



Sistemi operativi e programmazione concorrente

Docente: Antonio Vito Ciliberto

A.A. 2023/2024

Informazioni sul corso

- ▶ Home page del corso: <https://learn.unimol.it/course/view.php?id=7209>
- ▶ Docente: Antonio Vito Ciliberto
- ▶ Periodo: Il semestre marzo 2024 – luglio 2024
 - ▶ Martedì dalle 14:00 alle 17:00 aula Bird (secondo piano)
 - ▶ Giovedì dalle 14:00 alle 17:00 aula Bird (secondo piano)
 - ▶ Venerdì dalle 10:00 alle 13:00 aula Bird (secondo piano)

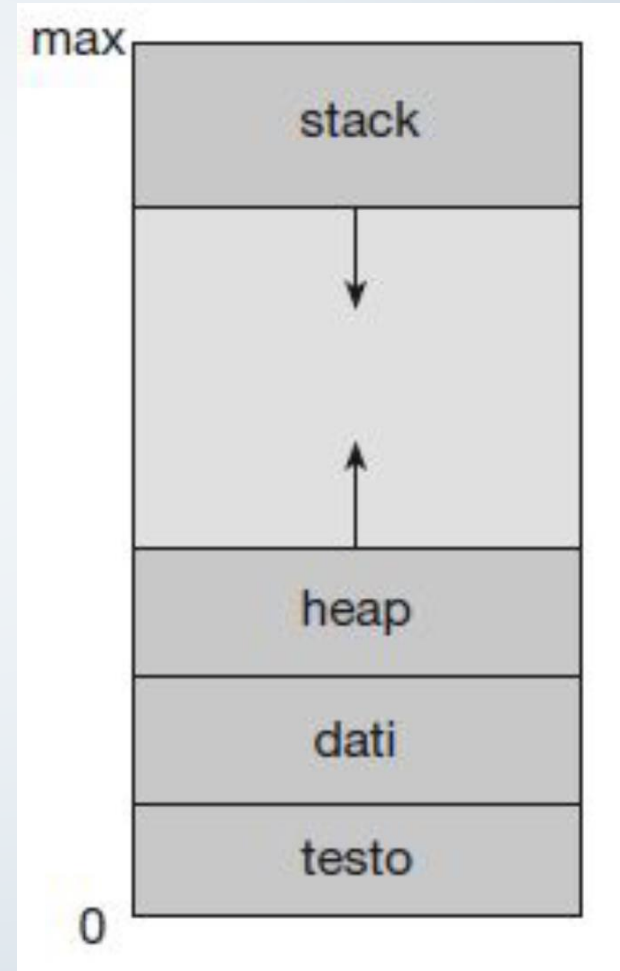
Ricevimento

- ▶ In presenza, durante il periodo delle lezioni: da concordare con il docente tramite mail
- ▶ Tramite google meet : da concordare con il docente tramite mail
- ▶ Per prenotare un appuntamento inviare una email a
 - ▶ antonio.ciliberto@unimol.it

Concetto di processo

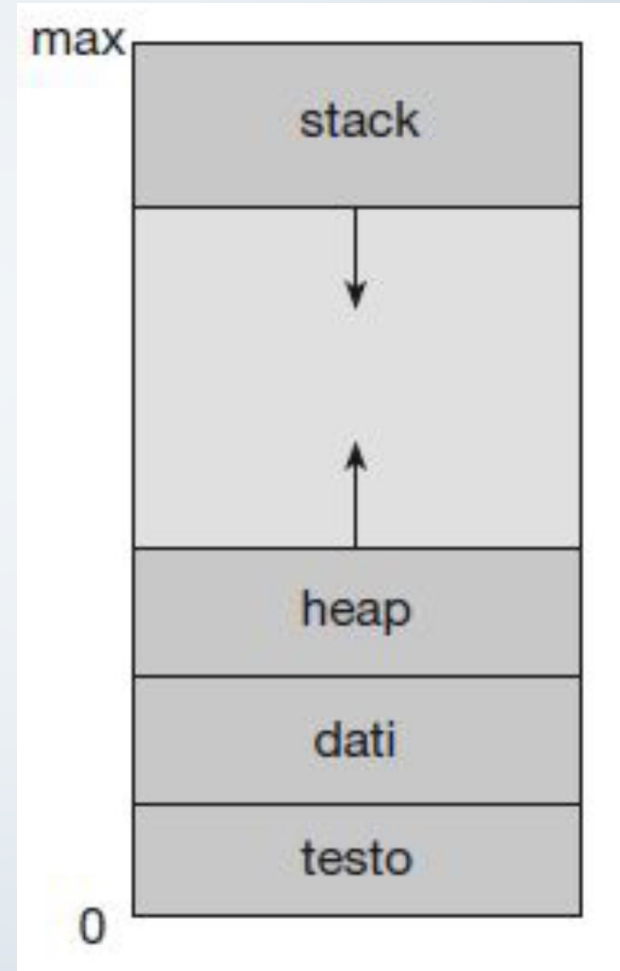
Un sistema batch (lotti) esegue job (lavori), mentre un sistema time-sharing esegue programmi utente o task; queste attività sono simili per molti aspetti, perciò sono chiamate **processi**.

Un **processo** è un programma in esecuzione. La struttura di un processo in memoria è generalmente suddivisa in più sezioni.



Sezioni di processo

- ▶ **Stack:** memoria temporaneamente utilizzata durante le chiamate di funzioni.
- ▶ **Heap:** memoria allocata dinamicamente durante l'esecuzione del programma
- ▶ **Dati:** contenente le variabili globali
- ▶ **Testo:** contenente il codice eseguibile



Stato di un processo

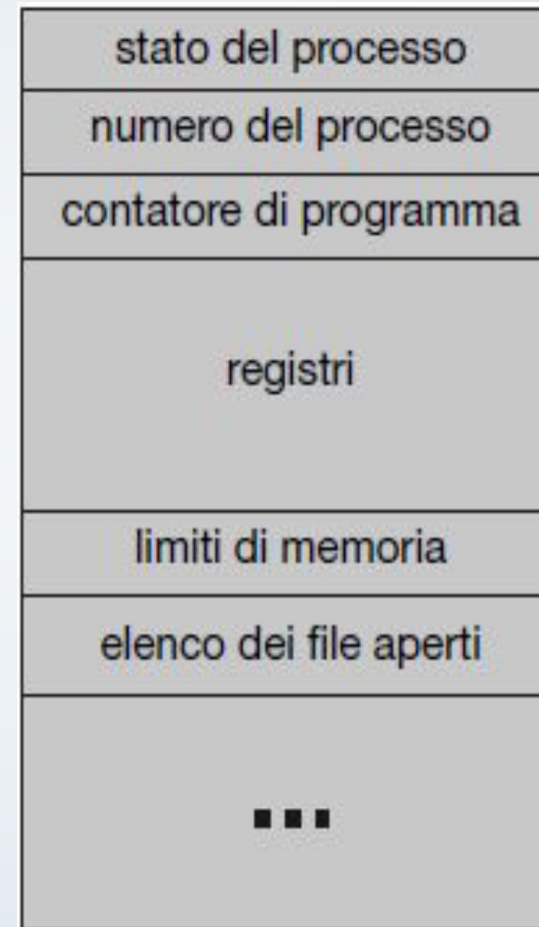
- ▶ **Nuovo:** Si crea il processo
- ▶ **Esecuzione (running):** Le sue istruzioni vengono eseguite
- ▶ **Attesa (waiting):** Il processo attende che si verifichi qualche evento
- ▶ **Pronto (ready):** Il processo attende di essere assegnato a un'unità di elaborazione
- ▶ **Terminato:** Il processo termina l'esecuzione

PCB – Process Control Block

- ▶ Ogni processo è rappresentato nel sistema operativo da un **blocco di controllo** (*process control block, PCB, o task control block, TCB*)
- ▶ Il PCB contiene un insieme di informazioni connesse a un processo specifico.

PCB – Process Control Block

- ▶ **Stato del processo:** nuovo, pronto, esecuzione, attesa, arresto
- ▶ **Contatore di programma:** che contiene l'indirizzo della successiva istruzione da eseguire per tale processo.
- ▶ **Registri della CPU:** accumulatori, registri indice, puntatori alla cima dello stack (*stack pointer*), registri di uso generale e registri contenenti i codici di condizione (*condition codes*)



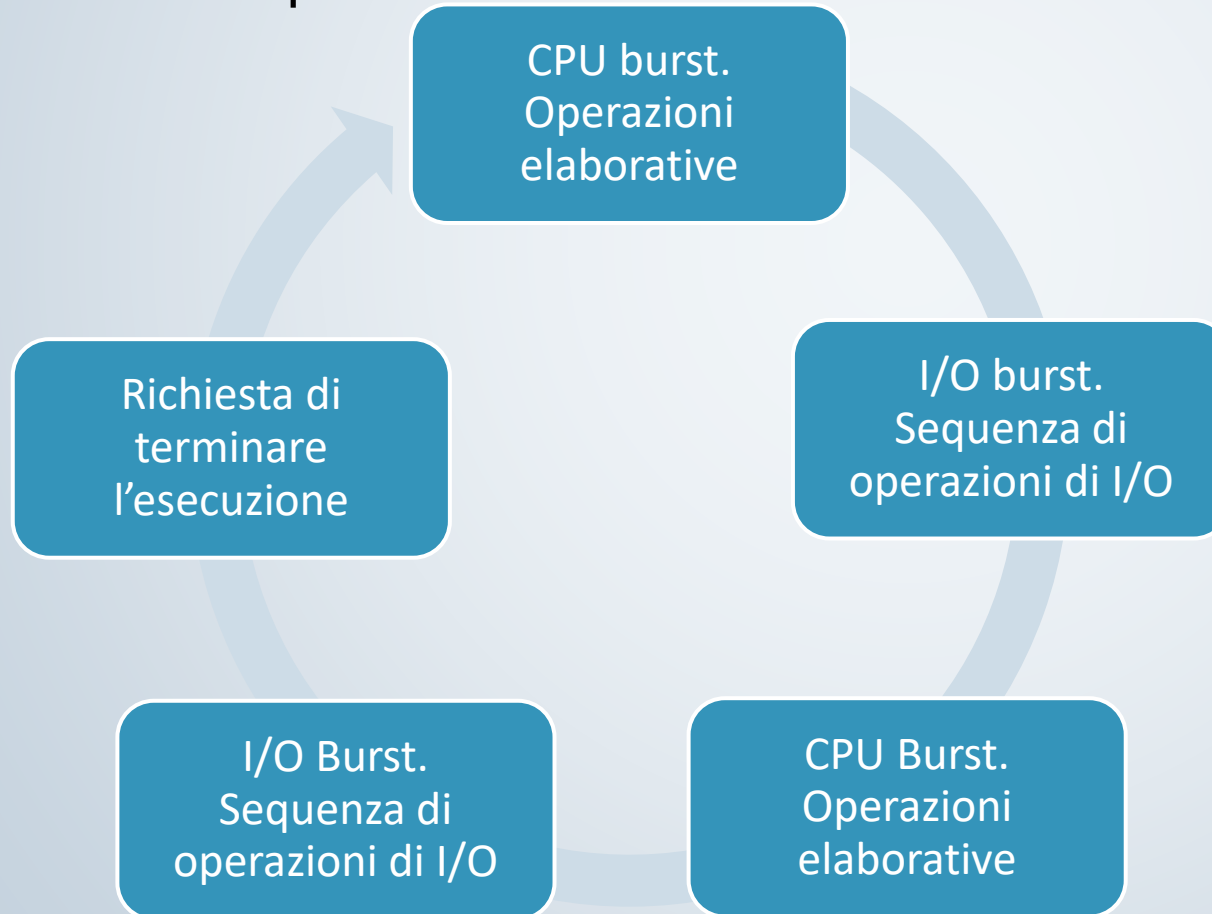
Scheduling dei processi

- ▶ Lo **scheduler dei processi** seleziona un processo da eseguire dall'insieme di quelli disponibili.
 - ❑ **Processo I/O bound** : impiega la maggior parte del proprio tempo nell'esecuzione di operazioni di I/O
 - ❑ **Processo CPU bound** : impiega la maggior parte del proprio tempo nelle elaborazioni

...dalle puntate precedenti!

Ciclicità delle fasi d'elaborazione

L'esecuzione di un processo consiste in un ciclo di elaborazione:



Scheduler della CPU

Lo scheduler interviene nelle seguenti situazioni:

1. Un processo
passa dallo stato di
esecuzione allo
stato di attesa

2. Un processo
passa dallo stato di
esecuzione allo
stato pronto

3. Un processo
passa dallo stato di
attesa allo stato
pronto

4. Un processo
termina

Scheduler della CPU

Lo scheduler interviene nelle seguenti situazioni:

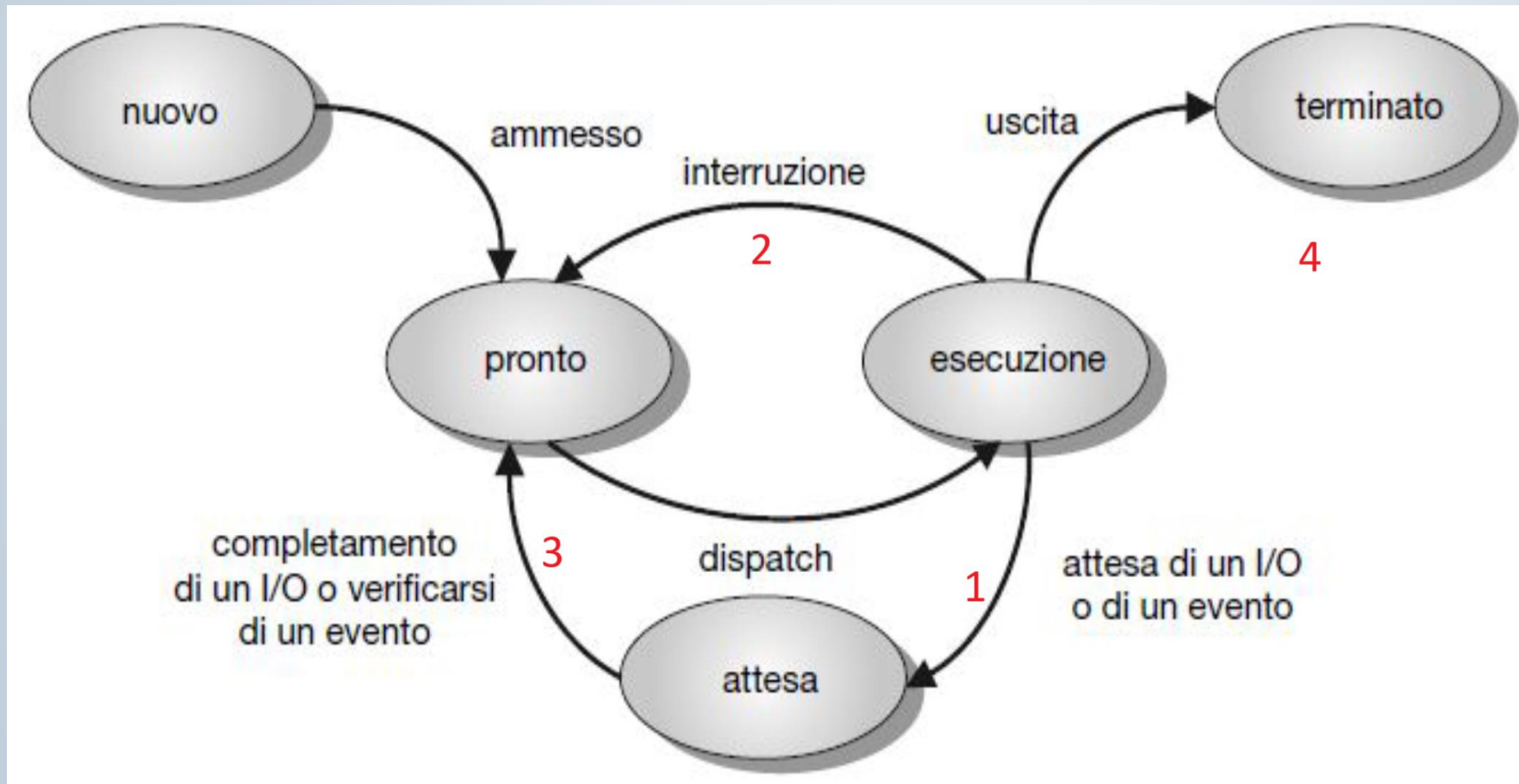
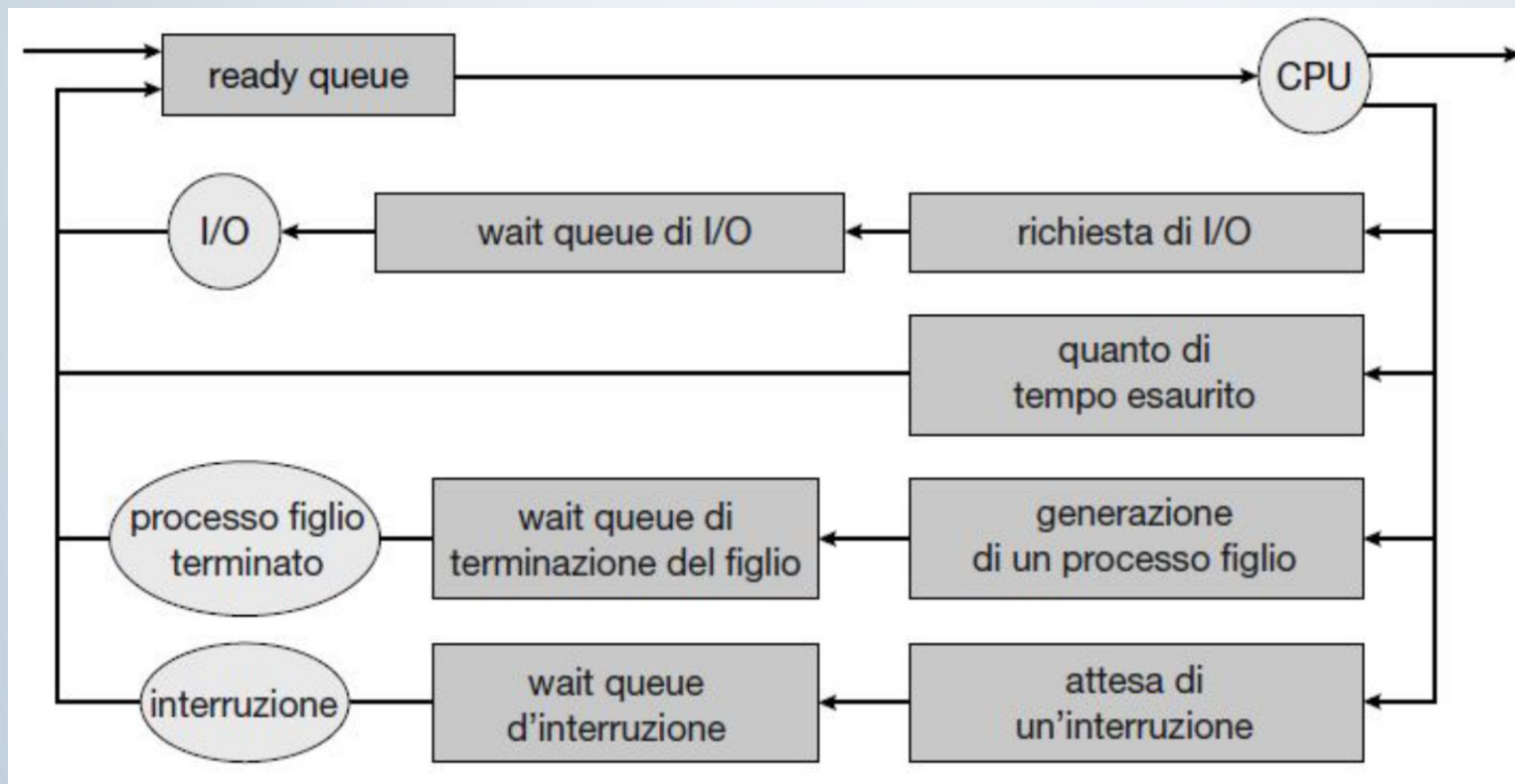


Diagramma di accodamento

Una comune rappresentazione dello scheduling dei processi è data da un **diagramma di accodamento**

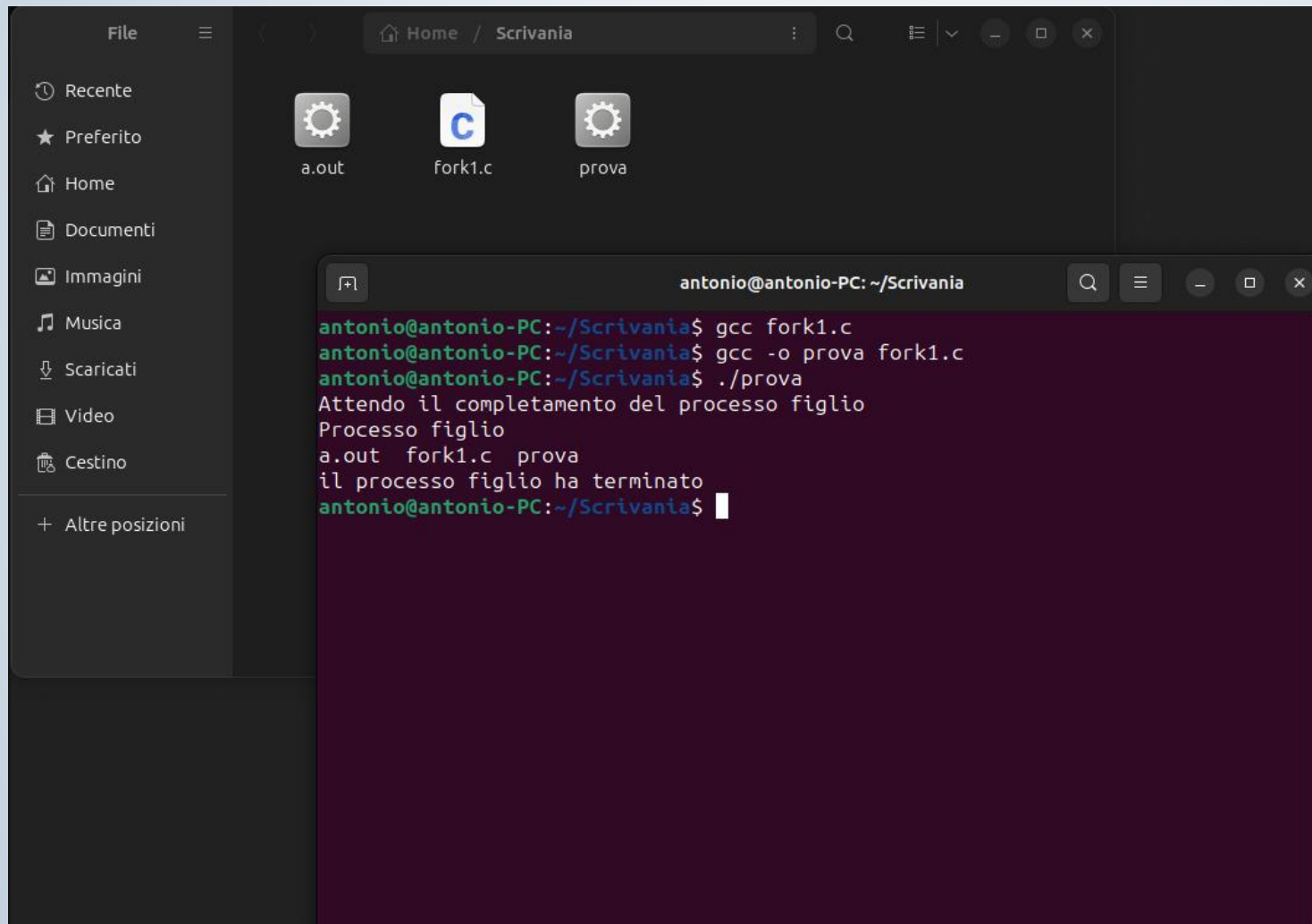


Creazione dei processi – fork() Unix

```
#include<stdio.h>
#include<sys/types.h>
#include <sys/wait.h>
#include<unistd.h>
int main(){
    pid_t pid;
    pid = fork();
    if(pid<0){
        fprintf(stderr, "generazione del nuovo processo fallita");
        return 1;
    }else if(pid==0){
        printf("Processo figlio\n");
        execl("/bin/ls", "ls", NULL);
    } else {
        printf("Attendo il completamento del processo figlio\n");
        wait(NULL);
        printf("il processo figlio ha terminato\n");
    }
    return 0;
}
```


...dalle puntate precedenti!

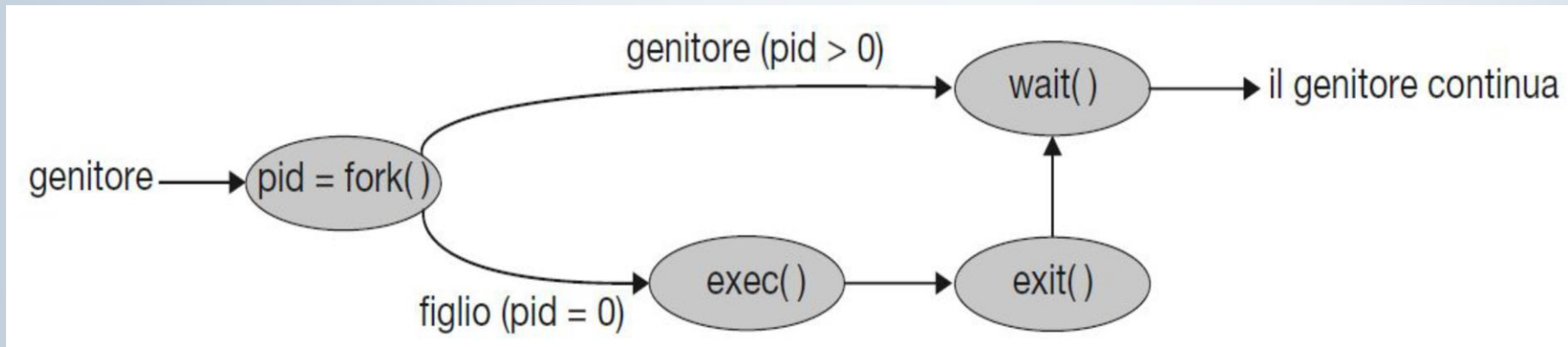
Creazione dei processi – fork() Unix



The screenshot shows a Linux desktop environment. On the left is a sidebar with a 'File' menu and a list of locations: Recente, Preferito, Home, Documenti, Immagini, Musica, Scaricati, Video, Cestino, and Altre posizioni. The main window is a file manager showing the contents of the 'Scrivania' (Desktop) directory. It contains three files: 'a.out' (represented by a gear icon), 'fork1.c' (represented by a document icon with a 'C'), and 'prova' (represented by a gear icon). Overlaid on the file manager is a terminal window titled 'antonio@antonio-PC: ~/Scrivania'. The terminal shows the following commands and output:

```
antonio@antonio-PC:~/Scrivania$ gcc fork1.c
antonio@antonio-PC:~/Scrivania$ gcc -o prova fork1.c
antonio@antonio-PC:~/Scrivania$ ./prova
Attendo il completamento del processo figlio
Processo figlio
a.out fork1.c prova
il processo figlio ha terminato
antonio@antonio-PC:~/Scrivania$
```


Creazione dei processi – `fork()` Unix



Terminazione dei processi

- ▶ Un processo termina quando finisce l'esecuzione della sua ultima istruzione e inoltra la richiesta al sistema operativo di essere cancellato usando la **chiamata di sistema exit()**
- ▶ Un processo genitore può porre termine all'esecuzione di uno dei suoi processi figli per diversi motivi:

Il processo figlio ha ecceduto nell'uso di alcune tra le risorse che gli sono state assegnate

Il compito assegnato al processo figlio non è più richiesto

Il processo genitore termina e il sistema operativo non consente a un processo figlio di continuare l'esecuzione in tale circostanza

Comunicazione tra processi - IPC

- ▶ Un processo può essere **indipendente** nel momento in cui non influisce su altri processi del sistema o subisce l'influsso da altri processi, ne tanto meno condivide dati temporanei o permanenti con altri processi.
- ▶ Un processo può essere **cooperativo** se influenza o può essere influenzato da altri processi in esecuzione, condividendo con altri processi dati temporanei o permanenti.

Un ambiente che consente la cooperazione tra processi può essere utile per svariate ragioni:

1. Condivisione di informazioni: più utenti possono condividere gli stessi dati;
2. Velocizzazione di calcolo: alcune attività sono più rapidamente realizzabili;
3. Modularità: le funzionalità di sistema possono essere suddivise in processi distinti.

Come avviene la comunicazione

Due modelli fondamentali:

- ▶ **A memoria condivisa:** si stabilisce una zona di memoria condivisa dai processi cooperanti, che possono così comunicare scrivendo e leggendo da tale zona.
- ▶ **A scambio di messaggi:** in questo modello la comunicazione avviene per mezzo di messaggi. Utile per trasmettere piccole quantità di dati, ed è molto facile da realizzare.

Nei sistemi operativi sono diffusi entrambi i modelli, spesso coesistono in un unico sistema.

IPC a memoria condivisa

Un processo chiamato **producer** produce informazioni che sono consumate da un processo chiamato **consumer**.

Es. compilatore produce codice assembly, consumato dall'assemblatore per generare oggetti che a loro volta saranno consumati dal loader.

- ▶ Problema del producer/consumer: l'esecuzione concorrente dei due processi richiede la presenza di un buffer che possa essere riempito dal producer e svuotato dal consumer.
 - ▶ Buffer illimitato
 - ▶ Buffer limitato

Sincronizzazione tra processi

Come visto nella slide precedente la comunicazione tra processi avviene per mezzo di chiamate alle primitive `send()` e `receive()`. Ma lo scambio di messaggi può a sua volta essere:

- ▶ Sincrono (o bloccante)
 - ▶ Invio sincrono: il processo che invia il messaggio si blocca nell'attesa che il processo ricevente o la mailbox riceva il messaggio.
 - ▶ Ricezione sincrona: il ricevente si blocca nell'attesa dell'arrivo di un messaggio.
- ▶ Asincrono (o non bloccante)
 - ▶ Invio asincrono: il processo invia il messaggio e riprende la propria esecuzione.
 - ▶ Ricezione asincrona: il ricevente riceve un messaggio valido o un valore nullo.

Producer con scambio di messaggi

```
message next_produced;  
  
while (true) {  
    /* produce un elemento in next_produced */  
  
    send(next_produced);  
}
```


Producer con scambio di messaggi

```
message next_consumed;  
  
while (true) {  
    receive(next_consumed);  
  
    /* consuma l'elemento in next_consumed */  
}
```

Programma del corso

- ▶ Introduzione ai sistemi operativi. Attività e struttura di un sistema operativo.
- ▶ I sistemi a processi. Comunicazione: condivisione di memoria, scambio di messaggi. Thread e concorrenza. Scheduling della CPU.
- ▶ **I semafori. Cooperazione e sincronizzazione. Deadlock.**
- ▶ Gestione dell'unità centrale.
- ▶ Gestione della memoria di massa.
- ▶ Il file system. Attributi, operazioni e metodi di accesso.
- ▶ Sicurezza. Protezione.

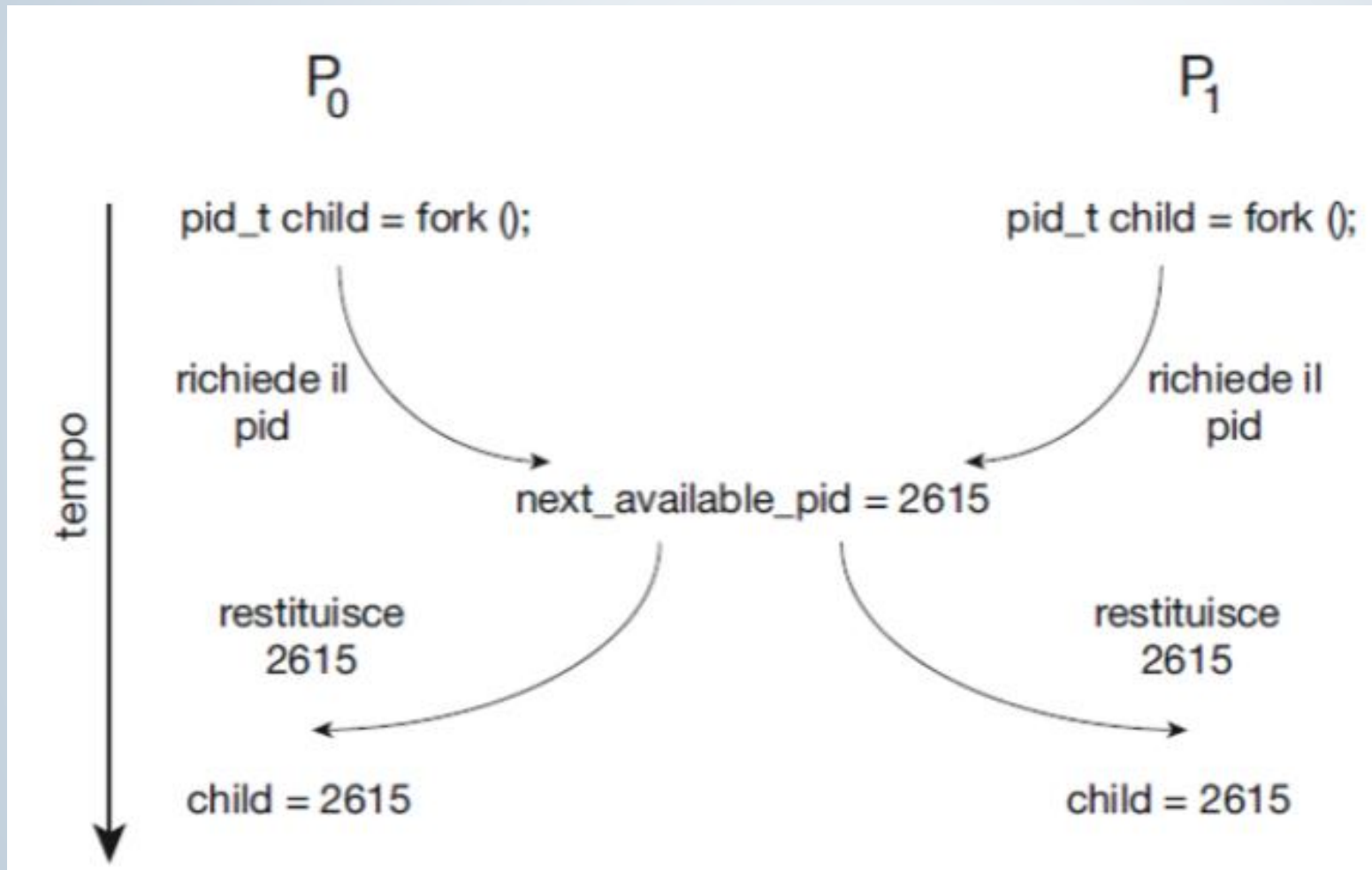
Processo cooperante

- ▶ Definizione:
 - ▶ **Processo cooperante:** è un processo che può influenzarne un altro in esecuzione nel sistema o anche subirne l'influenza.
- ▶ **I processi cooperanti** possono condividere direttamente uno spazio logico di indirizzi oppure condividere dati soltanto attraverso file o messaggi.
- ▶ **L'accesso concorrente** a dati condivisi può tuttavia causare situazioni di incoerenza degli stessi dati.

Producer / Consumer

- ▶ Definizione problema:
 - ▶ Producer scrive il messaggio all'istante «t0»
 - ▶ Consumer legge il messaggio all'istante «t0»
- ▶ Cosa si verifica?
- ▶ Come si può risolvere questa situazione?

Risultati inattesi



Producer / Consumer v.2

- ▶ Aggiunta di una variabile di tipo intero (contatore)
- ▶ Ogni volta che il producer esegue delle istruzioni la variabile intera viene incrementata.
- ▶ Ogni volta che il consumer esegue delle istruzioni la variabile intero viene decrementata.

Producer / Consumer v.2 es

- ▶ Supponiamo che il valore iniziale della variabile contatore sia 5
- ▶ L'esecuzione separata di producer e consumer lascia invariato il valore del contatore.
- ▶ Supponiamo anche che durante l'esecuzione il valore del contatore sia contenuto in un registro fisico condiviso tra il producer e il consumer (registro1 & registro2)

Producer esecuzione tipo

- ▶ Producer esegue `contatore++` ad alto livello ma a livello macchina questa operazione potrebbe essere implementata come segue:

- ▶ `registro1 = contatore`

(registro1 = 5 «» contatore = 5)

- ▶ `registro1 = registro1 + 1`

(registro1 = 6 «» contatore = 5)

- ▶ `contatore = registro1`

(registro1 = 6 «» contatore = 6)

Consumer esecuzione tipo

- ▶ Consumer esegue contatore-- ad alto livello ma a livello macchina questa operazione potrebbe essere implementata come segue:

- ▶ registro2 = contatore

(registro2 = 5 «» contatore = 5)

- ▶ registro2 = registro2 - 1

(registro2 = 4 «» contatore = 5)

- ▶ contatore = registro2

(registro2 = 4 «» contatore = 4)

Producer / Consumer v.2 es

- ▶ L'esecuzione concorrente di producer e consumer equivale a un'esecuzione sequenziale delle istruzioni introdotte precedentemente, intercalate tra loro in qualunque sequenza che però conservi l'ordine interno di ogni singola istruzione.

Producer / Consumer v.2 es

Istante	Soggetto	Operazione	Registro	Contatore
T0	Producer	registro1 = contatore	5	5
T1	Producer	registro1 = registro1 + 1	6	5
T2	Consumer	registro2 = contatore	5	5
T3	Consumer	registro2 = registro2 - 1	4	5
T4	Producer	contatore = registro1	6	6
T5	Consumer	contatore = registro2	4	4

Valore del contatore pari a 4 o contatore pari a 6

Race condition

- ▶ Il problema della **sezione critica** consiste nel progettare un protocollo che i processi possano usare per cooperare.
- ▶ Una **sezione critica** è un segmento di codice in cui il processo può modificare variabili comuni, scrivere file e così via.
- ▶ 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

Race condition

- ▶ In sostanza:
 - ▶ **Sezione d'ingresso:** ogni processo deve chiedere il permesso per entrare nella propria sezione critica.
 - ▶ **Sezione d'uscita:** fase finale dell'esecuzione della sezione critica
 - ▶ **Sezione non critica:** restante parte del codice.

```
while (true) {  
    sezione d'ingresso  
    sezione critica  
    sezione d'uscita  
    sezione non critica  
}
```

Problema della sezione critica

- ▶ Una possibile soluzione del problema della sezione critica deve soddisfare i tre seguenti requisiti:
 1. **Mutua Esclusione:** se il processo P_i è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
 2. **Progresso:** se un processo desidera entrare nella propria sezione critica deve necessariamente trovarsi in una sezione non critica della sua esecuzione.
 3. **Attesa limitata:** dopo una richiesta di accesso esiste un numero limitato di richieste da eseguire prima di accordare la richiesta.

Gestione della sezione critica

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

- ▶ **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:** non consente di applicare la prelazione a un processo attivo in modalità di sistema. È immune da race condition.

Soluzione Peterson

La soluzione Peterson è una proposta **software** per la soluzione del problema della sezione critica:

1. Limitato a 2 processi P_i ($i = 0$) , P_j ($j = 1 - i$)
2. Ogni processo esegue alternativamente sezione critica e sezione non critica
3. Condivisione di 2 variabili
int turn
boolean flag[2]

Soluzione Peterson

```
while (true) {  
  
    flag[i]= true;  
    turn = j;  
    while (flag[j] && turn == j);  
  
    /*sezione critica*/  
  
    flag[i] = false;  
  
    /*sezione non critica*/  
  
}
```

Soluzione Peterson

- ▶ La variabile `turn` segnala di chi sia il turno d'accesso alla sezione critica, quindi se `turn == i` allora il processo P_i è autorizzato a eseguire la propria sezione critica.
- ▶ L'array `flag` indica se un processo sia pronto ad entrare nella propria sezione critica. Se `flag[i]` è true, P_i è pronto a entrare nella propria sezione critica.

Soluzione Peterson – dim.1

- ▶ La mutua esclusione è rispettata?
- ✓ Ogni processo P_i accede alla propria sezione critica solo se $\text{flag}[j] == \text{false}$ oppure $\text{turn} == i$.
- ✓ Se entrambi i processi sono eseguibili in concomitanza nelle rispettive sezioni critiche allora $\text{flag}[0] == \text{flag}[1] == \text{true}$.
- ▶ Sotto queste due condizioni si deduce che la variabile turn non può contemporaneamente valere sia 0 che 1. di conseguenza un solo processo per volta può entrare nella rispettiva sezione critica.

Soluzione Peterson – dim.2

- ▶ Il requisito di progresso e attesa limitata sono rispettati?
- ✓ L'ingresso di un processo P_i nella propria sezione critica può essere impedito solo se l'altro processo è bloccato nella sua interazione con $\text{flag}[j] == \text{true}$ e $\text{turn} == j$.
- ✓ Se P_j non è pronto per entrare nella sezione critica allora $\text{flag}[j] == \text{false}$ e quindi P_i può accedere alla sezione critica.

Soluzione Peterson – dim.2

- ▶ Il requisito di progresso e attesa limitata sono rispettati?
- ✓ Se P_j sta eseguendo la propria iterazione, se $turn == i$ allora sarà P_i ad entrerà nella propria sezione critica.
- ✓ Se P_j sta eseguendo la propria iterazione, se $turn == j$ allora sarà P_i ad entrerà nella propria sezione critica.
- ✓ In entrambi i casi quando P_j uscirà dalla propria sezione critica reimposterà il valore di flag, permettendo a P_i di entrare nella propria sezione critica.

Soluzione Peterson Vs riordino

- ▶ I processori moderni per migliorare le prestazioni possono riordinare le operazioni di lettura e scrittura che non hanno dipendenze.
- ▶ Si considerino per esempio 2 thread su un unico processo (condivisione dei dati – variabili globali)
 - ▶ Boolean flag = false;
 - ▶ Int x = 0;

Soluzione Peterson Vs riordino

- ▶ Thread 1 esegue :

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

- ▶ Thread 2 esegue :

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

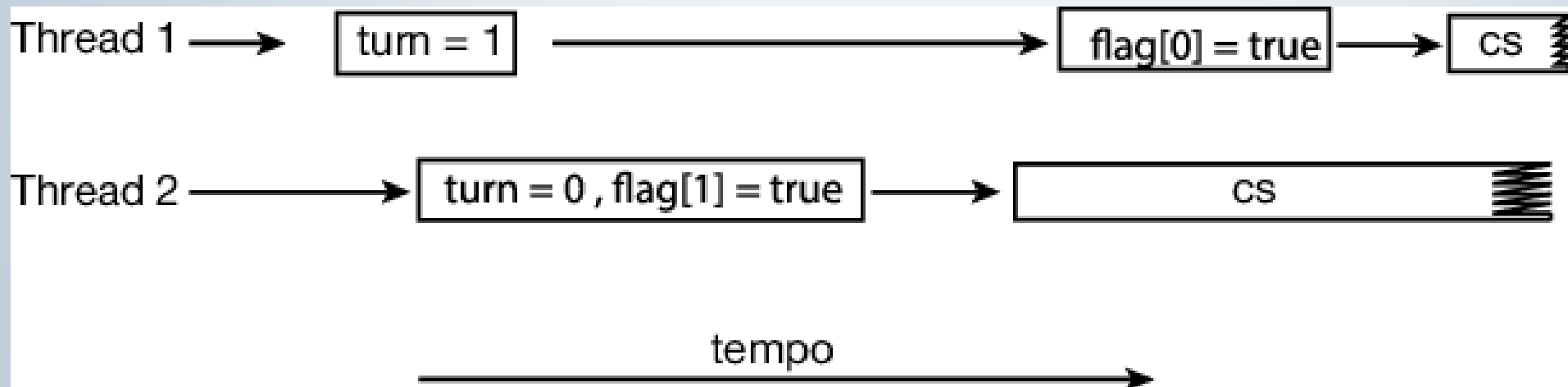
Soluzione Peterson Vs riordino

Il comportamento previsto è che Thread 1 restituisca il valore di x pari a 100.

- ▶ Ma non esistono dipendenze tra le variabili flag e x , è quindi possibile che un processore riordini le istruzioni di Thread 2 in modo che il flag sia impostato a true prima dell'assegnazione. Con conseguente restituzione da parte di Thread 1 dell valore 0.

Soluzione Peterson Vs riordino

Si consideri il problema del riordino applicato alla soluzione di Peterson, cosa accadrebbe se gli assegnamenti che compaiono nella sezione d'ingresso venissero riordinati? Sarebbe possibile avere entrambi i processi attivi nelle rispettive sezioni critiche.



Soluzione Hardware

Come appena constatato le soluzioni basate sul software, come quella di Peterson non garantiscono il funzionamento sulle moderne architetture elaborative. È necessario implementare istruzioni hardware che possano fornire supporto per la soluzione del problema:

1. Barriere di memoria
2. Istruzioni hardware
3. Variabili atomiche

Barriere di memoria

Il modello di memoria rappresenta il modo in cui l'architettura di un computer determina quali garanzie vengono fornite ad un programma applicativo, esistono due tipi di modelli:

- ▶ **Fortemente ordinato:** in cui una modifica alla memoria su un processore è immediatamente visibile a tutti gli altri processori
- ▶ **Debolmente ordinato:** in cui le modifiche alla memoria su un processore potrebbero non essere immediatamente visibili agli altri processori

Barriere di memoria

Il concetto è quello di inserire istruzioni che prendono il nome di **memory barrier** durante la quale il sistema garantisce che tutte le istruzioni di **store** e **load** siano completate prima di eseguire qualunque altra istruzione.

Thread 1 esegue :

```
while(!flag){  
    memory_barrier();  
    print x;  
}
```

Thread 2 esegue :

```
x = 100;  
memory_barrier();  
flag = true;
```


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**.

- ▶ Istruzione : `test_and_set()`
- ▶ Istruzione : `compare_and_swap()`

Istruzione – test_and_set()

La caratteristica fondamentale è l'esecuzione atomica, quindi l'esecuzione concorrente su 2 unità di elaborazione diverse prevedere la realizzazione di una sequenza.

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

Istruzione – test_and_set()

Se si dispone di questa istruzione, la mutua esclusione può essere preservata dichiarando una variabile globale «**LOCK**» inizializzata a false.

```
do {  
    while (test_and_set(&lock));  
        ; /*non fa niente*/  
  
    /*sezione critica*/  
  
    lock = false;  
  
    /*sezione non critica*/  
} while (true);
```

Istruzione – compare_and_swap()

Basata sullo stesso fondamento dell'istruzione precedente, opera su parole atomiche, ma utilizza un meccanismo diverso che si basa sullo scambio del contenuto delle parole.

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

Istruzione – compare_and_swap()

Se si dispone di questa istruzione, la mutua esclusione può essere preservata dichiarando una variabile globale «**LOCK**» inizializzata a 0.

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

Istruzione – compare_and_swap()

L'algoritmo presentato nella slide precedente soddisfa il requisito della mutua esclusione ma non preserva l'attesa limitata. Un implementazione completa può essere quella della figura accanto.

```
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 */
}
```

Variabili Atomiche

L'istruzione `compare_and_swap()` generalmente non viene direttamente utilizzata ma fornisce una base per la costruzione di altri strumenti.

Uno di questi strumenti sono le variabili atomiche, che forniscono operazioni inscindibili su tipi di dati di base come interi e booleani.

Soluzione software – 2

Le soluzioni Hardware proposte per la soluzione al problema della sezione critica sono complicate e generalmente inaccessibili ai programmatori di applicazioni. L'alternativa è rappresentata dall'implementazione nei sistemi operativi di strumenti software quali:

1. Lock Mutex
2. Semafori
3. Monitor

Lock Mutex

- ▶ Uno degli strumenti più semplici implementati all'interno di un sistema operativo prende il nome di Lock Mutex.
- ▶ Concetto alla base di Lock Mutex:
 - ▶ Un processo deve acquisire il lock prima di entrare in una sezione critica e rilasciarlo quando esce dalla sezione critica.
 - ▶ Funzione atomica - `acquire()` per acquisire il lock
 - ▶ Funzione atomica - `release()` per il rilascio del lock

Lock Mutex

```
while (true) {
```

acquisisci lock

sezione critica

rilascia lock

sezione non critica

```
}
```

Lock Mutex – acquire()

- ▶ Un lock mutex sfrutta una variabile booleana «available» che ha la funzione di indicare la disponibilità del lock.
- ▶ Possibile implementazione della funzione di acquisizione:

```
acquire() {  
    while (!available)  
        ; /* attesa attiva */  
    available = false;  
}
```

Lock Mutex – release()

- ▶ Un lock mutex sfrutta una variabile booleana «available» che ha la funzione di indicare la disponibilità del lock.
- ▶ Possibile implementazione della funzione di rilascio:

```
release() {  
    available = true;  
}
```

Lock Mutex – Vantaggi e Svantaggi

- ▶ Vantaggi - Spinlock:
 - ▶ Non è necessario alcun cambio di contesto.
- ▶ Svantaggi - Busy Waiting:
 - ▶ Mentre un processo si trova nella sua sezione critica ogni altro processo che cerchi di entrare nella rispettiva sezione critica è costretto a effettuare continuamente chiamate alla funzione `acquire()`.

Semafori

- ▶ Soluzione **robusta** per la soluzione del problema della sezione critica.
- ▶ Concetto alla base dei Semafori:
 - ▶ Un Semaforo «S» è una variabile intera cui si può accedere solo tramite due operazioni predefinite.
 - ▶ Funzione atomica - wait()
 - ▶ Funzione atomica - signal()

Semafori – wait()

- Possibile implementazione della funzione wait():

```
wait(S){  
    while(S <= 0)  
        ;//attesa attiva  
    S - -  
}
```

Semafori – Signal()

- Possibile implementazione della funzione signal():

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

Semafori

- ▶ **Semafori contatore** : trovano applicazione nel controllo dell'accesso a una data risorsa presente in un numero finito di esemplari.
- ▶ **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.

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

Semafori – esempio P1 e P2

Si considerino due processi in esecuzione concorrente:

- ▶ P1 – istruzione S1
- ▶ P2 – istruzione S2

Si vuole eseguire S2 solo dopo che il processo P1 abbia terminato S1.

La sincronizzazione di questi 2 processi è facilmente realizzabile sfruttando un semaforo comune «synch» inizializzato a 0.

Semafori – esempio P1 e P2

Vengono aggiunte ai processi rispettivamente:

- ▶ P1 – istruzioni S1 e `signal(synch)`
- ▶ P2 – istruzioni S2 e `wait(synch)`

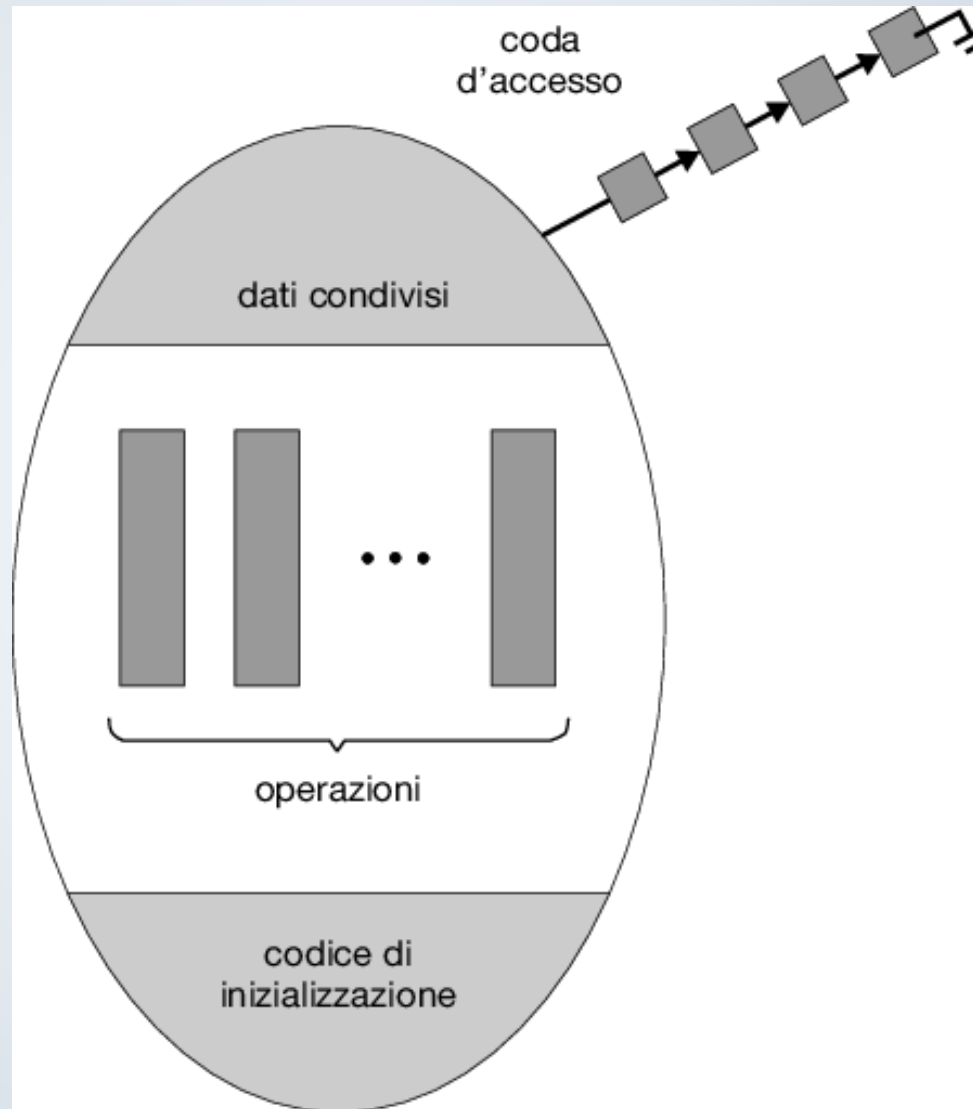
Poiché `synch` è inizializzato a 0 p2 esegue la sua istruzione S2 solo dopo che P1 ha eseguito la sua istruzione `signal(synch)`.

Semafori – Implementazione

- ▶ Risolvere il problema dell'attesa attiva rende questa soluzione più robusta rispetto ai lock mutex.
- ▶ Concetto di base:
 - ▶ Un processo invoca funzione `wait()` e trova che il valore del semaforo non è positivo allora deve attendere, ma anziché restare in attesa attiva, sospende se stesso.
 - ▶ Un processo sospeso sarà riavviato in seguito all'esecuzione di un operazione `signal()` da parte di un altro processo.

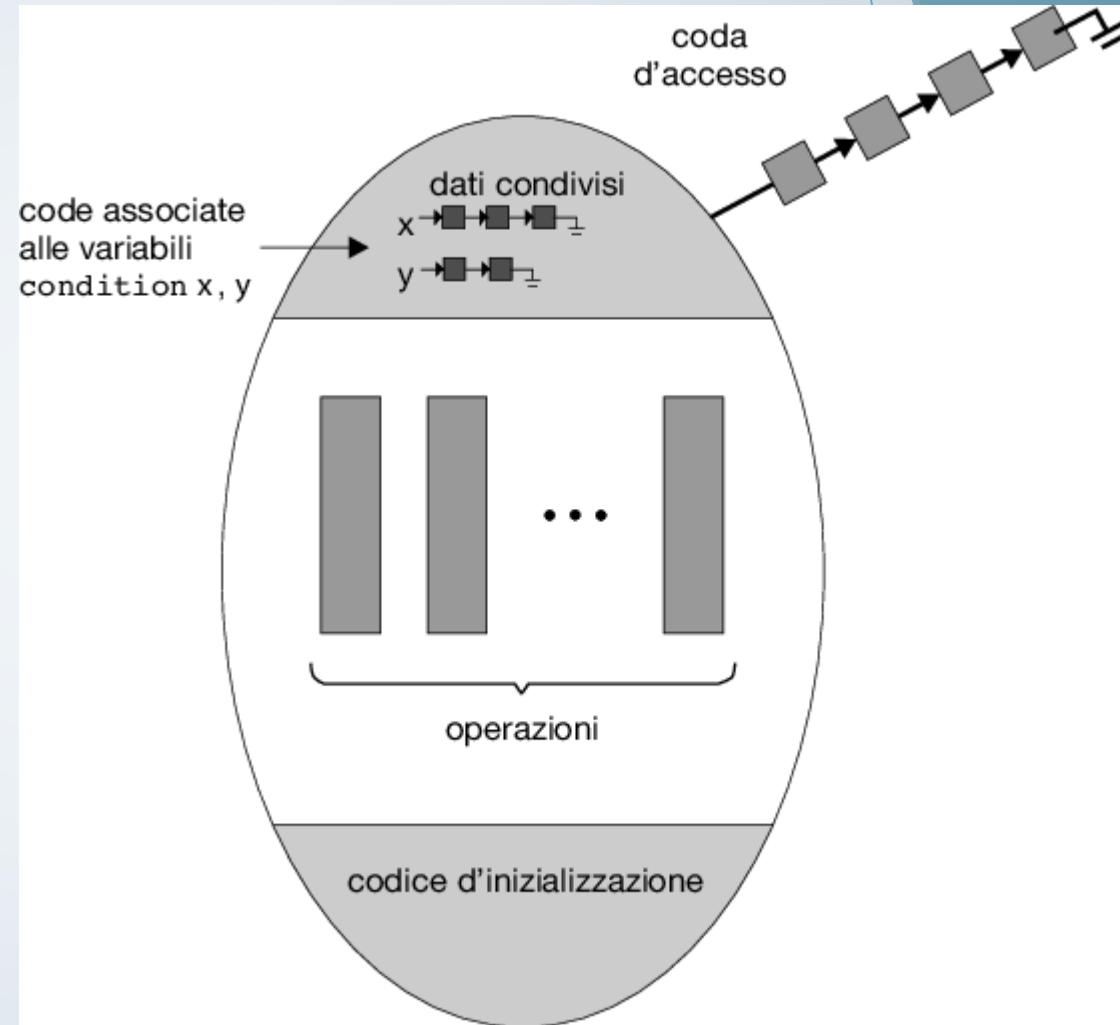
Monitor

- 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.



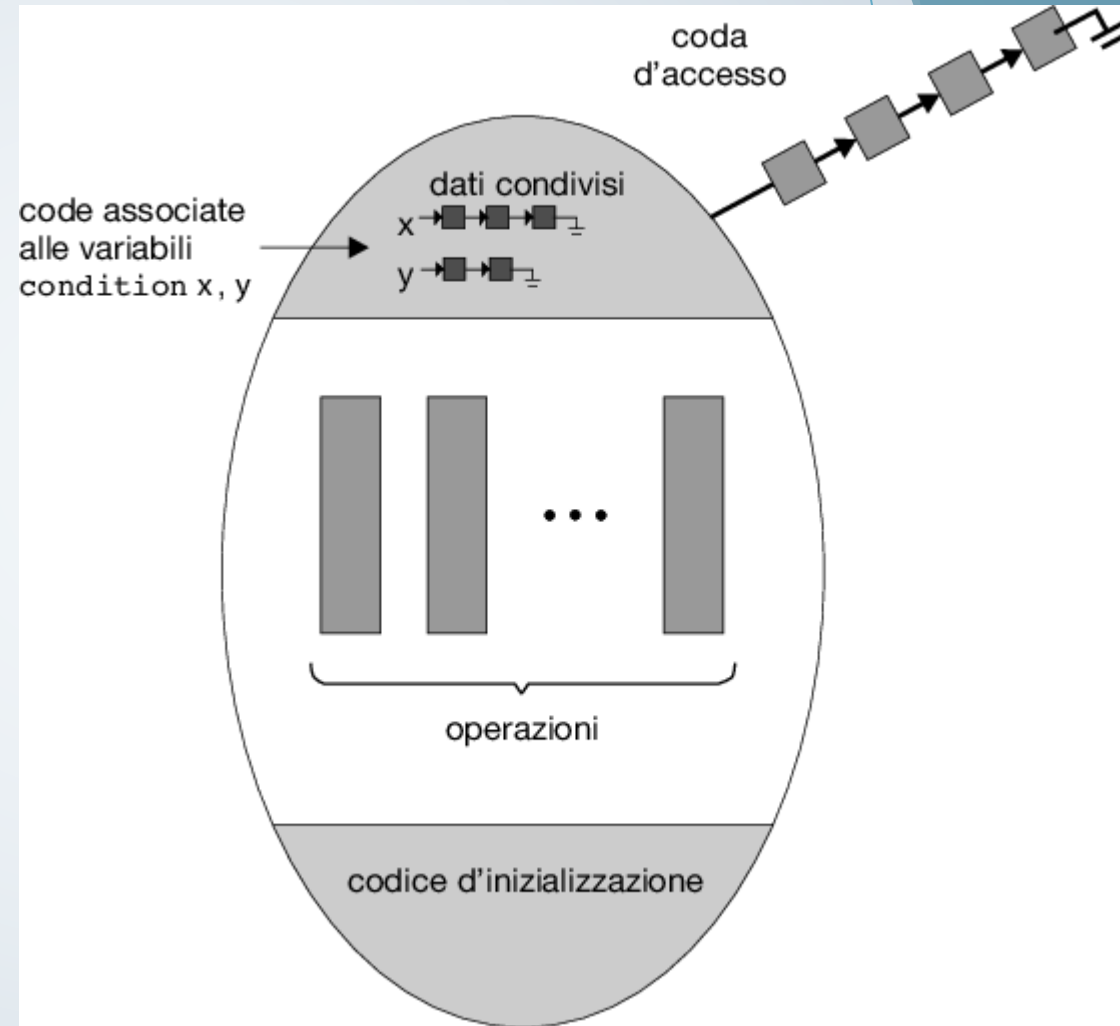
Monitor

- ▶ Un programmatore che necessita di implementare un particolare schema di sincronizzazione può definire una o più variabili di tipo condition:
- ▶ Condition x, y;
- ▶ Le uniche operazioni eseguibili su queste variabili sono wait() e signal()



Monitor

- L'operazione `signal()` risveglia un processo sospeso, ma se non esistono processi sospesi allora non produce nessun effetto. Ovvero il valore della variabile resta immutato, come se l'operazione `signal()` non fosse stata mai eseguita.



Liveness

- ▶ Riferimento a un insieme di proprietà che un sistema deve soddisfare per garantire che i processi facciano progressi durante il loro ciclo di vita.
 - ▶ Stallo (Deadlock)
 - ▶ Inversione di priorità

Deadlock

- ▶ Un insieme di processi è in stallo se ciascun processo dell'insieme che attende un evento che può essere causato solo da un altro processo dell'insieme.
- ▶ Un'altra questione connessa alle situazioni di stallo è quella dell'attesa indefinita (Starvation), un esempio di facile comprensione è una coda di tipo LIFO.

Inversione di priorità

- ▶ Nello scheduling dei processi si possono incontrare difficoltà ogniqualvolta un processo a priorità più alta abbia bisogno di leggere o modificare dati a livello kernel utilizzati da un processo, o da una catena di processi, a priorità più bassa. Visto che i dati a livello kernel sono tipicamente protetti da un lock, il processo a priorità maggiore dovrà attendere finché il processo a priorità minore non avrà finito di utilizzare le risorse.

Inversione di priorità

- ▶ Una buona lettura per questo problema: Mars Pathfinder

http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html

Valutazione delle soluzioni

- ▶ Le linee guida a seguire ci permettono di identificare delle regole generali per **distinguere le prestazioni** della sincronizzazione basata su (compare_and_swap() - CAS) e della sincronizzazione tradizionale (lock mutex, semafori).
- ▶ L'indicatore delle prestazioni prende il nome di : **Livello di contesa**

Valutazione delle soluzioni

- ▶ **Nessuna contesa:** Sebbene entrambe le opzioni siano 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.

Riferimenti - fonti

1. © Pearson Italia S.p.A. – Silberschatz, Galvin, Gagne, *Sistemi operativi*