni di stallo che, invece, è applicabile a tali sistemi.
Esso si serve di strutture dati variabili nel tempo, simili a quelle adoperate nell'algoritmo del banchiere

Disponibili *Assegnate* *Richieste*

Rilevamento delle situazioni di stallo

Più istanze di ciascun tipo di risorsa

Uso dell'algoritmo di rilevamento

si ricorre all'algoritmo di rilevamento in base a

1. frequenza presunta con la quale si verifica uno stallo;
2. numero dei thread che sarebbero influenzati da tale stallo.

Ripristino da situazioni di stallo

Terminazione di processi e thread

Per eliminare le situazioni di stallo attraverso la terminazione di processi o thread si possono adoperare due metodi:

- Terminazione di tutti i processi in stallo
- Terminazione di un processo alla volta fino all'eliminazione del ciclo di stallo

Ripristino da situazioni di stallo

Prelazione delle risorse

le risorse si sottraggono in successione ad alcuni processi e si assegnano ad altri finché si ottiene l’interruzione del ciclo di stallo.

Prelazione delle risorse

Ricorrendo alla prelazione delle risorse per l'eliminazione di uno stallo si devono considerare i seguenti problemi:

- Selezione di una "vittima"
- Ristabilimento di un precedente stato sicuro
- Attesa indefinita (starvation)