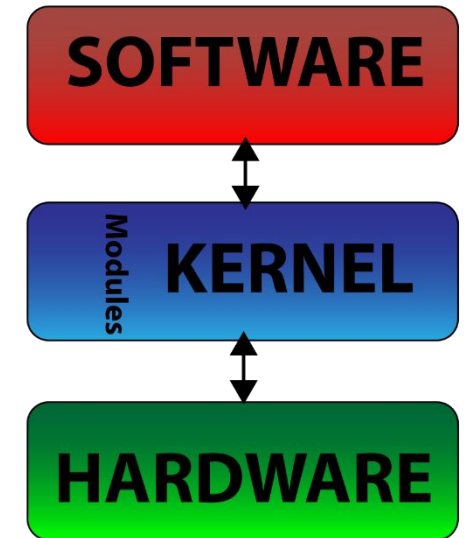
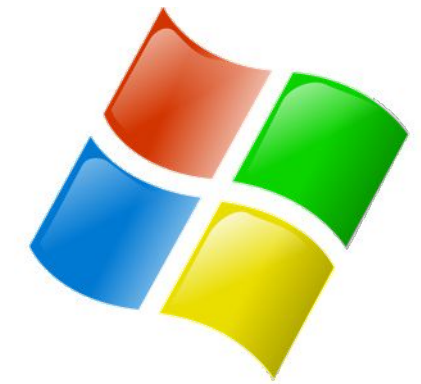




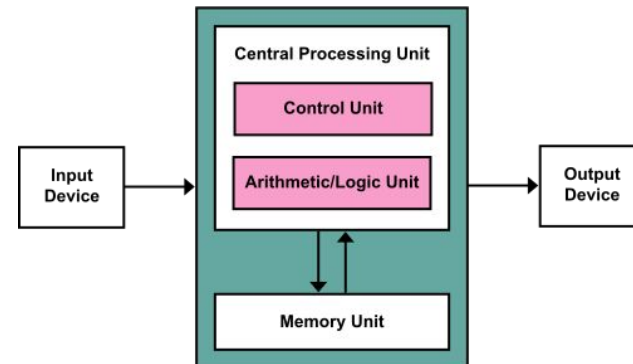
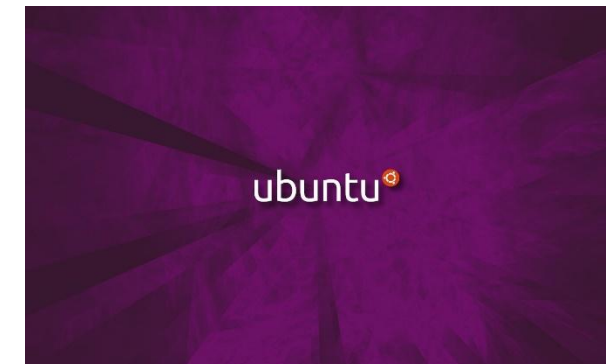
**UNIVERSITÀ DEGLI STUDI
DELLA BASILICATA**

Corso di Sistemi Operativi

Stallo dei processi

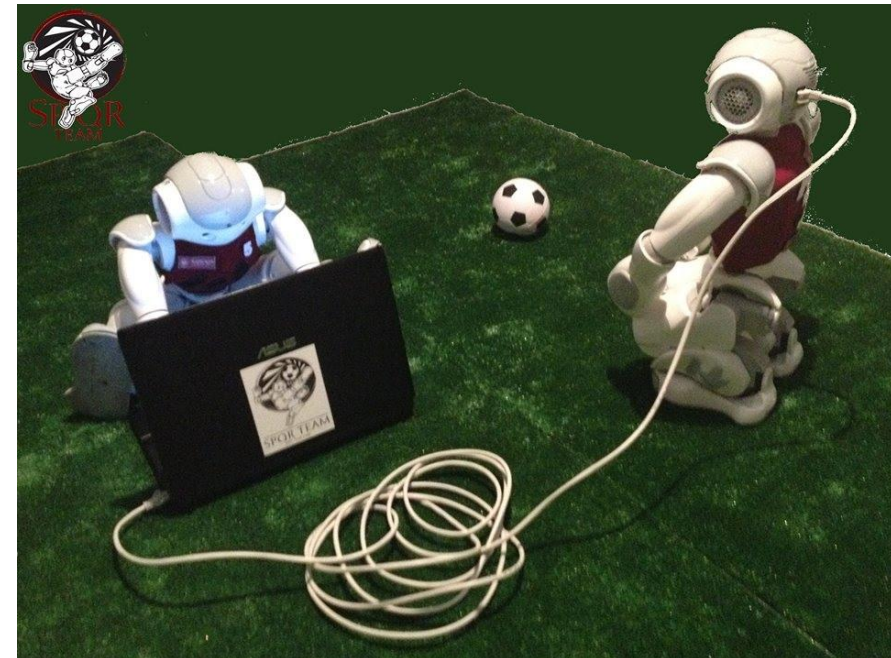
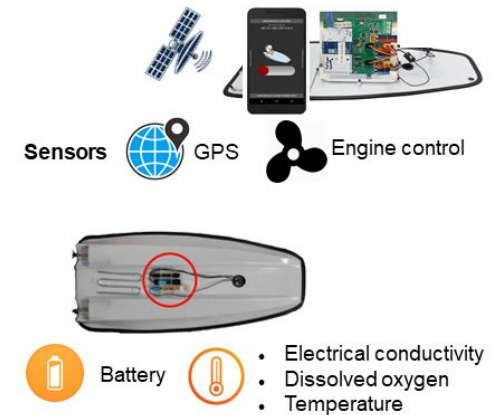


Docente:
**Domenico Daniele
Bloisi**



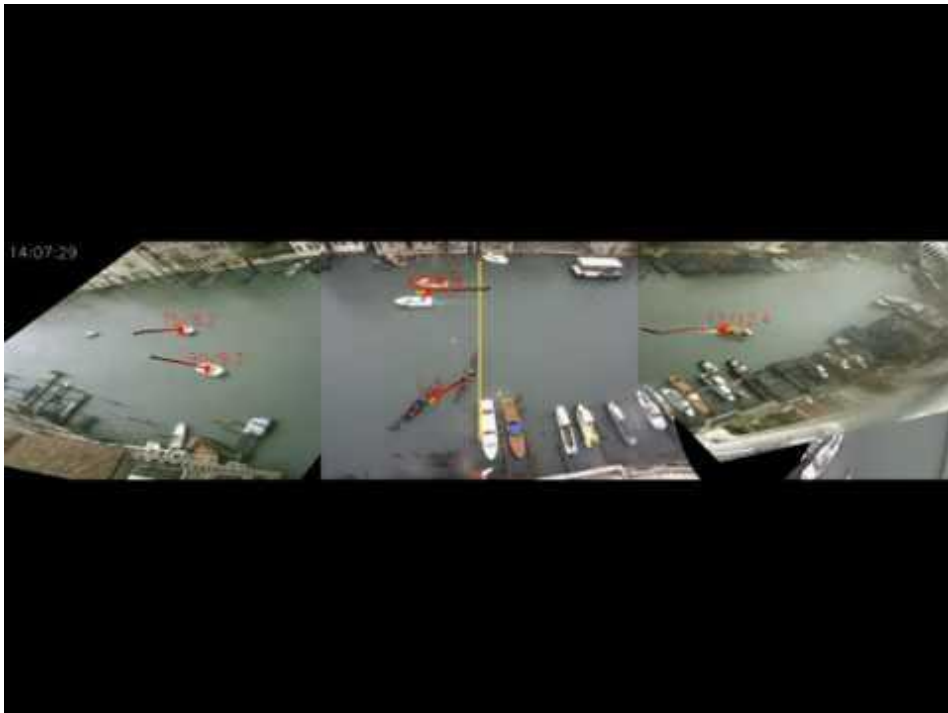
Domenico Daniele Bloisi

- Professore Associato
Dipartimento di Matematica, Informatica
ed Economia
Università degli studi della Basilicata
<http://web.unibas.it/bloisi>
- SPQR Robot Soccer Team
Dipartimento di Informatica, Automatica
e Gestionale Università degli studi di
Roma “La Sapienza”
<http://spqr.diag.uniroma1.it>



Interessi di ricerca

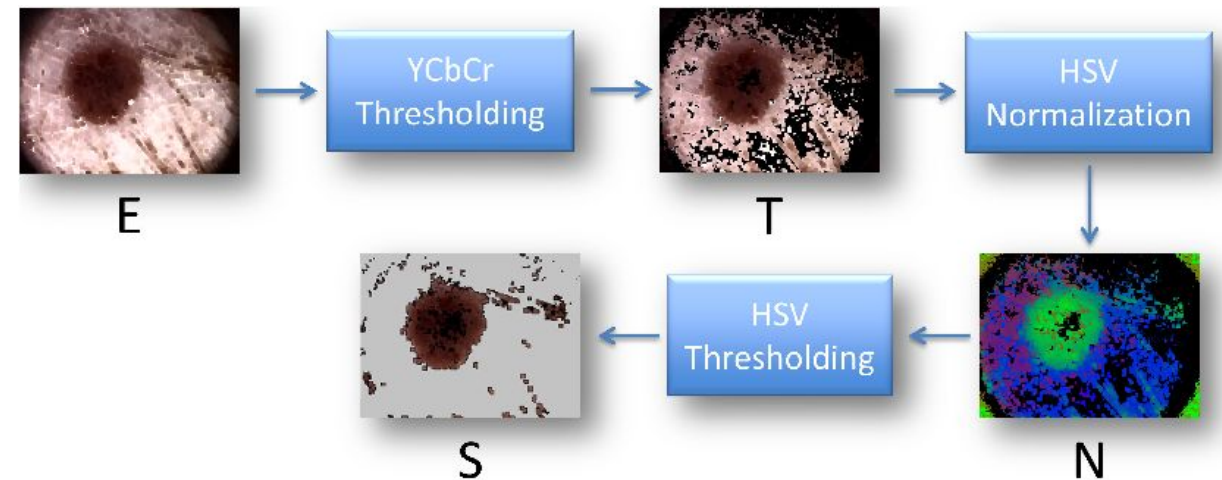
- Intelligent surveillance
- Robot vision
- Medical image analysis



https://youtu.be/9a70Ucgbi_U



<https://youtu.be/2KHNZX7UIWQ>



UNIBAS Wolves <https://sites.google.com/unibas.it/wolves>



- UNIBAS WOLVES is the robot soccer team of the University of Basilicata. Established in 2019, it is focussed on developing software for NAO soccer robots participating in RoboCup competitions.

- UNIBAS WOLVES team is twinned with SPQR Team at Sapienza University of Rome



<https://youtu.be/ji0OmkaWh20>

Informazioni sul corso

- Home page del corso:
<http://web.unibas.it/bloisi/corsi/sistemi-operativi.html>
- Docente: Domenico Daniele Bloisi
- Periodo: I semestre ottobre 2022 – gennaio 2023
 - Lunedì dalle 15:00 alle 17:00 (Aula Leonardo)
 - Martedì dalle 08:30 alle 10:30 (Aula 1)

Ricevimento

- In presenza, durante il periodo delle lezioni:
Lunedì dalle 17:00 alle 18:00
presso Edificio 3D, Il piano, stanza 15
Si invitano gli studenti a controllare regolarmente la bacheca degli avvisi per eventuali variazioni
- Tramite google Meet e al di fuori del periodo delle lezioni:
da concordare con il docente tramite email

Per prenotare un appuntamento inviare
una email a
domenico.bloisi@unibas.it



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

Risorse

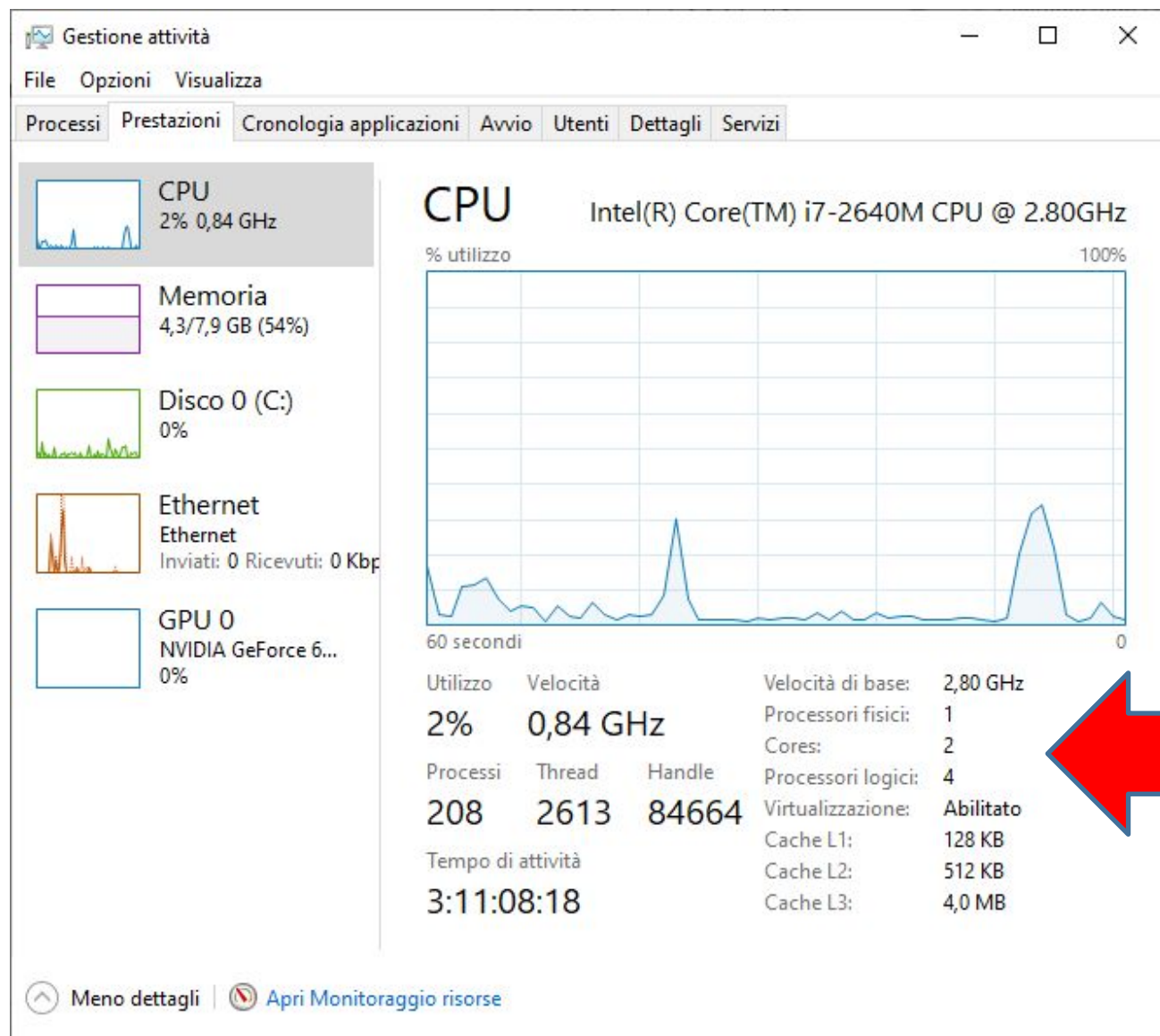
Un **sistema** è composto da un numero finito di risorse.

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 ch  non sar  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 */  
    . . .  
}
```

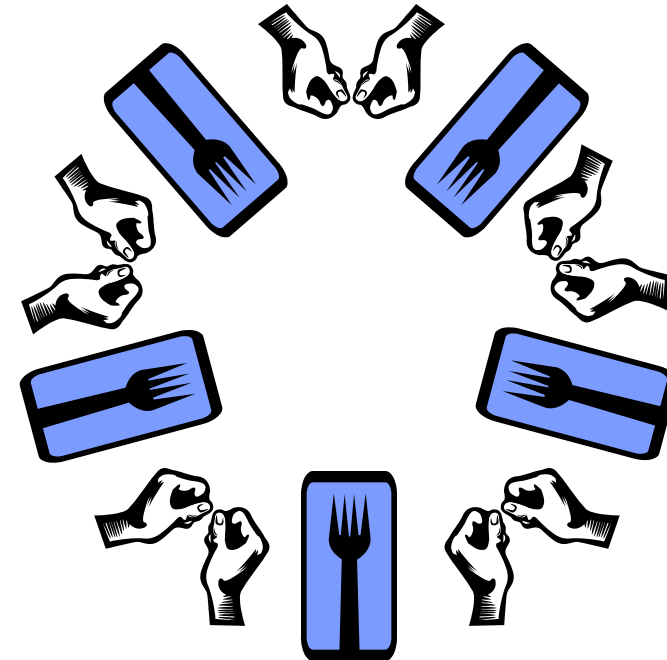


Figura 7.6 Struttura del filosofo *i*.

Esecuzione



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

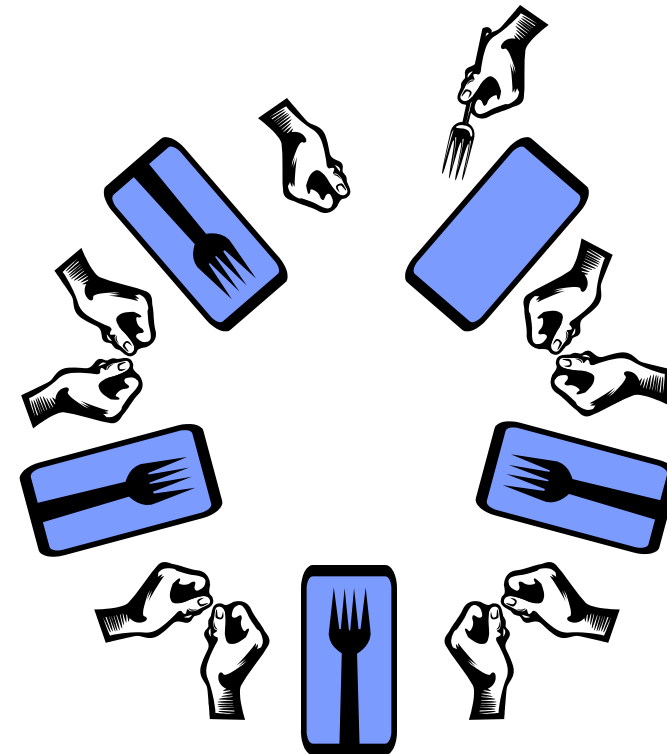


Figura 7.6 Struttura del filosofo *i*.

Esecuzione

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

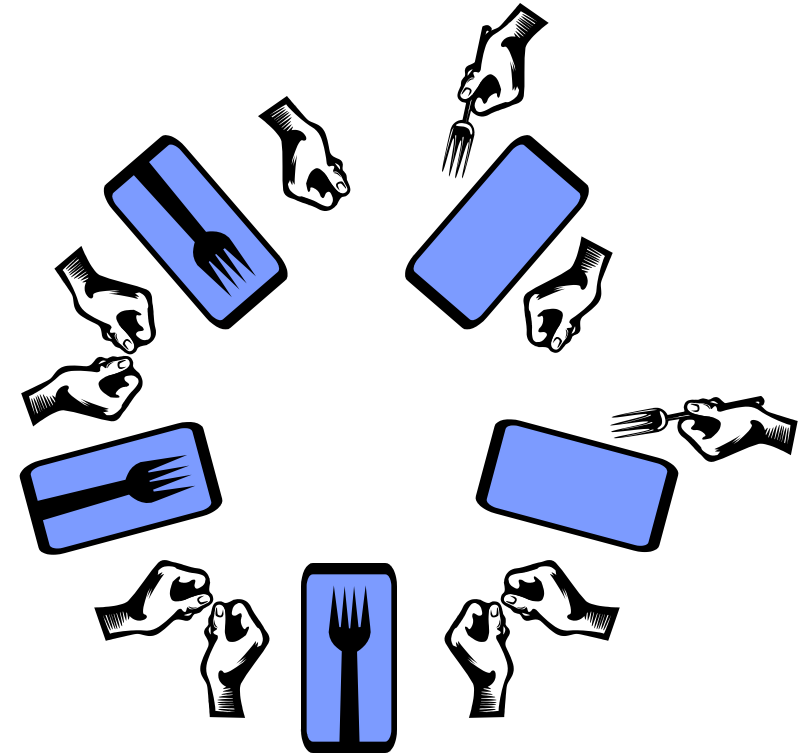


Figura 7.6 Struttura del filosofo i .

Esecuzione

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

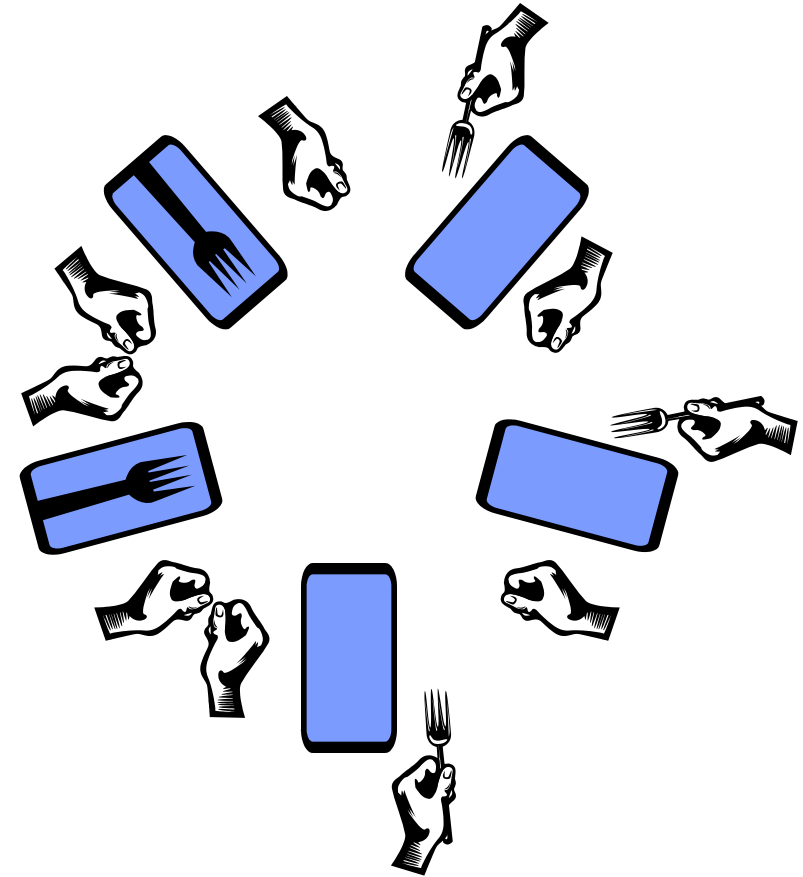


Figura 7.6 Struttura del filosofo i .

Esecuzione

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

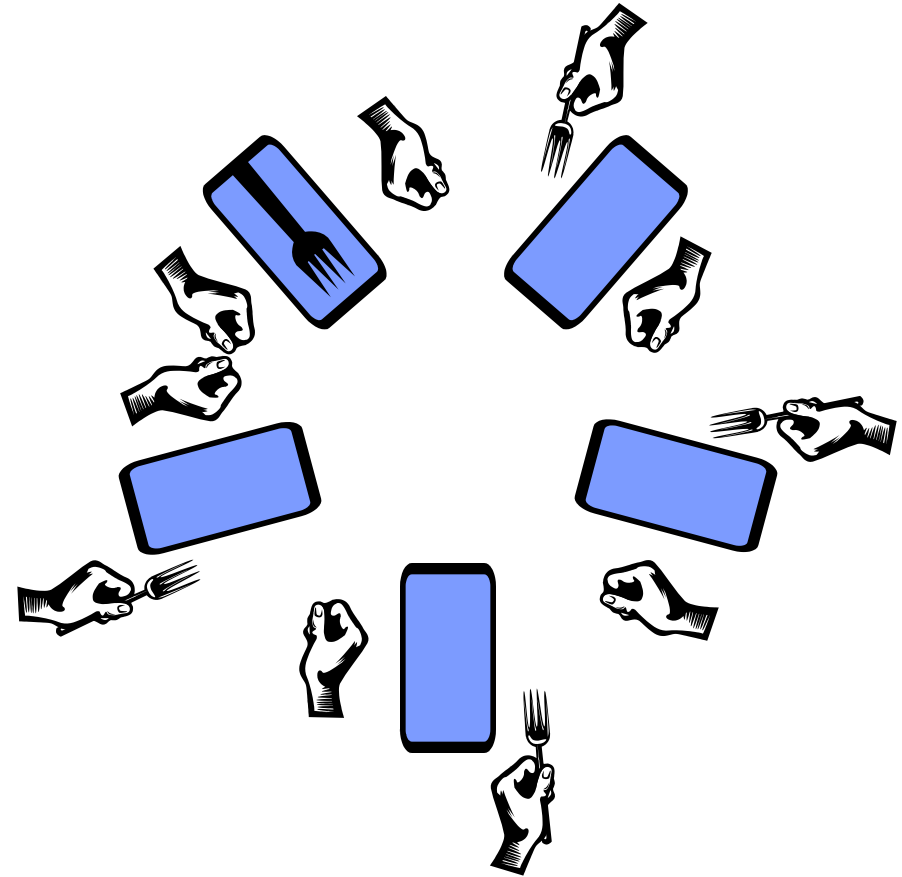


Figura 7.6 Struttura del filosofo i .

Esecuzione

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

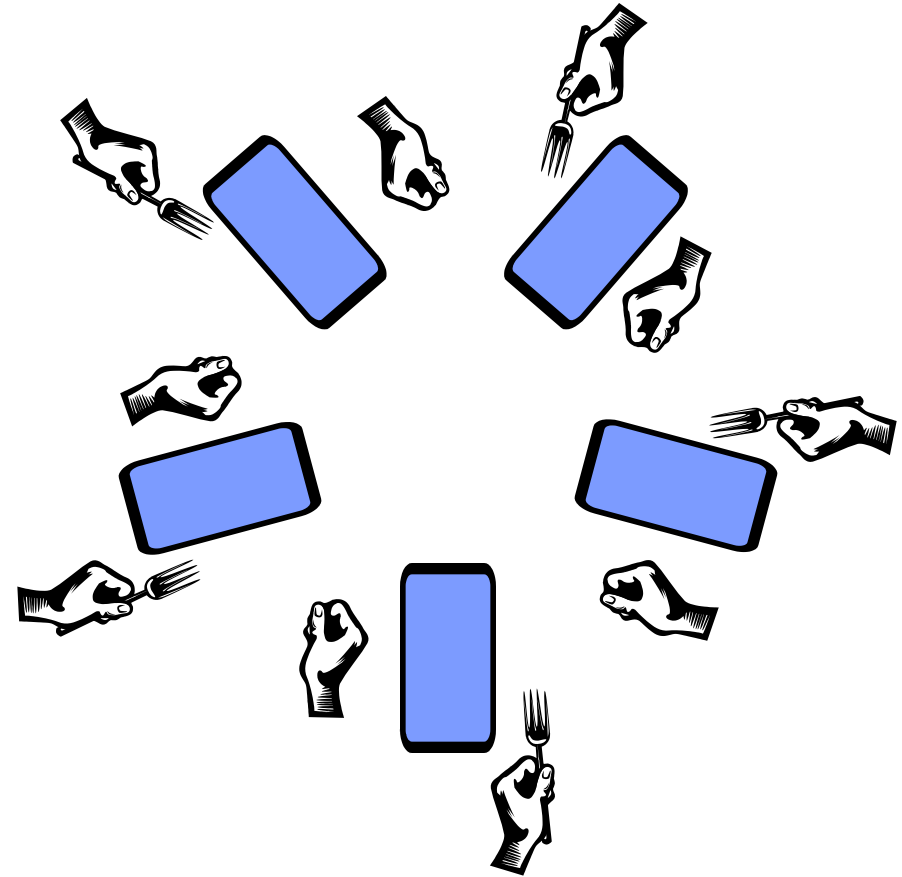
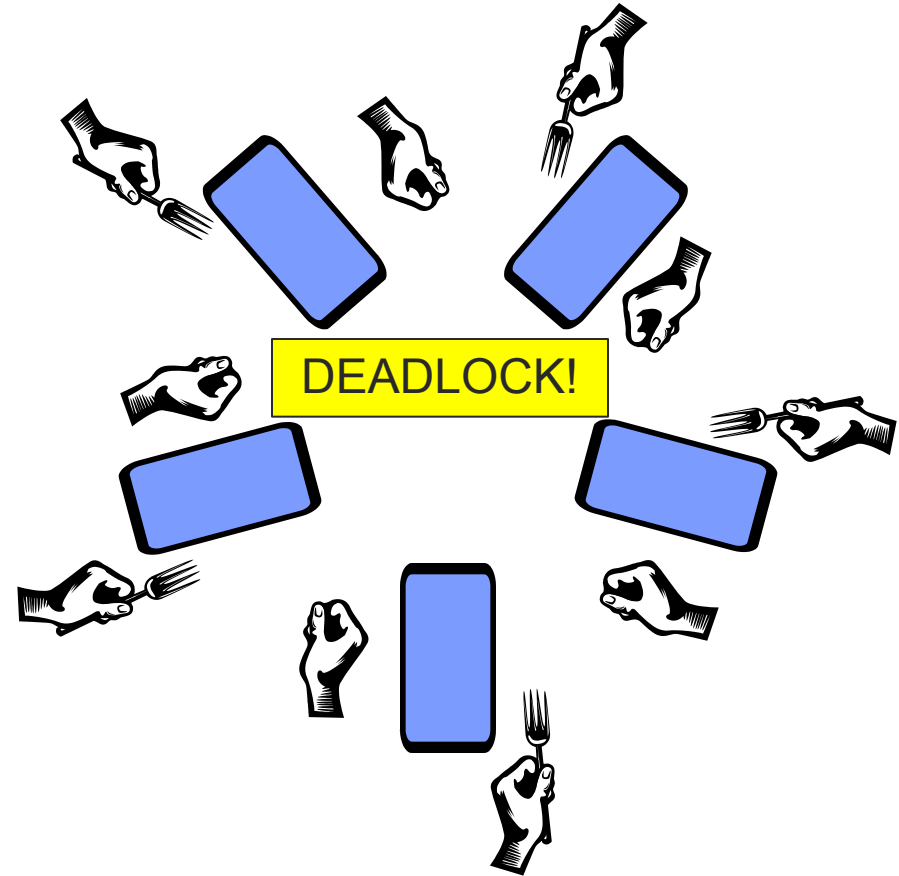


Figura 7.6 Struttura del filosofo *i*.

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 .



Esempio di stallo in applicazioni multithread

Inizializzazione:

```
pthread_mutex_t first_mutex;  
pthread_mutex_t second_mutex;  
  
pthread_mutex_init(&first_mutex, NULL);  
pthread_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.

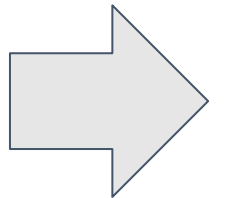
Codice completo

```
#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);
}
```



Codice completo

```
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;
}
```

Esecuzione

```
bloisi@bloisi-U36SG: ~/workspace/3.4-stallo-dei-processi
bloisi@bloisi-U36SG:~$ cd workspace/
bloisi@bloisi-U36SG:~/workspace$ git clone https://github.com/dbloisi/3.4-stallo-dei-processi.git
Cloning into '3.4-stallo-dei-processi'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.
Checking connectivity... done.
bloisi@bloisi-U36SG:~/workspace$ cd 3.4-stallo-dei-processi/
bloisi@bloisi-U36SG:~/workspace/3.4-stallo-dei-processi$ gcc mutex-deadlock.c -o
  deadlock -lpthread
bloisi@bloisi-U36SG:~/workspace/3.4-stallo-dei-processi$ ./deadlock
doing work one
doing work two
bloisi@bloisi-U36SG:~/workspace/3.4-stallo-dei-processi$ ./deadlock
█
```



Debug

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


Modifichiamo l'esempio di 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);
}
```

Esecuzione Esempio Modificato

```
bloisi@bloisi-U36SG: ~/3.4-stallo-dei-processi
bloisi@bloisi-U36SG:~/3.4-stallo-dei-processi$ gcc mutex-deadlock-mod.c -o deadlock -pthread
bloisi@bloisi-U36SG:~/3.4-stallo-dei-processi$ ./deadlock
work_one: first_mutex acquired. Going to sleep...
work_two: second_mutex acquired. Going to sleep...
█
```



DEADLOCK

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);
}
```

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

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

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 prelezionate. 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 $\{T_0, T_1, \dots, T_n\}$ 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, \dots, 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**

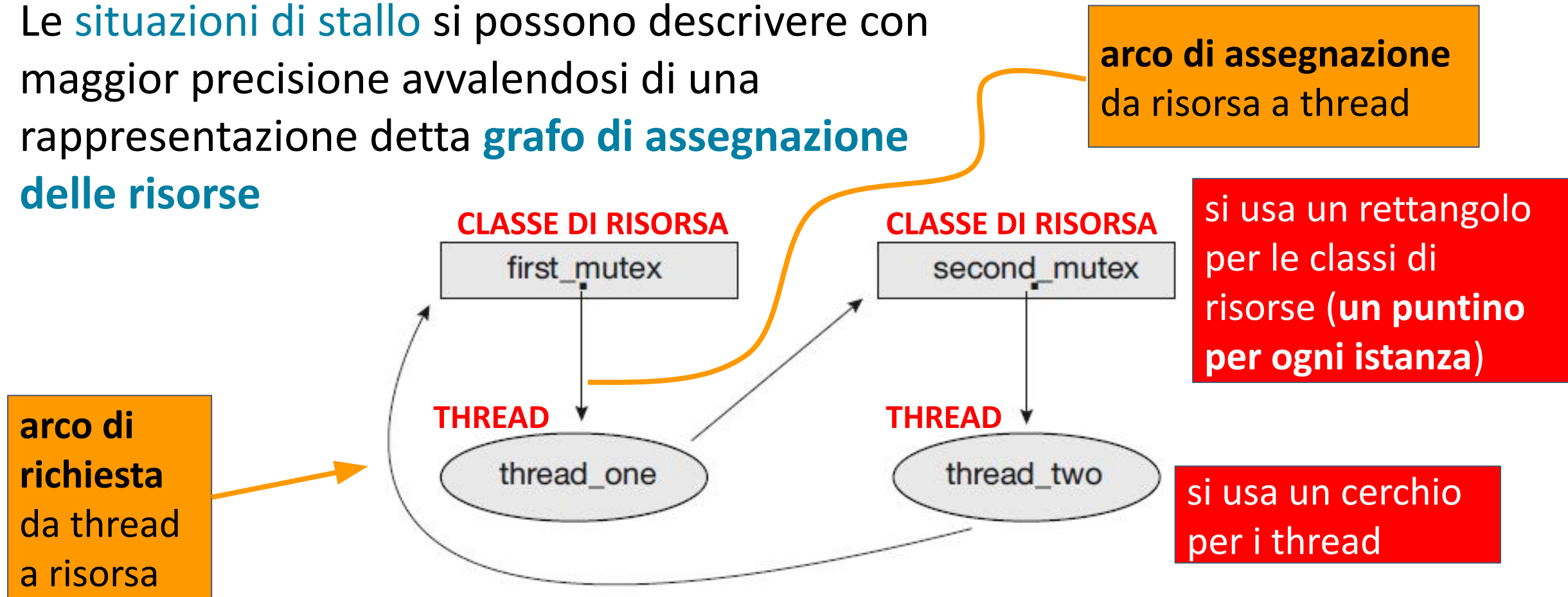


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.

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

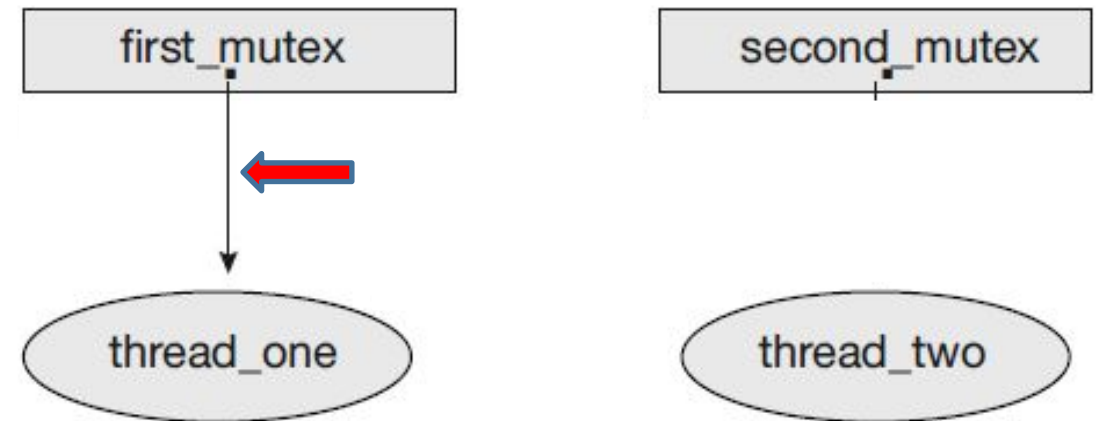


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.

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

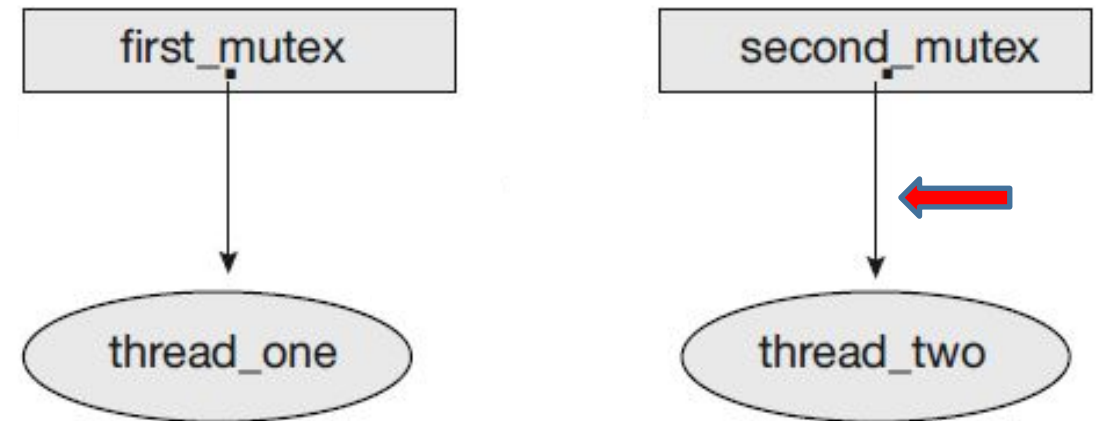


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.

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

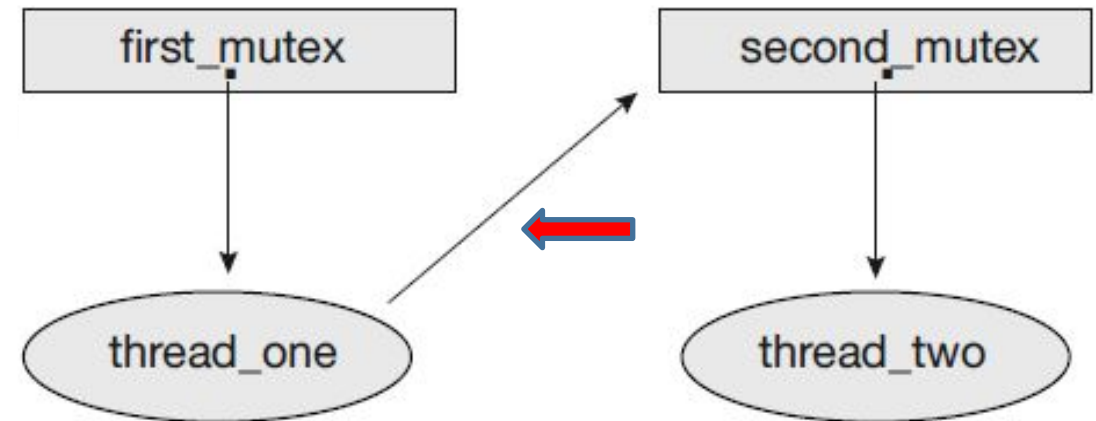


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.

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

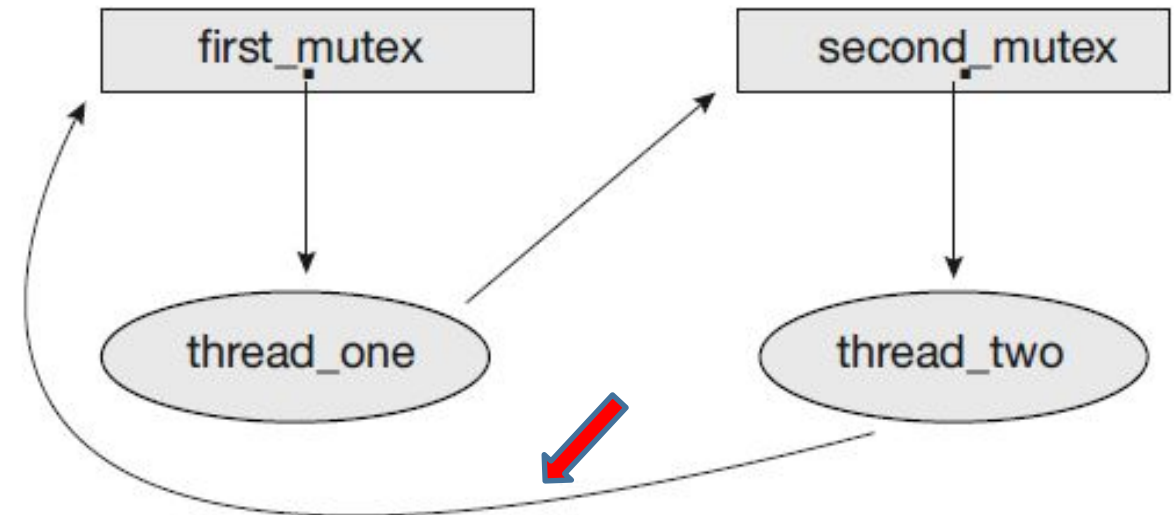


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.

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!

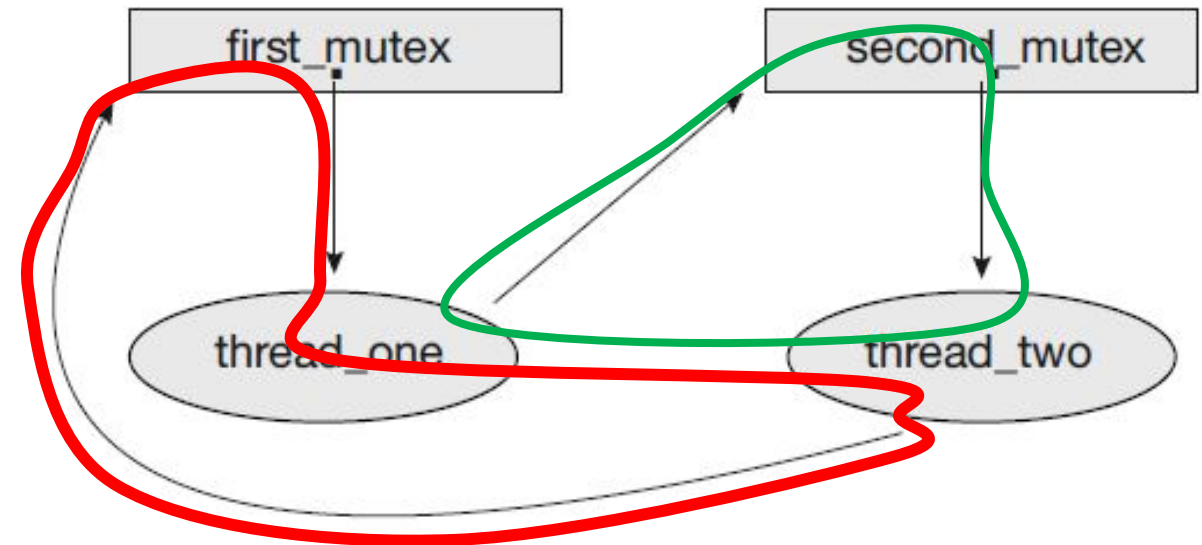


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.

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

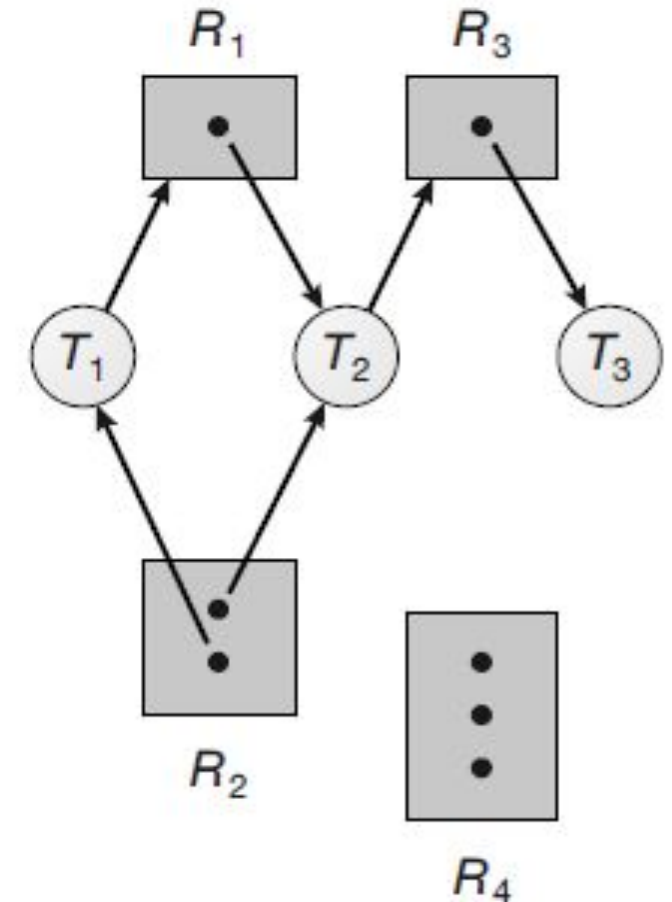
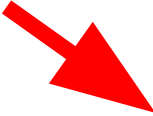


Figura 8.4 Grafo di assegnazione delle risorse.

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

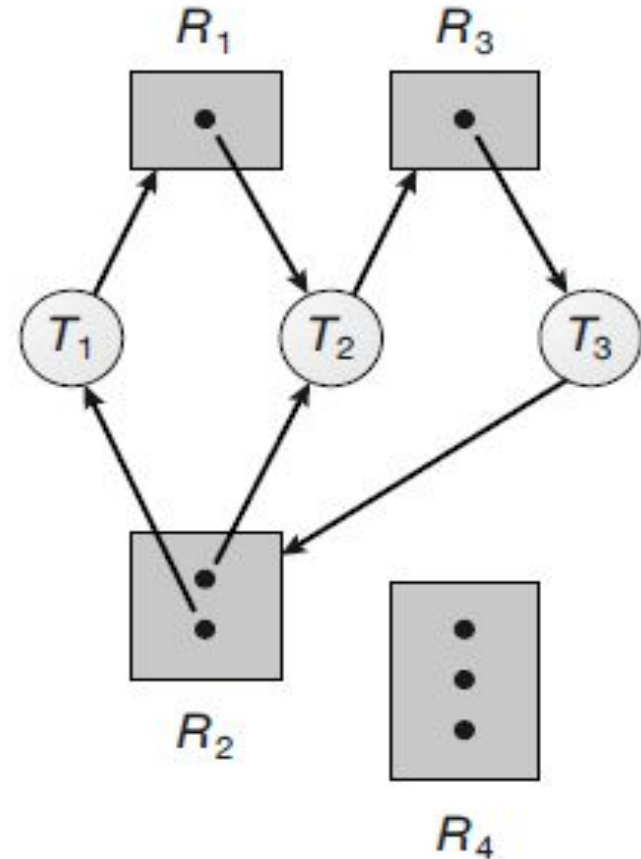


Figura 8.5 Grafo di assegnazione delle risorse con uno stallo.

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.

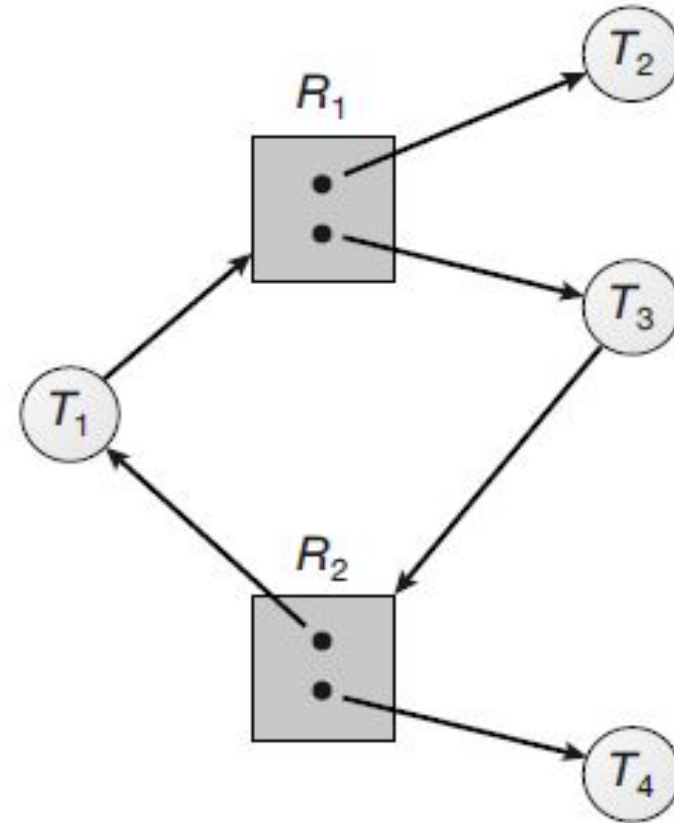


Figura 8.6 Grafo di assegnazione delle risorse con un ciclo, ma senza stallo.

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

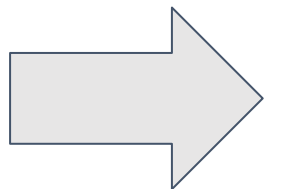
Assenza di
prelazione

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.

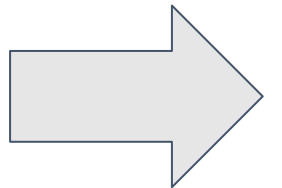
Mutua esclusione → Tutte le risorse devono essere condivisibili.
Impossibile in pratica, per esempio un lock mutex non può essere condiviso



Prevenire le situazioni di stallo

Possesso e attesa → Occorre garantire che un thread che richiede una risorsa non ne posseda 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 → imporre un ordinamento totale all'insieme di tutti i tipi di risorse e imporre che ciascun thread richieda le risorse in ordine crescente.

Soluzione pratica

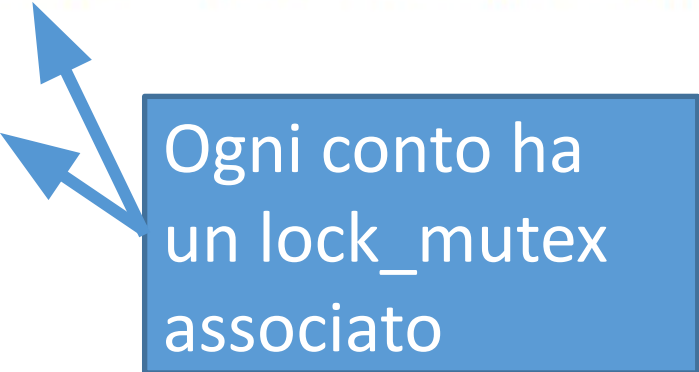
Ordinamento

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);  
}
```

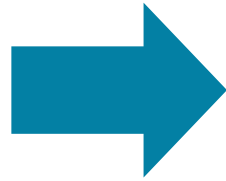


Ogni conto ha un lock_mutex associato

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

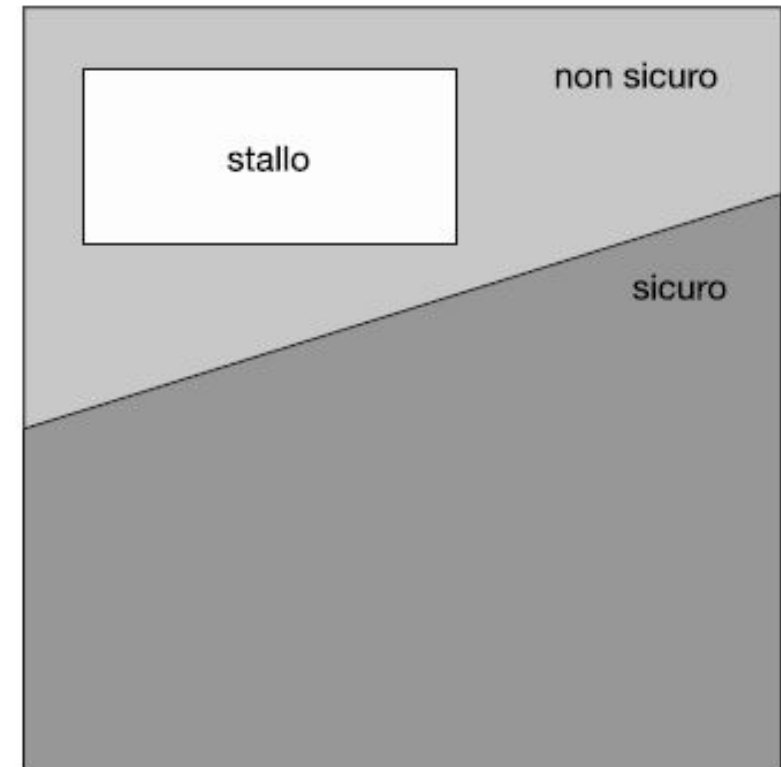


Figura 8.8 Spazi degli stati sicuri, non sicuri e di stallo.

Esempio: Stato sicuro

Risorse totali a disposizione: 12

Thread in esecuzione: 3

Situazione al tempo t_0 :

	Richieste massime	Unità possedute
T0	10	5
T1	4	2
T2	9	2

La sequenza $\langle T1, T0, T2 \rangle$ è sicura?

Esempio: Stato sicuro

Risorse totali a disposizione: 12

Thread in esecuzione: 3

Situazione al tempo t_0 :

	Richieste massime	Unità possedute
T0	10	5
T1	4	2
T2	9	2

La sequenza $\langle T1, T0, T2 \rangle$ 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 :

	Richieste massime	Unità possedute
T0	10	5
T1	4	2
T2	9	3

La sequenza $\langle T1, T0, T2 \rangle$ è sicura?

Esempio: Stato non sicuro

Risorse totali a disposizione: 12

Thread in esecuzione: 3

Situazione al tempo t_1 :

	Richieste massime	Unità possedute
T0	10	5
T1	4	2
T2	9	3

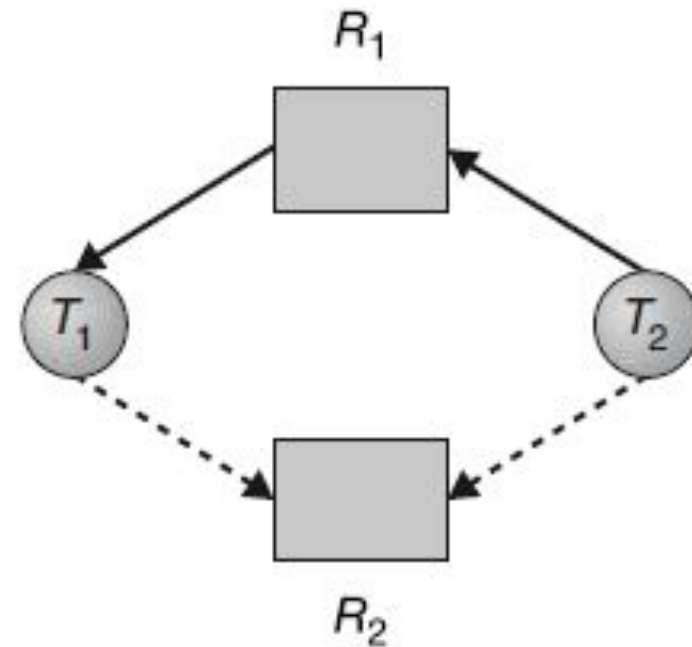
La sequenza $\langle T1, T0, T2 \rangle$ è sicura?

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

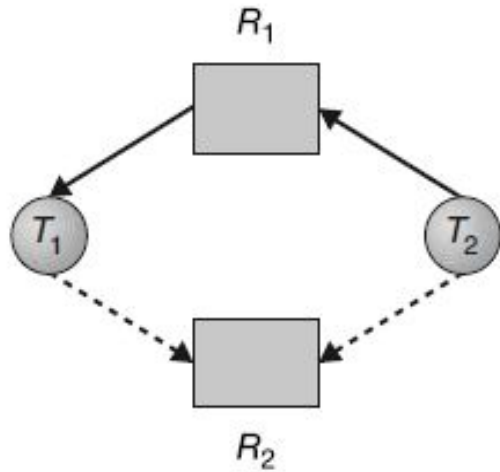
- Un **arco di rivendicazione** $T_i \rightarrow R_j$ indica che un thread T_i può richiedere la risorsa R_j
- Viene indicato con una freccia tratteggiata

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



Algoritmo con grafo di assegnazione delle risorse

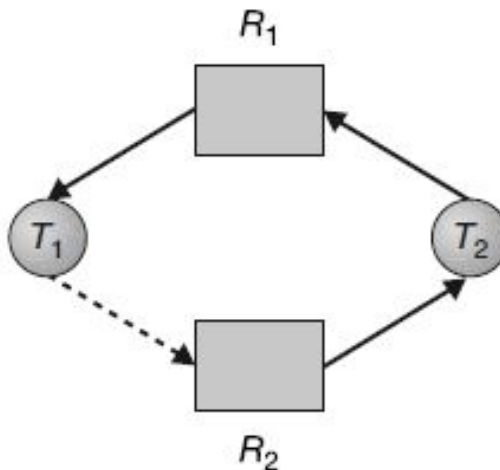
Si può applicare se ogni classe di risorse ha una sola istanza



- Sia T_1 che T_2 richiederanno in futuro R_2

- Si supponga 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 disponibili per ogni classe j
- ***Available*** $[j] = k$ indica che ci sono k istanze libere della risorsa R_j

Massimo

- Matrice di dimensione $n \times m$ 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 $n \times m$ 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 $n \times m$ 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.
- **$Need[i, j] = Max[i, j] - Allocation[i, j]$**

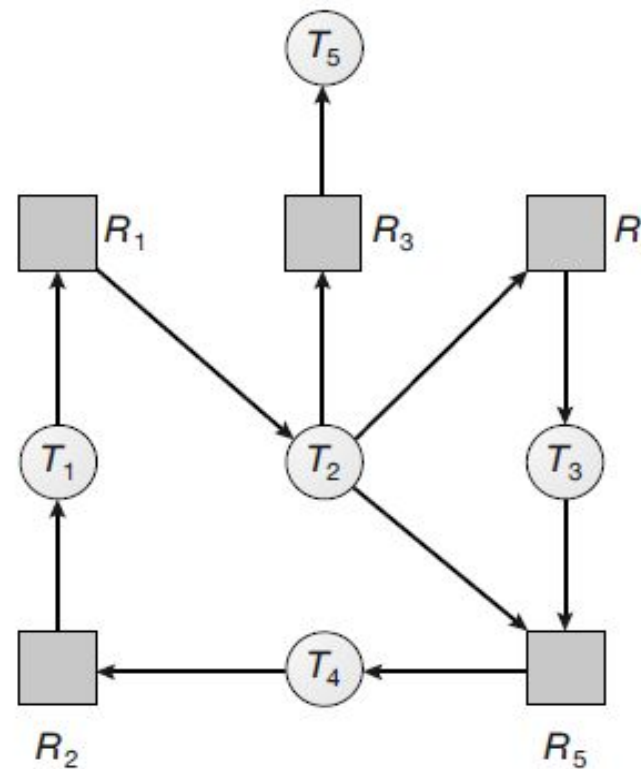
Rilevamento delle situazioni di stallo

Istanza singola di ciascun tipo di risorsa

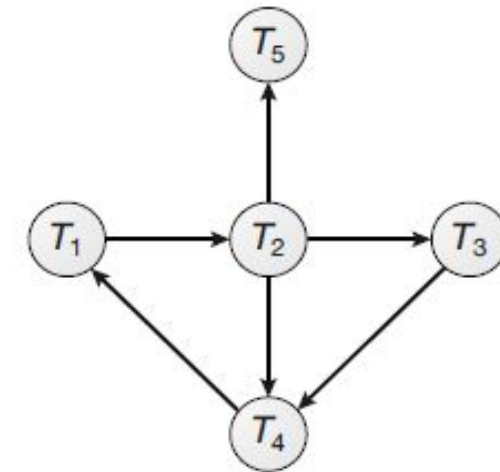
grafo di attesa



variante del grafo di assegnazione delle risorse



(a)



(b)

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

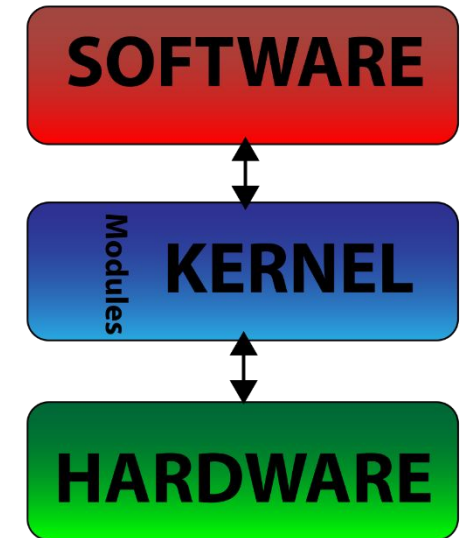
Attesa indefinita
(starvation)



**UNIVERSITÀ DEGLI STUDI
DELLA BASILICATA**

Corso di Sistemi Operativi

Stallo dei processi



Docente:
**Domenico Daniele
Bloisi**

