

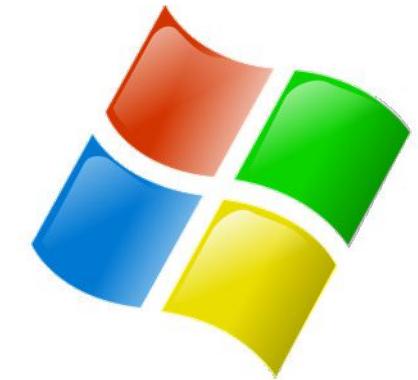
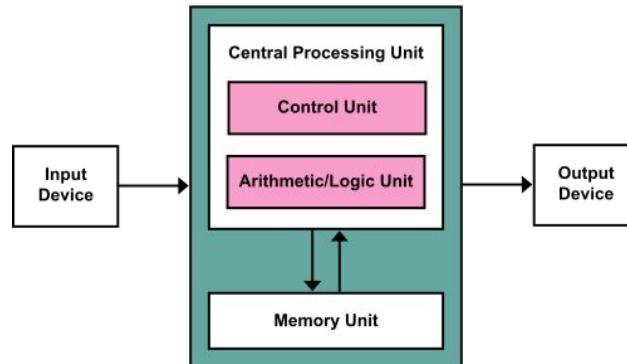


# UNIVERSITÀ DEGLI STUDI DELLA BASILICATA

*Corso di Sistemi Operativi*  
A.A. 2019/20

# Stallo dei processi

Novembre 2019



Docente:  
**Domenico Daniele  
Bloisi**



# Domenico Daniele Bloisi

---

- Ricercatore RTD B

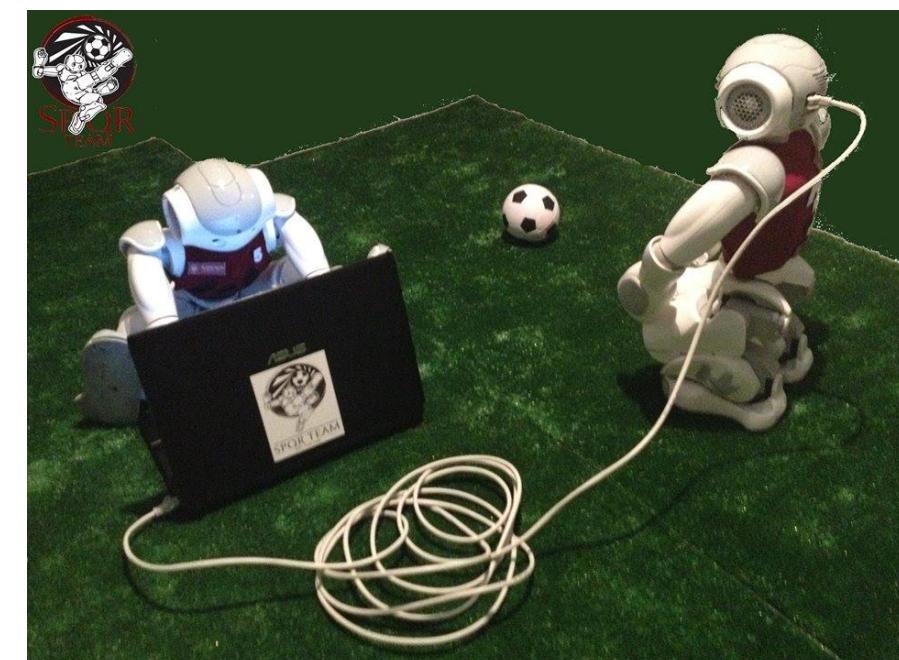
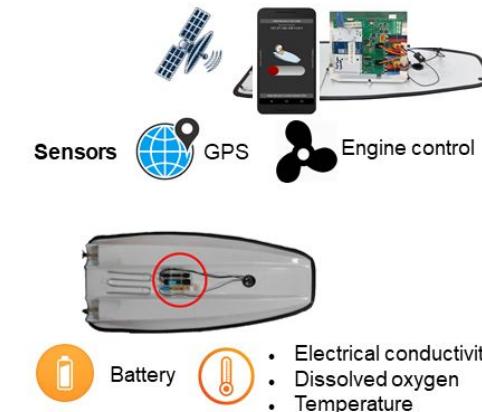
Dipartimento di Matematica, Informatica  
ed Economia

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>



# Ricevimento

---

- In aula, subito dopo le lezioni
- Martedì dalle 11:00 alle 13:00 presso:  
Campus di Macchia Romana  
[Edificio 3D](#) (Dipartimento di Matematica,  
Informatica ed Economia)  
[Il piano, stanza 15](#)

Email: [domenico.bloisi@unibas.it](mailto: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

---

In un ambiente con multiprogrammazione più **thread** possono competere per ottenere tali risorse

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.

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

---

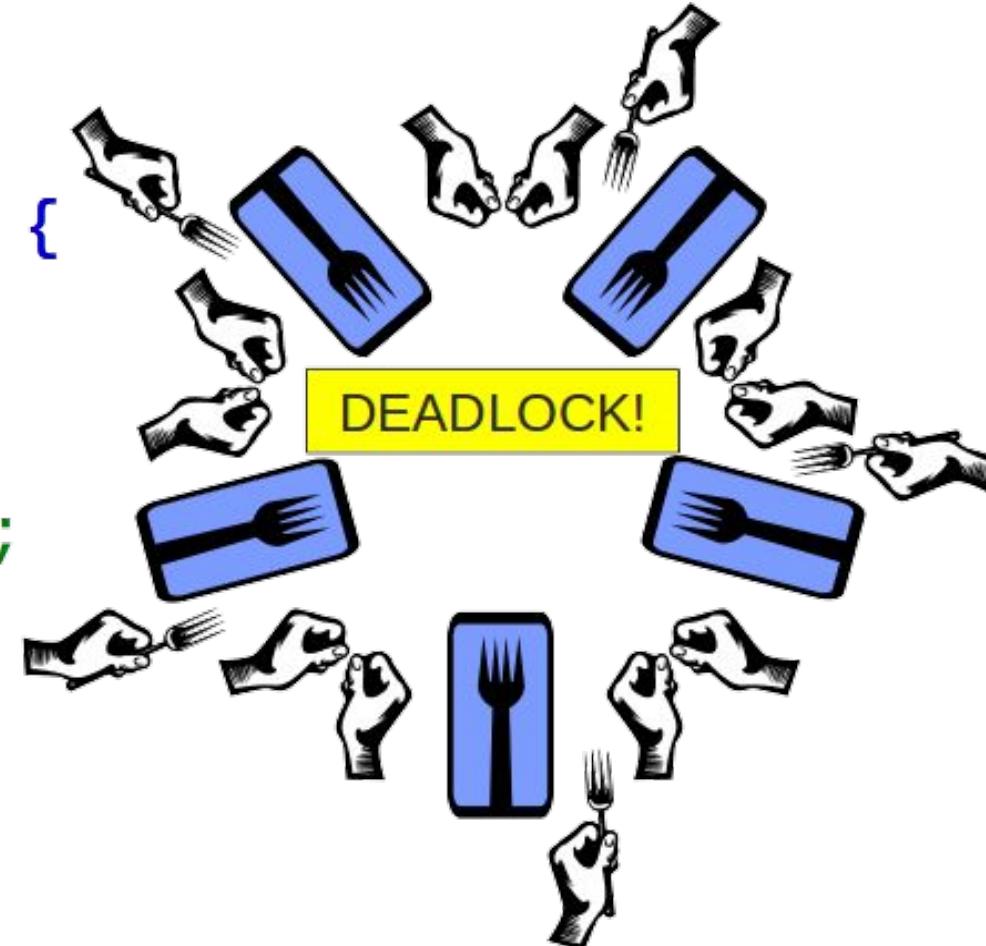
Se le risorse richieste da un thread T sono trattenute da altri thread, a loro volta nello stato di attesa, il thread T potrebbe non cambiare più il suo stato.

Situazioni di questo tipo sono chiamate di **stallo (deadlock)**

# Esempio di stallo

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat(); /* yummy */
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```



# 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.

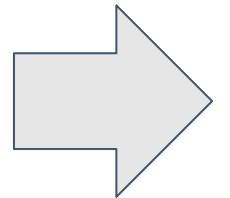
# Esempio di stallo in applicazioni multithread

```
#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);
}
```



# Esempio di stallo in applicazioni multithread

```
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;
}
```

# Esempio di stallo in applicazioni multithread

```
bloisi@bloisi-U36SG: ~/workspace/3.4-stallo-dei-processi
bloisi@bloisi-U36SG:~/workspace$ 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
```

# 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)

```
/* thread_one esegue in questa funzione */
void *do_work_one(void *param)
{
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&first_mutex);
        if (pthread_mutex_trylock(&second_mutex)) {
            /**
             * Fa qualcosa
             */
            pthread_mutex_unlock(&second_mutex);
            pthread_mutex_unlock(&first_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&first_mutex);
    }

    pthread_exit(0);
}
```

```
/* thread_two esegue in questa funzione */
void *do_work_two(void *param)
{
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&second_mutex);
        if (pthread_mutex_trylock(&first_mutex)) {
            /**
             * Fa qualcosa
             */
            pthread_mutex_unlock(&first_mutex);
            pthread_mutex_unlock(&second_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&second_mutex);
    }

    pthread_exit(0);
}
```

Figura 8.2 Esempio di stallo attivo.

# 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

# Situazioni di stallo

---

Condizioni necessarie per generare una situazione di stallo

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

---

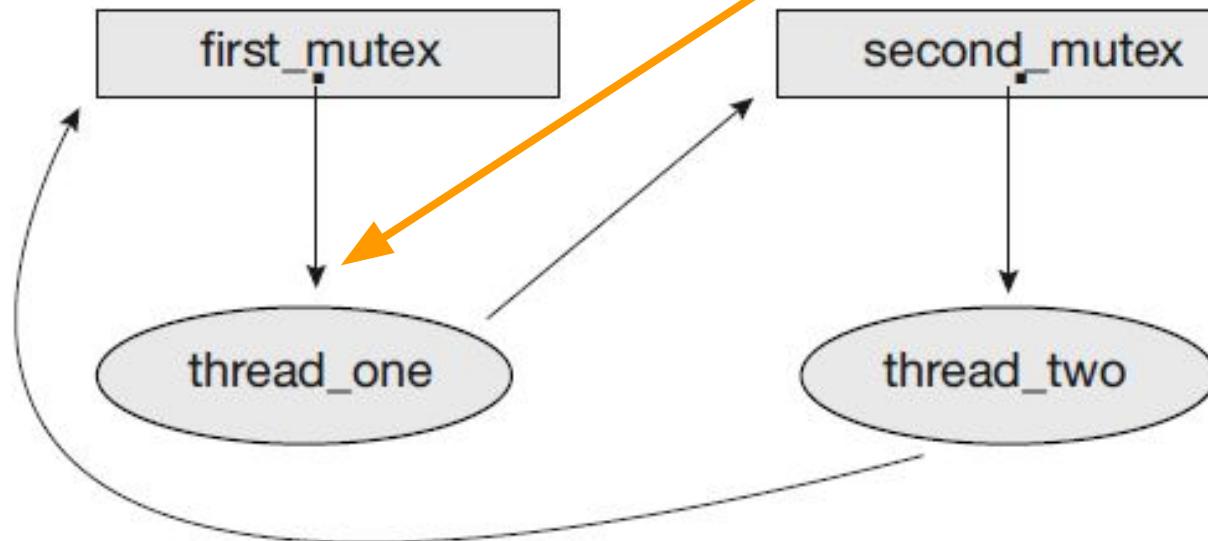
## Attesa circolare

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



arco di assegnazione da risorsa a thread

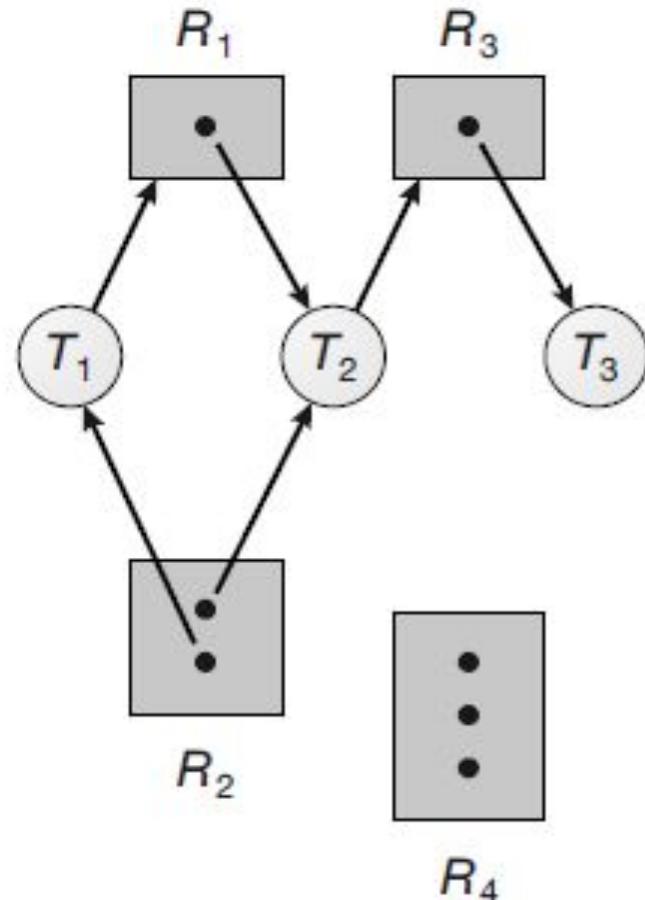
si usa un rettangolo per le classi di risorse (un puntino per ogni istanza)

si usa un cerchio per i thread

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

# Esempio - 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

---

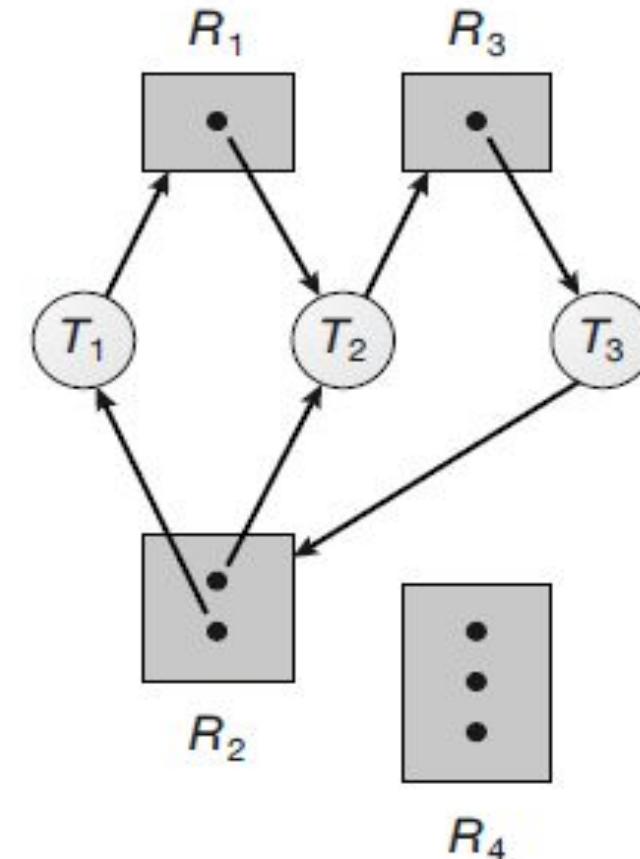
**Se il grafo di assegnazione delle risorse non contiene cicli, allora nessun thread del sistema subisce uno stallo.**

**Se il grafo contiene un ciclo, allora può sopraggiungere uno stallo.**

# Esempio - 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



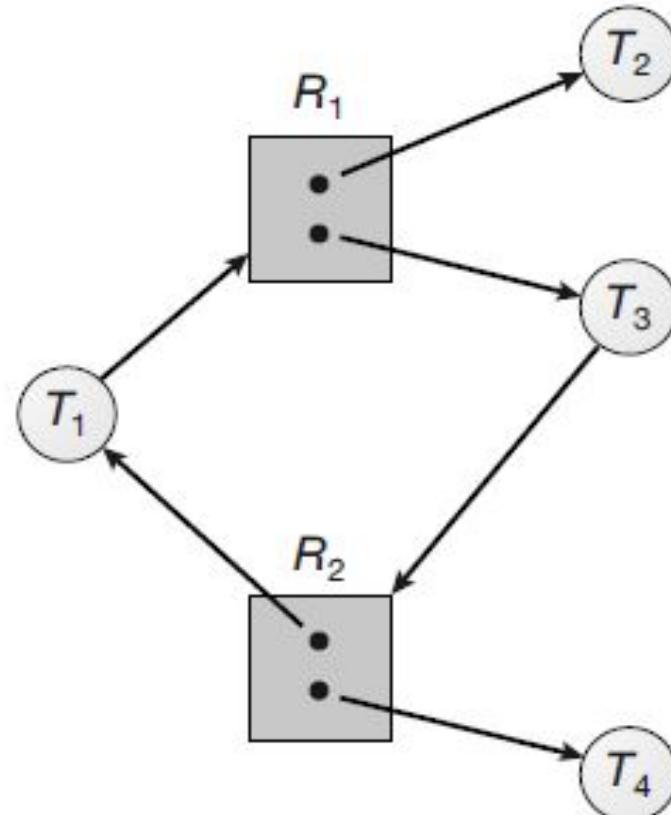
**Figura 8.5** Grafo di assegnazione delle risorse con uno stallo.

## Esempio - 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
2. usare un protocollo per *prevenire* o evitare le situazioni di stallo, assicurando che il sistema non entri *mai* in stallo
3. *permettere* al sistema di entrare in stallo, individuarlo e, quindi, eseguire il ripristino

**soluzione adottata  
da Linux e Windows**

**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

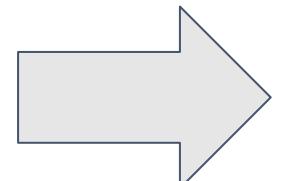
# Prevenire le situazioni di stallo

---

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**

**Posesso 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**



# 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  
trascation(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);

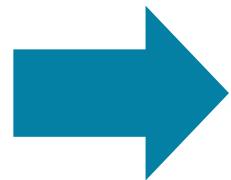
    release(lock2);
    release(lock1);
}
```

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

---

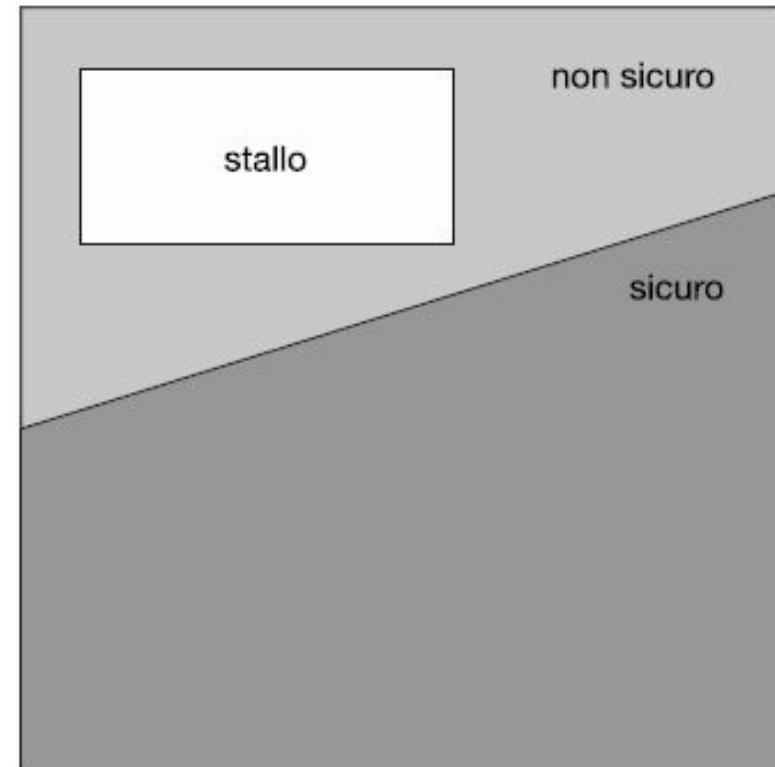
Uno **stato** si dice **sicuro** se il sistema è in grado di assegnare risorse a ciascun thread (fino al suo massimo) in un certo ordine e impedire il verificarsi di uno stallo.

# 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 t0:

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

La sequenza <T1,T0,T2> è sicura?

# Esempio: Stato sicuro

---

Risorse totali a disposizione: 12

Thread in esecuzione: 3

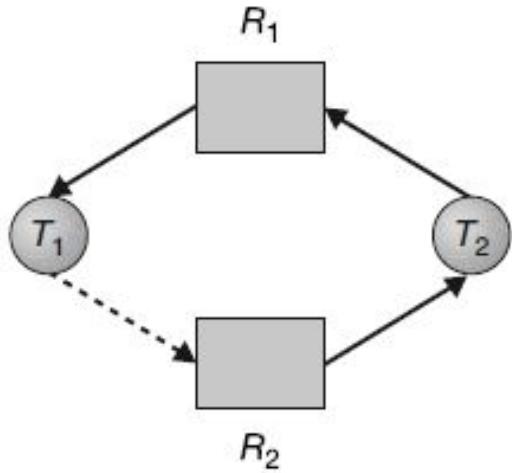
Situazione al tempo t1:

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

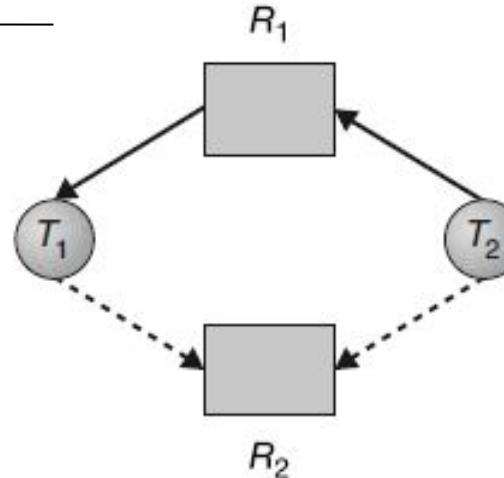
La sequenza <T1,T0,T2> è sicura?

# 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



- 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*

# Rilevamento delle situazioni di stallo

Istanza singola di ciascun tipo di risorsa

grafo d'attesa



variante del grafo  
di assegnazione  
delle risorse

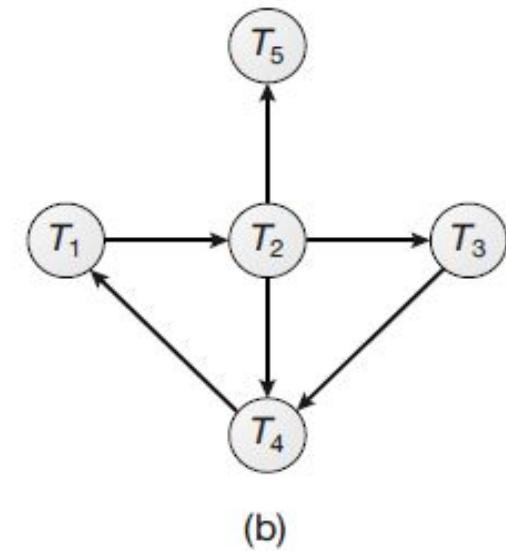
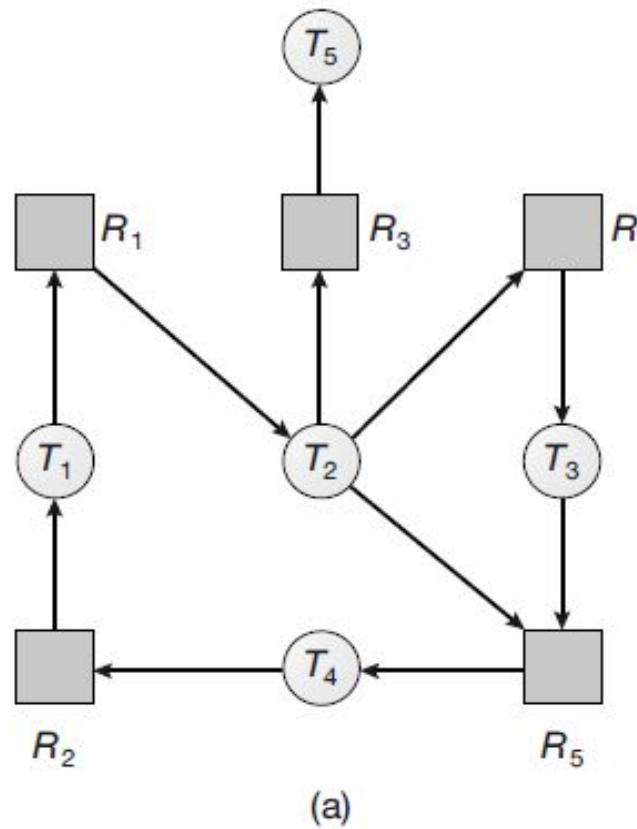


Figura 8.11 (a) Grafo di assegnazione delle risorse; (b) Grafo d'attesa corrispondente.

# Rilevamento delle situazioni di stallo

---

## Più istanze di ciascun tipo di risorsa

Lo **schema con grafo d'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)

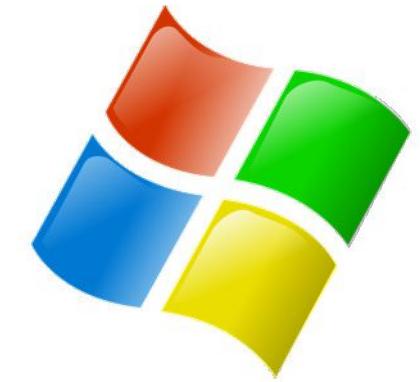
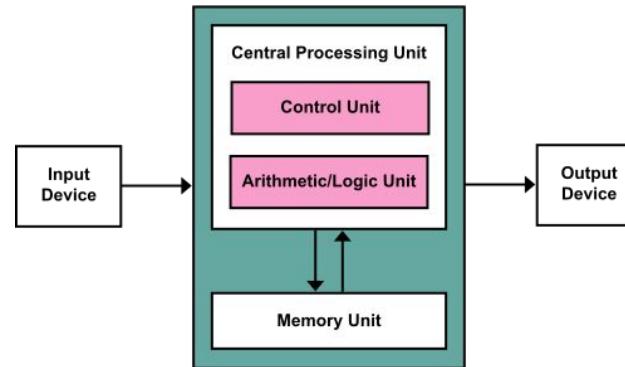


# UNIVERSITÀ DEGLI STUDI DELLA BASILICATA

*Corso di Sistemi Operativi*  
A.A. 2019/20

# Stallo dei processi

Novembre 2019



Docente:  
Domenico Daniele  
Bloisi

