

# Sistemi Operativi

**C.d.L. in Informatica (laurea triennale)**

Anno Accademico 2021-2022

Canale A-L

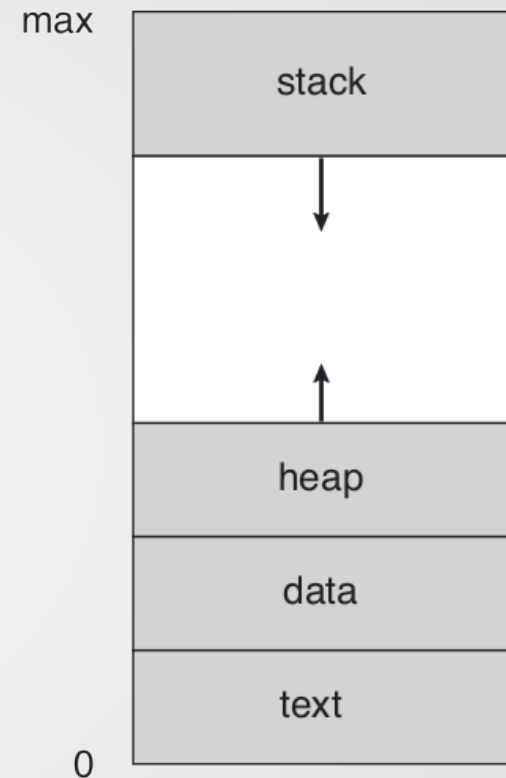
Dipartimento di Matematica e Informatica – Catania

**Processi, Thread, IPC e Scheduling**

Prof. Mario Di Raimondo

# Processo

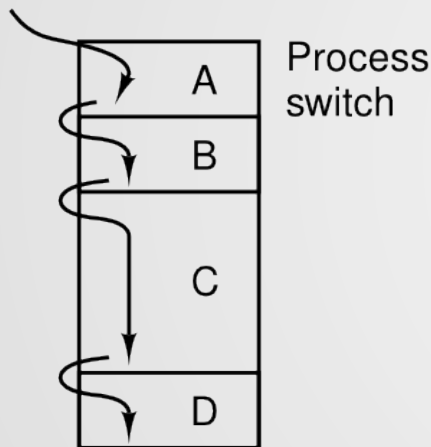
- **Definizione:** una istanza di esecuzione di un programma.
- Si associano ad esso:
  - **spazio degli indirizzi:**
    - codice;
    - dati;
    - stack;
  - copia dei **registri** della CPU;
  - **file aperti;**
  - **allarmi** pendenti;
  - processi imparentati.
- **Tabella dei processi** con un **Process Control Block (PCB)** per ogni processo.



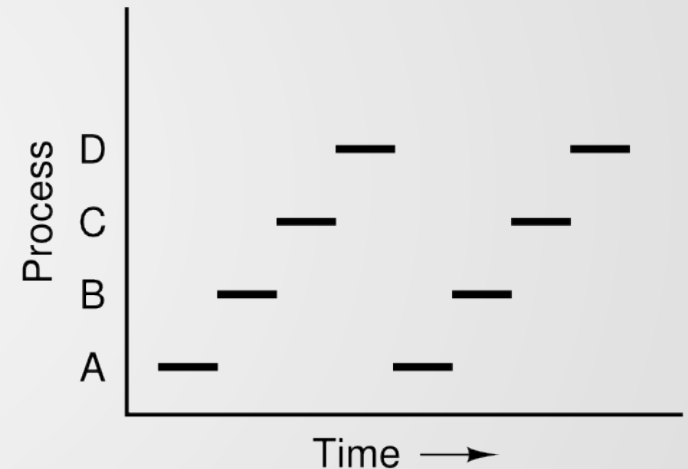
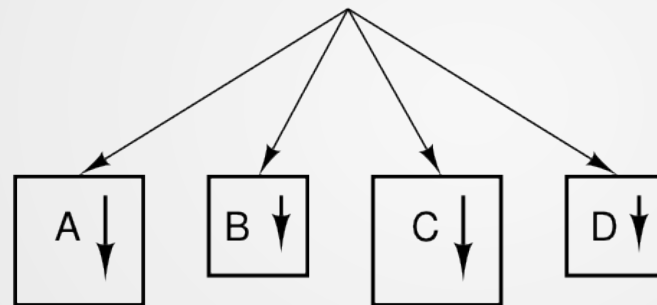
# Modello dei processi

- **Multiprogrammazione e pseudo-parallelismo.**
- E' più semplice ragionare pensando a **processi sequenziali** con una **CPU virtuale** dedicata.

One program counter



Four program counters

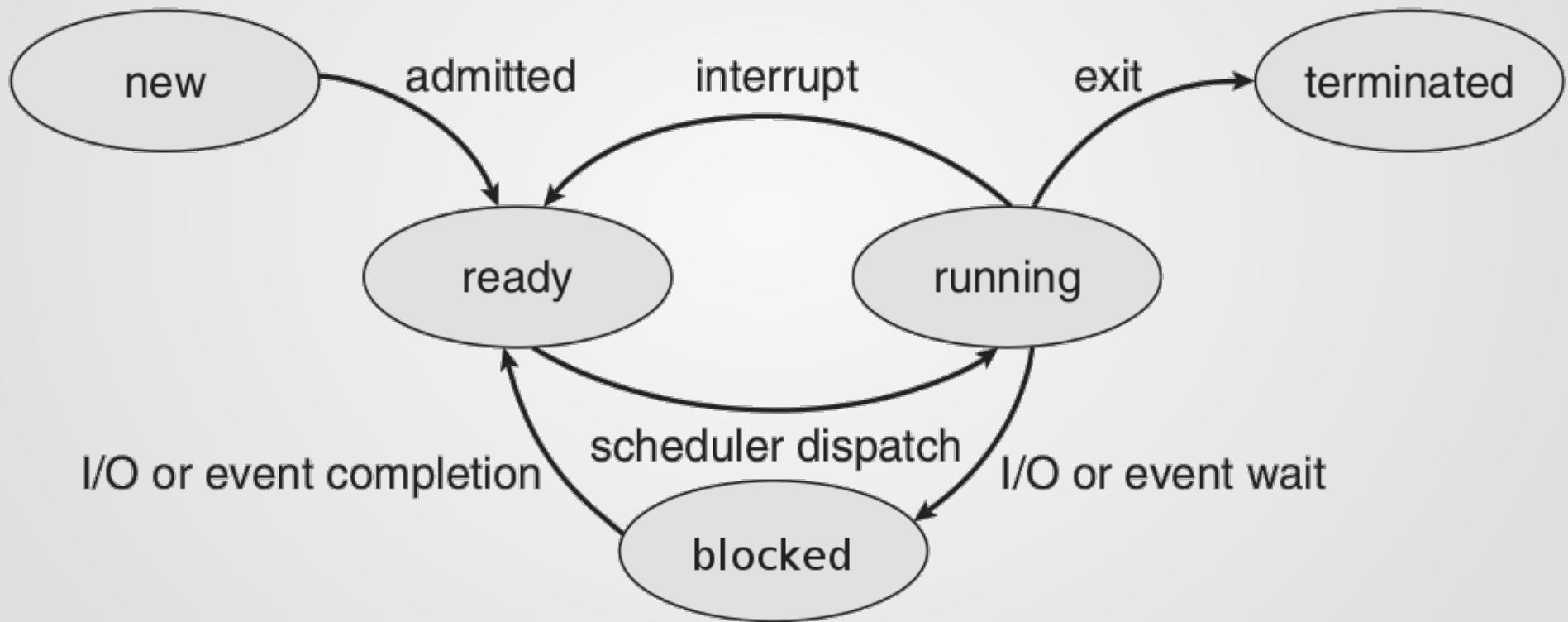


# Creazione e terminazione dei processi

- **Creazione** di un processo:
  - in fase di inizializzazione del sistema;
  - da parte di un altro processo (padre) o per un'azione dell'utente;
  - metodologie:
    - **sdoppiamento del padre**: fork e exec (UNIX);
    - **nuovo processo per nuovo programma**: CreateProcess (Win32).
- **Terminazione**:
  - uscita normale (volontario): exit (UNIX), ExitProcess (Win32);
  - uscita su errore (volontario);
  - errore critico (involontario): alcuni sono gestibili, altri no;
  - terminato da un altro processo (involontario): kill (UNIX), TerminateProcess (Win32).

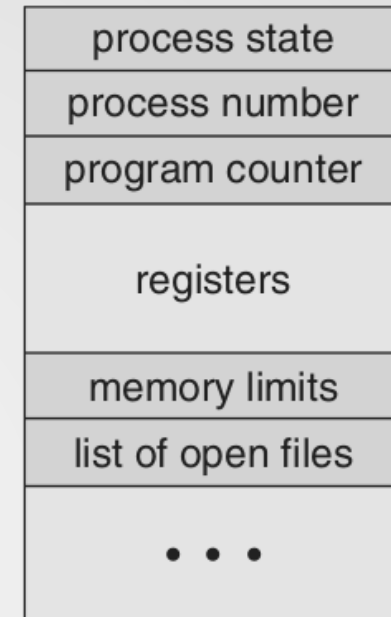
# Stato di un processo

- 3 stati principali (+ 2 aggiionali);
- transizioni.

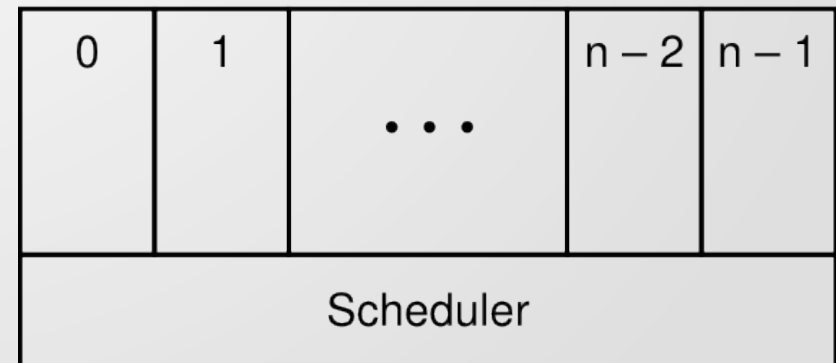


# Tabella dei processi

- **Tabella dei processi;**
- **Process Control Block (PCB);**
- **Scheduler;**
- **Gestione degli interrupt per il passaggio di processo:**
  - salvataggio nello stack del PC e del PSW nello stack attuale;
  - caricamento dal vettore degli interrupt l'indirizzo della procedura associata;
  - salvataggio registri e impostazione di un nuovo stack;
  - esecuzione procedura di servizio per l'interrupt;
  - interrogazione dello scheduler per sapere con quale processo proseguire;
  - ripristino dal PCB dello stato di tale processo (registri, mappa memoria);
  - ripresa nel processo corrente.

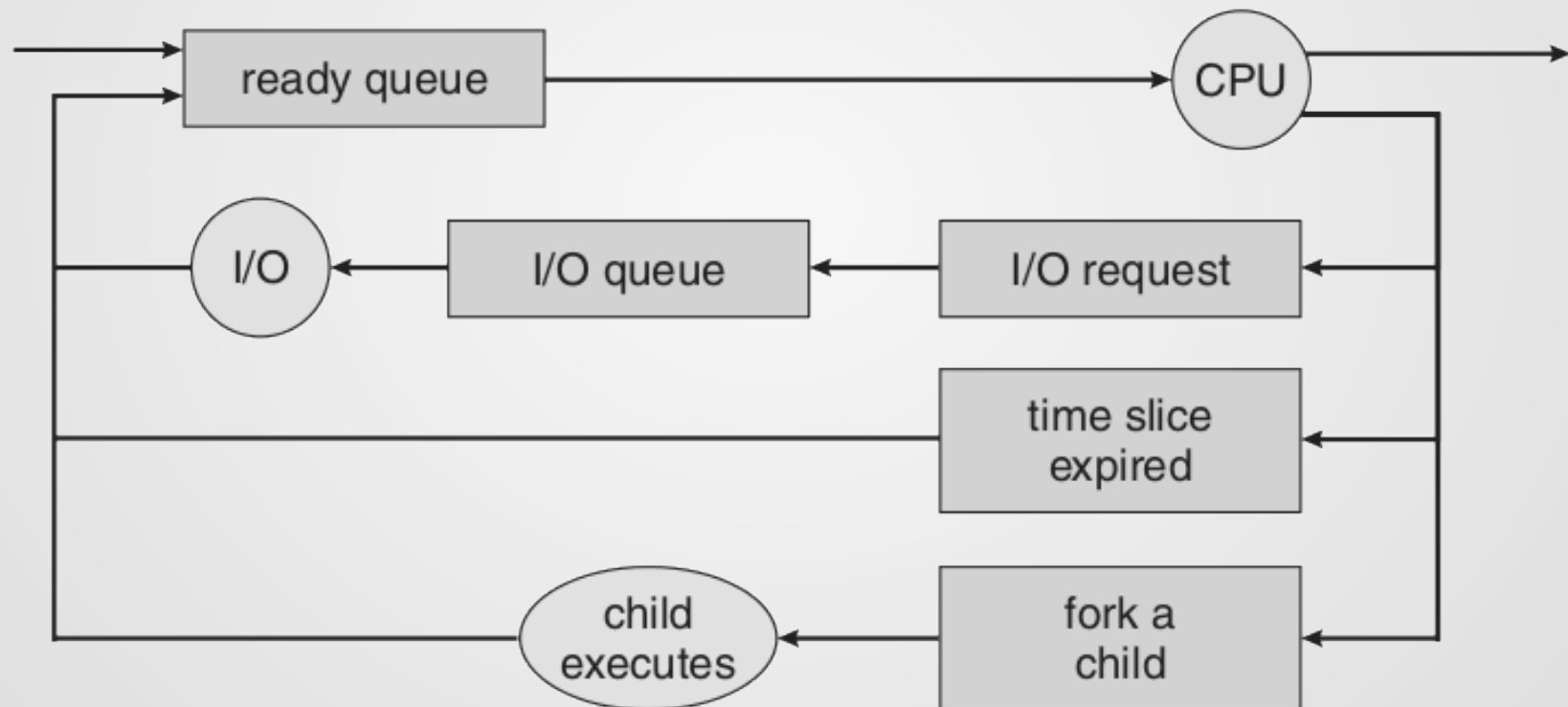


Processes



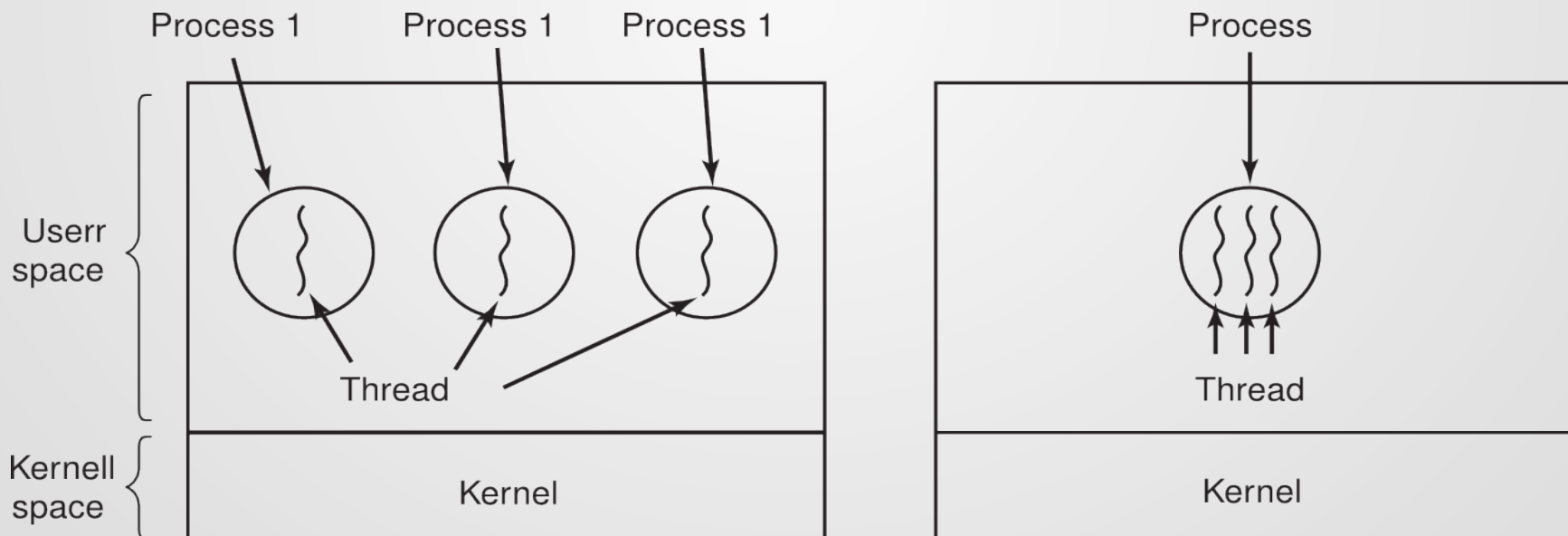
# Code e accodamento

- **Coda dei processi pronti e code dei dispositivi;**
  - strutture collegate sui PCB;
- **Diagramma di accodamento:**



# Thread

- Modello dei processi: entità indipendenti che **raggruppano risorse** e con un **flusso di esecuzione**;
- può essere utile far condividere a più flussi di esecuzione lo stesso spazio di indirizzi: **thread**;
- quando può essere utile?
  - esempi: web-browser, videoscrittura, web-server, ...

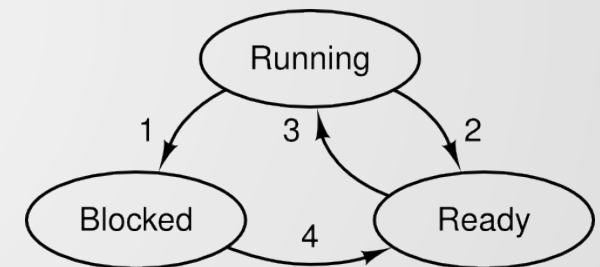
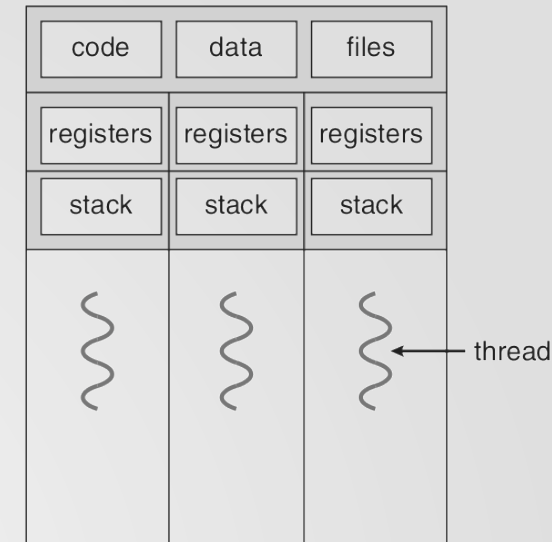
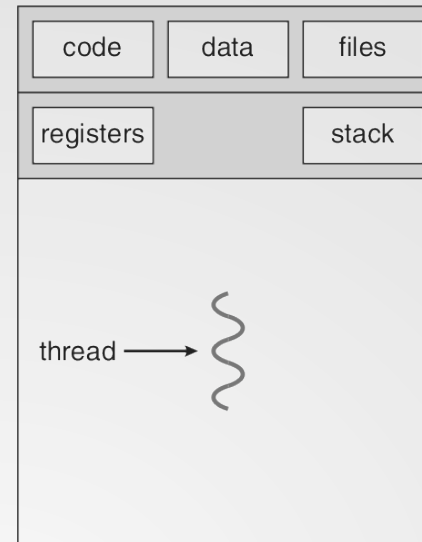




# Thread

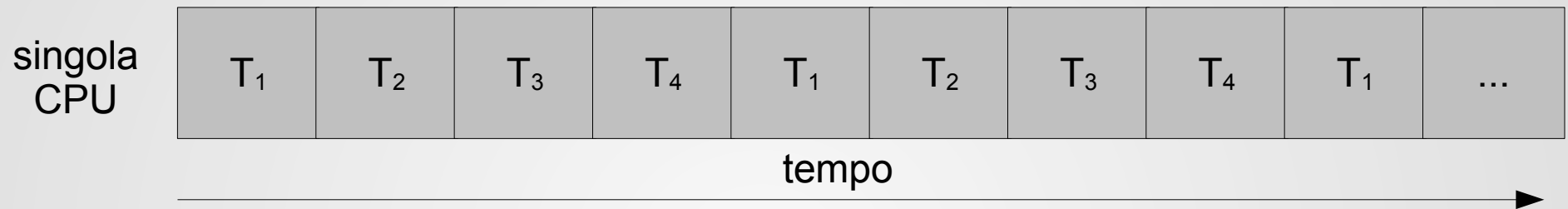
- Un thread è **caratterizzato** da:
  - PC, registri, stack, stato;
  - condivide tutto il resto;
    - non protezione di memoria.
- **scheduling** dei thread;
- **cambio di contesto** più veloce;
- **cambiamenti di stato** dei thread;
- **operazioni** tipiche sui thread:

- `thread_create`: un thread ne crea un altro;
- `thread_exit`: il thread chiamante termina;
- `thread_join`: un thread si sincronizza con la fine di un altro thread;
- `thread_yield`: il thread chiamante rilascia volontariamente la CPU.

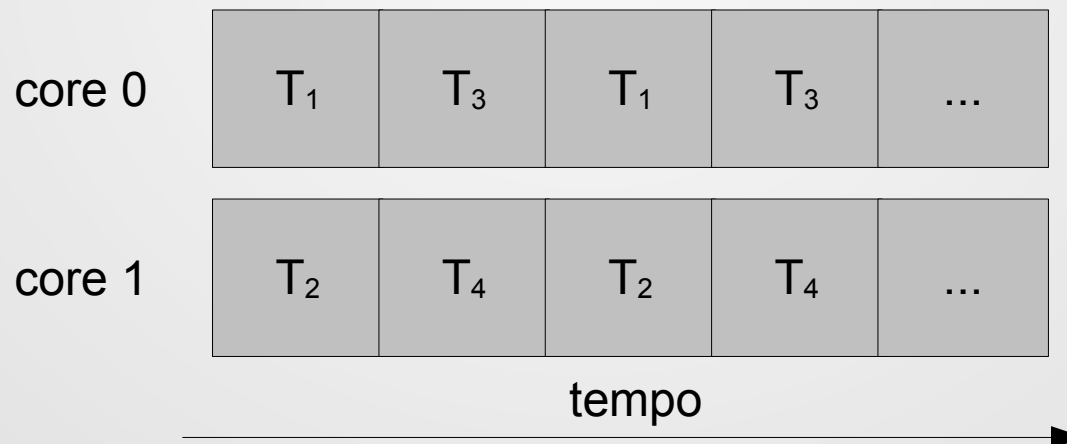


# Programmazione multicore

- I thread permettono una **migliore scalabilità** con core con hypertreading e soprattutto con sistemi multicore;
- con un sistema single-core abbiamo una esecuzione interleaved;



- su un sistema multi-core abbiamo parallelismo puro.

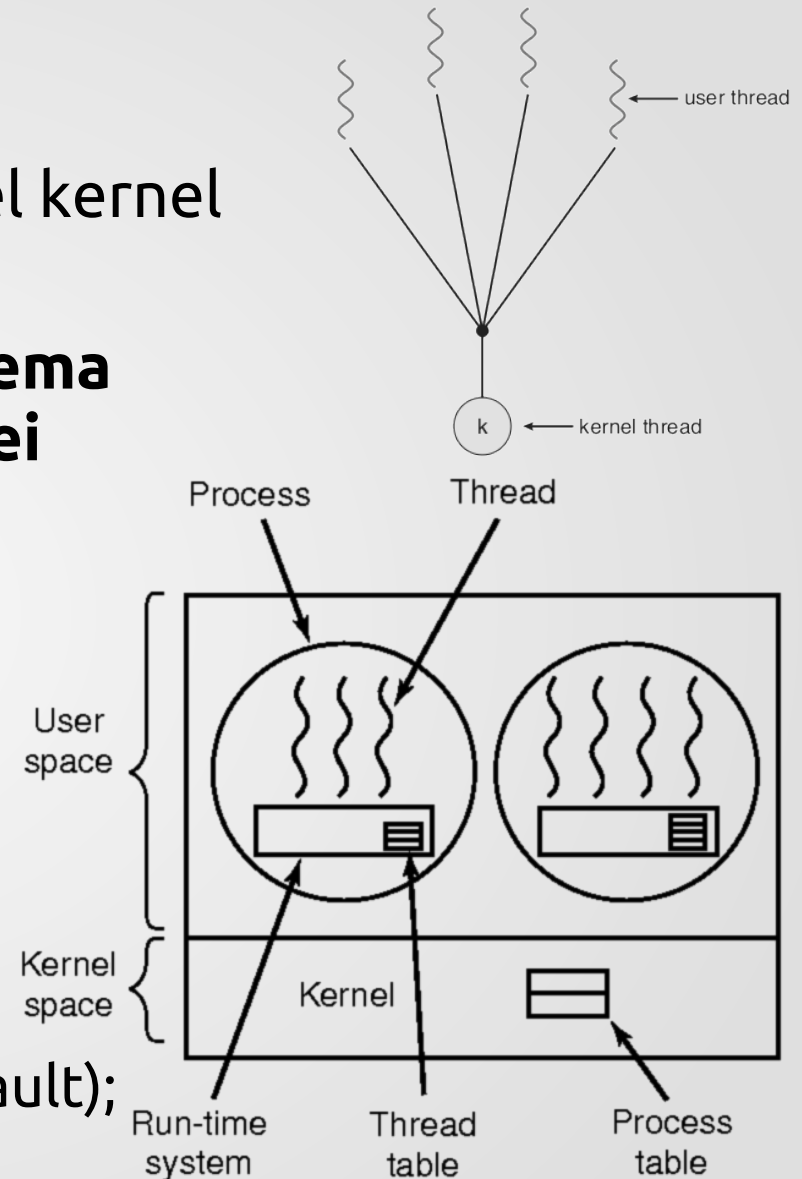


# Programmazione multicore

- Progettare programmi che sfruttino le moderne architetture multicore non è banale;
- principi base:
  - **separazione dei task;**
  - **bilanciamento;**
  - **suddivisione dei dati;**
  - **dipendenze dei dati;**
  - **test e debugging.**

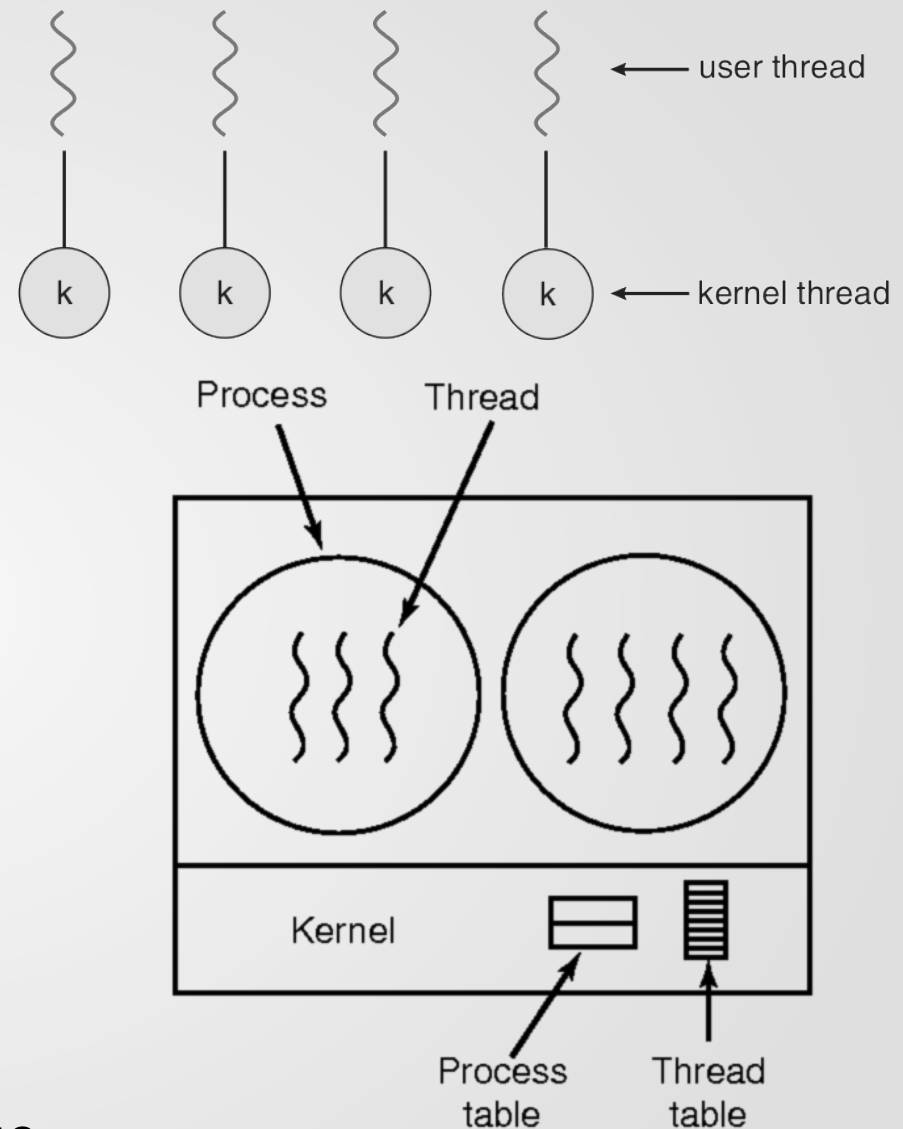
# Thread a livello utente

- Detto anche "**modello 1-a-molti**";
- utile se non c'è supporto da parte del kernel ai thread;
- una **libreria** che implementa un **sistema run-time** che gestisce una **tabella dei thread** del processo.
- **Pro:**
  - il dispatching non richiede trap nel kernel;
  - scheduling personalizzato;
- **Contro:**
  - chiamate bloccanti (select, page-fault);
  - possibilità di non rilascio della CPU.



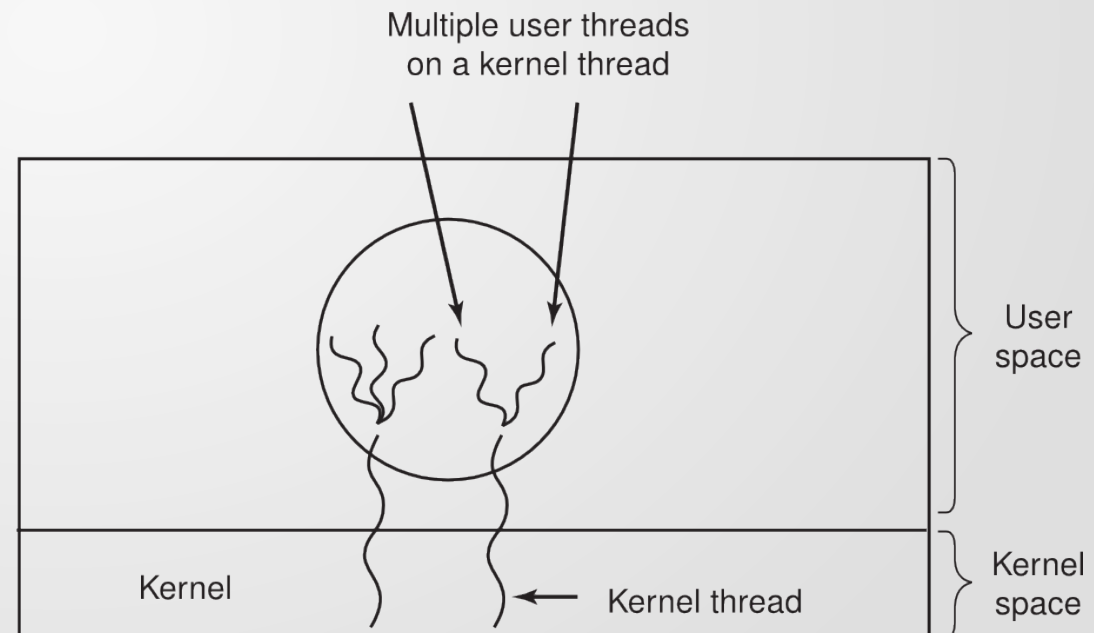
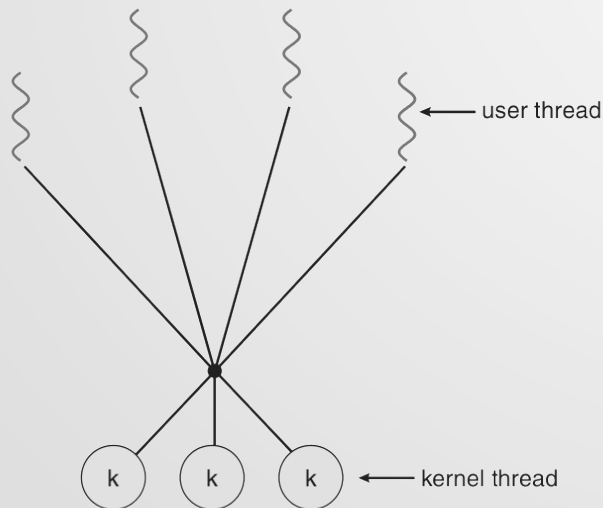
# Thread a livello kernel

- Detto anche "**modello 1-a-1**";
- richiede il supporto specifico dal kernel (praticamente tutti i moderni SO);
- **unica tabella dei thread** del kernel;
- **Pro:**
  - un thread su chiamata bloccante non intralcia gli altri;
- **Contro:**
  - cambio di contesto più lento (richiede trap);
  - creazione e distruzione più costose (numero di thread kernel tipicamente limitato, possibile riciclo).



# Modello ibrido

- Detto anche "**multi-a-molti**";
- prende il meglio degli altri due;
- prevede un certo numero di **thread del kernel**;
- ognuno di essi viene assegnato ad un certo numero di **thread utente** (eventualmente uno);
- **assegnazione** decisa dal programmatore.



# I thread nei nostri sistemi operativi

- Quasi tutti i sistemi operativi supportano i **thread a livello kernel**;
  - Windows, Linux, Solaris, Mac OS,...
- Supporto ai **thread utente** attraverso apposite librerie:
  - *green threads* su Solaris;
  - *GNU portable thread* su UNIX;
  - *fiber* su Win32.
- **Librerie di accesso ai thread** (a prescindere dal modello):
  - *Pthreads* di POSIX (Solaris, Linux, Mac OS, anche Windows);
    - una specifica da implementare sui vari sistemi;
  - threads Win32;
  - thread in Java;
    - wrapper sulle API sottostanti.

# Comunicazione fra processi

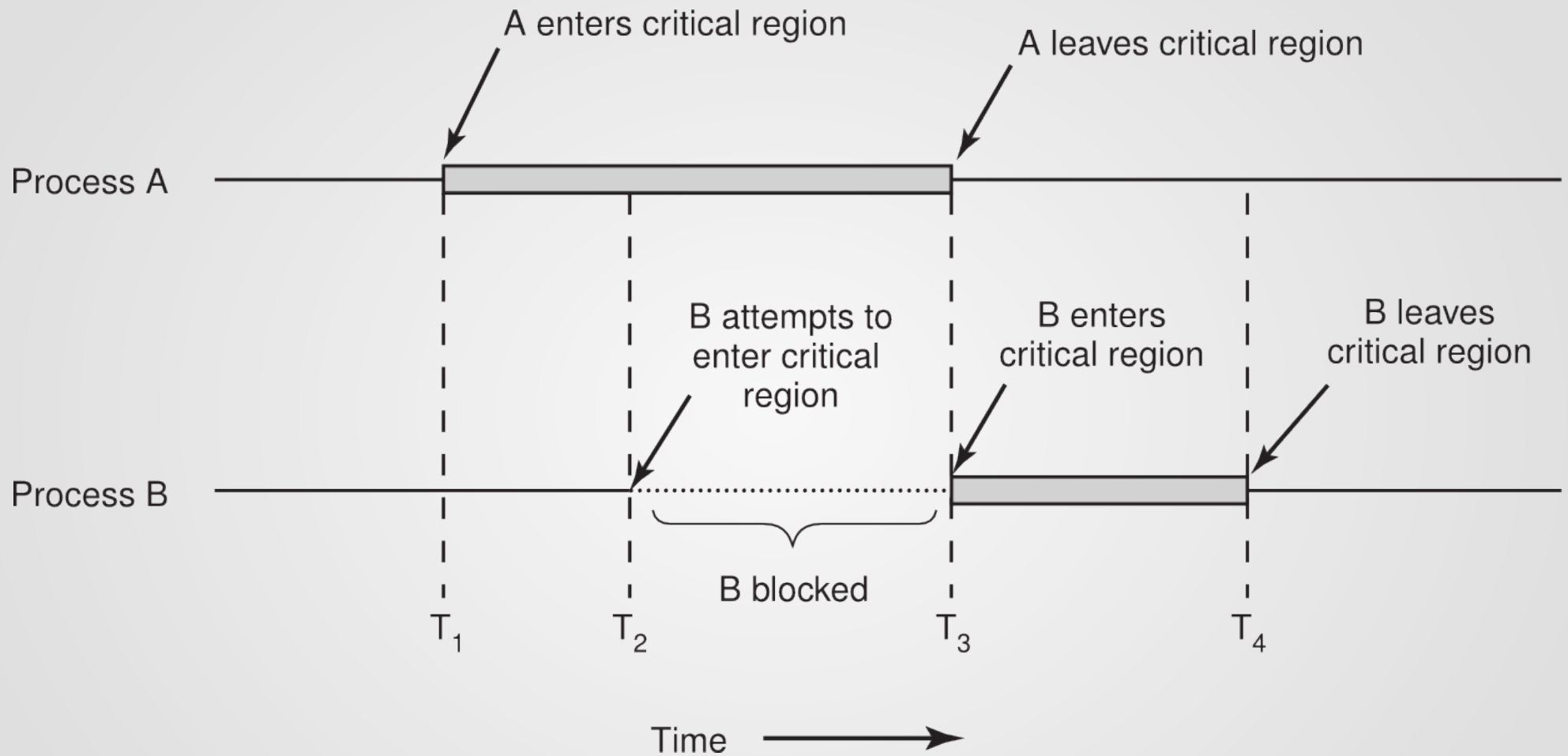
- Spesso i processi hanno bisogno di **cooperare**:
  - collegamento I/O tra processi (**pipe**);
  - **InterProcess Communication** (IPC);
  - possibili **problematiche**:
    - come scambiarsi i dati;
    - accavallamento delle operazioni su dati comuni;
    - coordinamento tra le operazioni (o sincronizzazione).
- **Corse critiche** (race conditions);
  - esempio: versamenti su conto-corrente;
  - corse critiche nel codice del **kernel**;
  - soluzione: **mutua esclusione** nell'accesso ai dati condivisi.



# Sezioni critiche

- Astrazione del problema: **sezioni critiche e sezioni non critiche.**
- **Quattro condizioni** per avere una buona soluzione:
  1. mutua esclusione nell'accesso alle rispettive sezioni critiche;
  2. nessuna assunzione sulla velocità di esecuzione o sul numero di CPU;
  3. nessun processo fuori dalla propria sezione critica può bloccare un altro processo;
  4. nessun processo dovrebbe restare all'infinito in attesa di entrare nella propria sezione critica.

# Sezioni critiche



# Come realizzare la mutua esclusione

- Disabilitare gli interrupt.
- Variabili di lock.
- Alternanza stretta:

```
int N=2
int turn

function enter_region(int process)
    while (turn != process) do
        nothing

function leave_region(int process)
    turn = 1 - process
```

- può essere facilmente generalizzato al caso N;
- fa **busy waiting** (si parla di **spin lock**);
- implica **rigidi turni** tra le parti (viola condizione 3).

# Soluzione di Peterson

```
int N=2
int turn
int interested[N]

function enter_region(int process)
    other = 1 - process
    interested[process] = true
    turn = process
    while (interested[other] = true and turn = process) do
        nothing

function leave_region(int process)
    interested[process] = false
```

- ancora busy waiting;
- può essere generalizzato al caso N;
- può avere problemi sui moderni multi-processori a causa del riordino degli accessi alla memoria centrale.

# Istruzioni TSL e XCHG

- Molte architetture (soprattutto multi-processore) offrono specifiche istruzioni:
  - **TSL (Test and Set Lock);**
    - uso: **TSL registro, lock**
    - operazione atomica e blocca il bus di memoria;

```
enter_region:  
    TSL REGISTER, LOCK  
    CMP REGISTER, #0  
    JNE enter_region  
    RET
```

```
leave_region:  
    MOVE LOCK, #0  
    RET
```

- **XCHG (eXCHanGe);**
    - disponibile in tutte le CPU Intel X86;
- ancora busy waiting.

# Sleep e wakeup

- Tutte le soluzioni viste fino ad ora fanno **spin lock**;
  - **problema dell'inversione di priorità.**
- **Soluzione:** dare la possibilità al processo di bloccarsi in modo passivo (rimozione dai processi pronti);
  - primitive: **sleep** e **wakeup**.
- **Problema del produttore-consumatore (buffer limitato – N):**
  - variabile condivisa count inizialmente posta a 0;

```
function producer()
  while (true) do
    item = produce_item()
    if (count = N) sleep()
    insert_item(item)
    count = count + 1
    if (count = 1)
      wakeup(consumer)
```

```
function consumer()
  while (true) do
    if (count = 0) sleep()
    item = remove_item()
    count = count - 1
    if (count = N - 1)
      wakeup(producer)
    consume_item(item)
```

- questa soluzione **non funziona bene**: usiamo un **bit di attesa wakeup**.

# Semafori

- Generalizziamo il concetto di sleep e wakeup – **semaforo**:
  - variabile intera condivisa S;
  - operazioni: **down** e **up** (dette anche **wait** e **signal**);
  - **operazioni atomiche**;
    - disabilitazione interrupt o spin lock TSL/XCHG;
  - tipicamente implementato **senza busy waiting** con una **lista di processi bloccati**.
- Due tipi di utilizzo:
  - **semaforo mutex** (mutua esclusione);
  - **conteggio risorse** (sincronizzazione).

# Produttore-consumatore con i semafori

```
int N=100  
semaphore mutex = 1  
semaphore empty = N  
semaphore full = 0
```

```
function producer()  
  while (true) do  
    item = produce_item()  
    down(empty)  
    down(mutex)  
    insert_item(item)  
    up(mutex)  
    up(full)
```

```
function consumer()  
  while (true) do  
    down(full)  
    down(mutex)  
    item = remove_item()  
    up(mutex)  
    up(empty)  
    consume_item(item)
```

- L'ordine delle operazioni sui semafori è fondamentale...



# Mutex e thread utente

- Tra i **thread utente** che fanno riferimento ad un unico processo (**modello 1-a-molti**) si possono implementare efficientemente i **mutex** facendo uso di TSL (o XCHG):

```
mutex_lock:
    TSL REGISTER,MUTEX
    CMP REGISTER,#0
    JZE ok
    CALL thread_yield
    JMP mutex_lock
ok:RET
```

```
mutex_unlock:
    MOVE MUTEX,#0
    RET
```

- **simili** a enter\_region/leave\_region ma:
  - senza **spin lock**;
  - il busy waiting sarebbe problematico con i thread utente;
- molto efficienti.

# Futex

- Osservazione: i mutex in user-space sono molto efficienti ma lo spin lock può essere lungo!  
→ **futex** = fast user space mutex (Linux)
- due componenti:
  - **servizio kernel**
    - ♦ coda di thread bloccati
  - **libreria utente**
    - ♦ variabile di **lock**
    - ♦ contesa in modalità utente (tipo con TSL/XCHG)
    - ♦ richiamo kernel solo in caso di bloccaggio

# I Monitor

- Costrutto ad alto-livello disponibile su alcuni linguaggi;
- un **tipo astratto di dato** (variabili + procedure) con:
  - garanzia di **mutua esclusione**;
    - ♦ esiste una coda di attesa interna;
  - vincolo di accesso ai dati (interni ed esterni).
- Meccanismo di sincronizzazione: **variabili condizione**
  - operazioni **wait** e **signal**;
    - ♦ esiste una coda di attesa per ogni variabile;
  - la signal può avere **diverse semantiche**:
    - ♦ monitor **Hoare** (teorico): **signal & wait**;
    - ♦ monitor **Mesa** (Java): **signal & continue**;
    - ♦ compromesso (concurrent Pascal): **signal & return**.
- Vantaggi e svantaggi.

# Produttore-consumatore con i monitor

```
monitor pc_monitor  
  condition full, empty;  
  integer count = 0;
```

```
function insert(item)  
  if count = N then wait(full);  
  insert_item(item);  
  count = count + 1;  
  if count = 1 then signal(empty)
```

```
function remove()  
  if count = 0 then  
    wait(empty);  
  remove = remove_item();  
  count = count - 1;  
  if count = N-1 then signal(full)
```

```
function producer()  
  while (true) do  
    item = produce_item()  
    pc_monitor.insert(item)
```

```
function consumer()  
  while (true) do  
    item = pc_monitor.remove()  
    consume_item(item)
```

# Scambio messaggi tra processi

- Primitive più ad alto livello:
  - `send(destinazione, messaggio)`
  - `receive(sorgente, messaggio)`
    - bloccante per il chiamante (o può restituire un errore);
  - estendibile al caso di più macchine (es., libreria MPI);
  - metodi di indirizzamento: **diretto** o tramite **mailbox**;
  - assumendo realisticamente l'esistenza di un **buffer** per i messaggi:
    - capienza finita  $N$ ;
    - la `send` può essere bloccante;

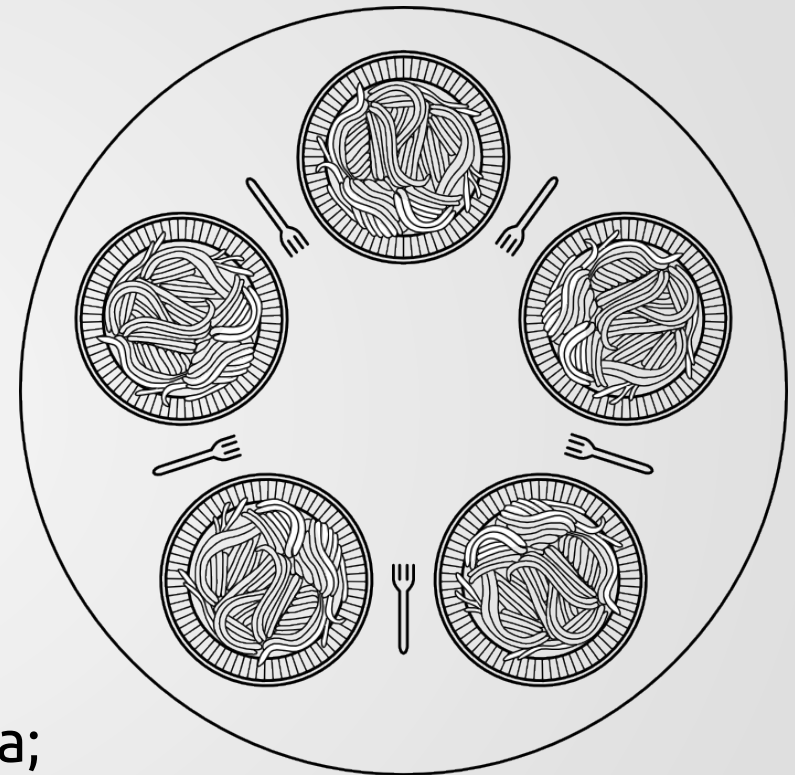
```
function producer()  
  while (true) do  
    item = produce_item()  
    build_msg(m,item)  
    send(consumer, msg)
```

```
function consumer()  
  while (true) do  
    receive(producer, msg)  
    item=extract_msg(msg)  
    consum_item(item)
```

# Problema dei 5 filosofi

- Problema classico che modella l'**accesso esclusivo** ad un **numero limitato di risorse** da parte di **processi in concorrenza**.
- **soluzione 1:**

```
int N=5
function philosopher(int i)
    think()
    take_fork(i)
    take_fork((i+1) mod N)
    eat()
    put_fork(i)
    put_fork((i+1) mod N)
```



- **soluzione 2:** controlla con rilascio, riprova;
- **soluzione 3:** controlla con rilascio e riprova aspettando un tempo random.
- **soluzione 4:** utilizzo di un semaforo mutex.

# Problema dei 5 filosofi: soluzione basata sui semafori

```
int N=5; int THINKING=0
int HUNGRY=1; int EATING=2
int state[N]
semaphore mutex=1
semaphore s[N]={0,...,0}
```

```
function philosopher(int i)
    while (true) do
        think()
        take_forks(i)
        eat()
        put_forks(i)
```

```
function take_forks(int i)
    down(mutex)
    state[i]=HUNGRY
    test(i)
    up(mutex)
    down(s[i])
```

```
function put_forks(int i)
    down(mutex)
    state[i]=THINKING
    test(left(i))
    test(right(i))
    up(mutex)
```

```
function left(int i)  = i-1 mod N
function right(int i) = i+1 mod N
```

```
function test(int i)
    if state[i]=HUNGRY and state[left(i)]!=EATING and state[right(i)]!=EATING
        state[i]=EATING
        up(s[i])
```

# Problema dei 5 filosofi: soluzione basata sui monitor

```
int N=5; int THINKING=0; int HUNGRY=1; int EATING=2
```

```
monitor dp_monitor  
    int state[N]  
    condition self[N]
```

```
    function take_forks(int i)  
        state[i] = HUNGRY  
        test(i)  
        if state[i] != EATING  
            wait(self[i])
```

```
    function put_forks(int i)  
        state[i] = THINKING;  
        test(left(i));  
        test(right(i));
```

```
    function test(int i)  
        if ( state[left(i)] != EATING and state[i] = HUNGRY  
            and state[right(i)] != EATING )  
            state[i] = EATING  
            signal(self[i])
```

```
function philosopher(int i)  
    while (true) do  
        think()  
        dp_monitor.take_forks(i)  
        eat()  
        dp_monitor.put_forks(i)
```



# Problema dei lettori e scrittori: soluzione basata sui semafori

- Problema classico che modella l'accesso ad un data-base;

```
function reader()
  while true do
    down(mutex)
    rc = rc+1
    if (rc = 1) down(db)
    up(mutex)
    read_database()
    down(mutex)
    rc = rc-1
    if (rc = 0) up(db)
    up(mutex)
    use_data_read()
```

```
semaphore mutex = 1
semaphore db = 1
int rc = 0
```

```
function writer()
  while true do
    think_up_data()
    down(db)
    write_database()
    up(db)
```

- **problema:** lo scrittore potrebbe attendere per un tempo indefinito.

# Problema dei lettori e scrittori: soluzione n.1 basata sui monitor

```
monitor rw_monitor
  int rc = 0; boolean busy_on_write = false
  condition read, write

  function start_read()
    if (busy_on_write) wait(read)
    rc = rc+1
    signal(read)

  function end_read()
    rc = rc-1
    if (rc = 0) signal(write)

  function start_write()
    if (rc > 0 OR busy_on_write) wait(write)
    busy_on_write = true

  function end_write()
    busy_on_write = false
    if (in_queue(read))
      signal(read)
    else
      signal(write)
```

```
function reader()
  while true do
    rw_monitor.start_read()
    read_database()
    rw_monitor.end_read()
    use_data_read()
```

```
function writer()
  while true do
    think_up_data()
    rw_monitor.start_write()
    write_database()
    rw_monitor.end_write()
```

# Problema dei lettori e scrittori: soluzione n.2 basata sui monitor

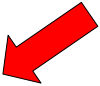
```
monitor rw_monitor
  int rc = 0; boolean busy_on_write = false
  condition read, write

  function start_read()
    if (busy_on_write OR in_queue(write)) wait(read)
    rc = rc+1
    signal(read)

  function end_read()
    rc = rc-1
    if (rc = 0) signal(write)

  function start_write()
    if (rc > 0 OR busy_on_write) wait(write)
    busy_on_write = true

  function end_write()
    busy_on_write = false
    if (in_queue(read))
      signal(read)
    else
      signal(write)
```



```
function reader()
  while true do
    rw_monitor.start_read()
    read_database()
    rw_monitor.end_read()
    use_data_read()
```

```
function writer()
  while true do
    think_up_data()
    rw_monitor.start_write()
    write_database()
    rw_monitor.end_write()
```

# Problema dei lettori e scrittori: soluzione n.3 basata sui monitor

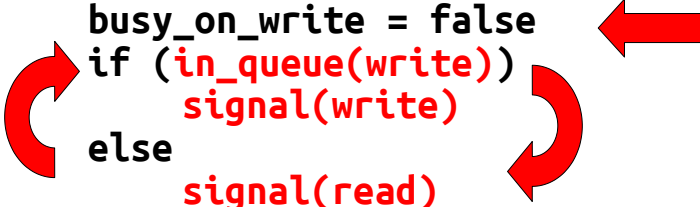
```
monitor rw_monitor
  int rc = 0; boolean busy_on_write = false
  condition read, write
```

```
function start_read()
  if (busy_on_write OR in_queue(write)) wait(read)
  rc = rc+1
  signal(read)
```

```
function end_read()
  rc = rc-1
  if (rc = 0) signal(write)
```

```
function start_write()
  if (rc > 0 OR busy_on_write) wait(write)
  busy_on_write = true
```

```
function end_write()
  busy_on_write = false
  if (in_queue(write))
    signal(write)
  else
    signal(read)
```

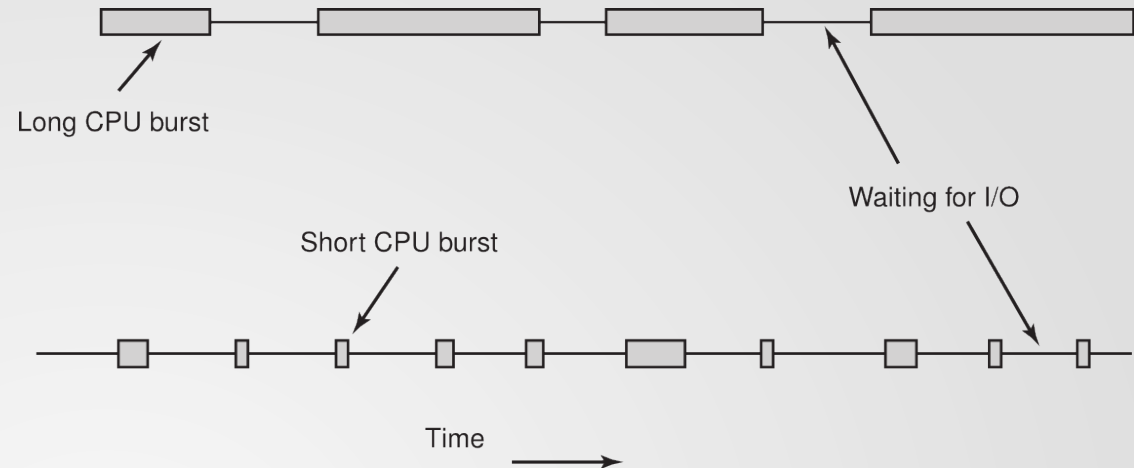


```
function reader()
  while true do
    rw_monitor.start_read()
    read_database()
    rw_monitor.end_read()
    use_data_read()
```

```
function writer()
  while true do
    think_up_data()
    rw_monitor.start_write()
    write_database()
    rw_monitor.end_write()
```

# Scheduling

- **Scheduler;**
- **algoritmo di scheduling;**
- tipologie di processi:
  - **CPU-bounded;**
  - **I/O-bounded;**
- **quando** viene attivato lo scheduler:
  - terminazione (e creazione) di processi;
  - chiamata bloccante (es., I/O) e arrivo del relativo interrupt;
  - interrupt periodici:
    - sistemi **non-preemptive** (senza prelazione);
    - sistemi **preemptive** (con prelazione);
- collabora con il **dispatcher: latenza di dispatch.**



# Obiettivi degli algoritmi di scheduling

- Ambienti differenti: **batch**, **interattivi** e **real-time**.
- **Obiettivi comuni:**
  - **equità** nell'assegnazione della CPU;
  - **bilanciamento** nell'uso delle risorse;
- Obiettivi tipici dei sistemi batch:
  - massimizzare il **throughput** (o produttività);
  - minimizzare il **tempo di turnaround** (o tempo di completamento);
  - minimizzare il **tempo di attesa**;
- Obiettivi tipici dei sistemi interattivi:
  - minimizzare il **tempo di risposta**;
- Obiettivi tipici dei sistemi real-time:
  - **rispetto delle scadenze**;
  - **prevedibilità**.

# Scheduling nei sistemi batch

- **First-Come First-Served (FCFS)** o per ordine di arrivo;
  - non-preemptive;
  - semplice coda FIFO.
- **Shortest Job First (SJF)** o per brevità:
  - non-preemptive;
  - presuppone la conoscenza del tempo impiegato da ogni lavoro;
  - ottimale solo se i lavori sono tutti subito disponibili.
- **Shortest Remaining Time Next (SRTN):**
  - versione preemptive dello SJF.

## Esempio

<u>Processo</u>	<u>Durata</u>
P <sub>1</sub>	24
P <sub>2</sub>	3
P <sub>3</sub>	3

$$\text{t.m.a.: } (0+24+27)/3 = 17$$

$$\text{t.m.c.: } (24+27+30)/3 = 27$$

## Esempio SJF non è ottimale

<u>Processo</u>	<u>Arrivo</u>	<u>Durata</u>
P <sub>1</sub>	0	2
P <sub>2</sub>	0	4
P <sub>3</sub>	3	1
P <sub>4</sub>	3	1
P <sub>5</sub>	3	1

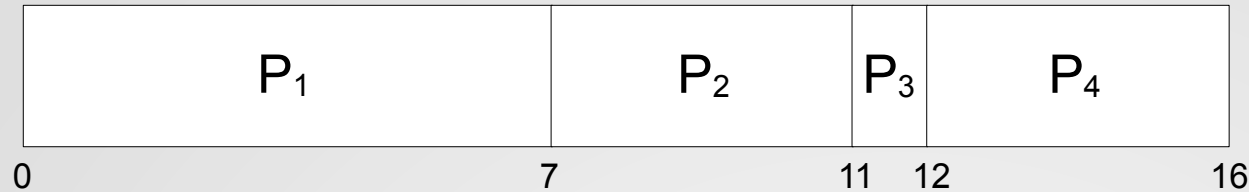
t.m.a.

$$\text{SJF } (0+2+3+4+5)/5 = 2.8$$

$$\text{altern. } (7+0+1+2+3)/5 = 2.6$$

# Scheduling nei sistemi batch

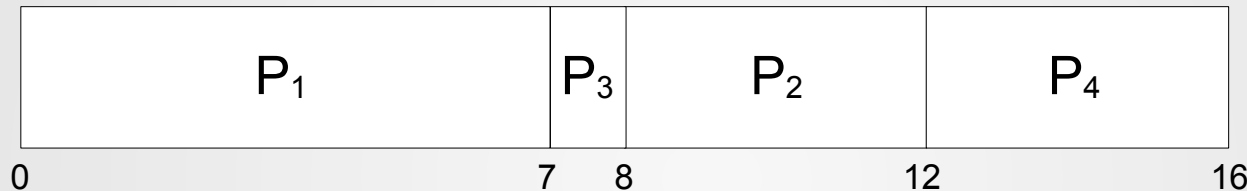
- FCFS:**



Processo	Arrivo	Durata
P <sub>1</sub>	0	7
P <sub>2</sub>	2	4
P <sub>3</sub>	4	1
P <sub>4</sub>	5	4

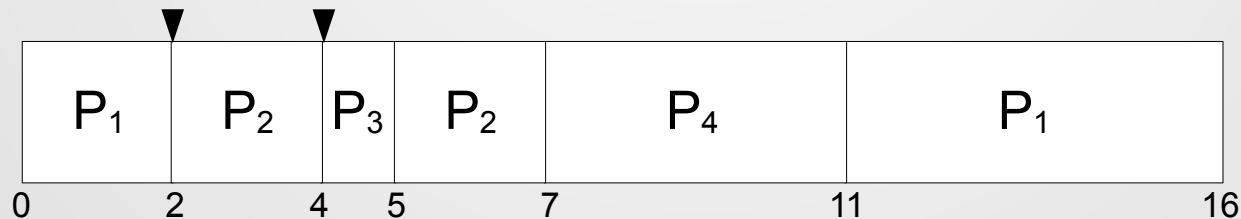
- tempi di attesa: P<sub>1</sub>=0; P<sub>2</sub>=5; P<sub>3</sub>=7; P<sub>4</sub>=7 (media 4.75);
- tempi di completamento: P<sub>1</sub>=7; P<sub>2</sub>=9; P<sub>3</sub>=8; P<sub>4</sub>=11 (media 8.75);

- SJF:**



- tempi di attesa: P<sub>1</sub>=0; P<sub>2</sub>=6; P<sub>3</sub>=3; P<sub>4</sub>=7 (media 4);
- tempi di completamento: P<sub>1</sub>=7; P<sub>2</sub>=10; P<sub>3</sub>=4; P<sub>4</sub>=11 (media 8);

- SRTN:**

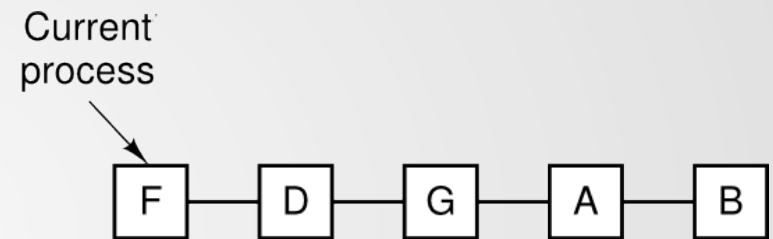
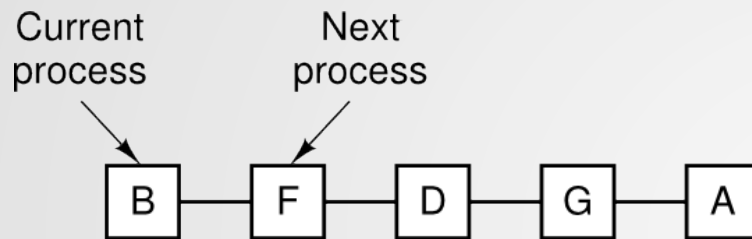


- tempi di attesa: P<sub>1</sub>=9; P<sub>2</sub>=1; P<sub>3</sub>=0; P<sub>4</sub>=2 (media 3);
- tempi di completamento: P<sub>1</sub>=16; P<sub>2</sub>=5; P<sub>3</sub>=1; P<sub>4</sub>=6 (media 7).



# Scheduling nei sistemi interattivi

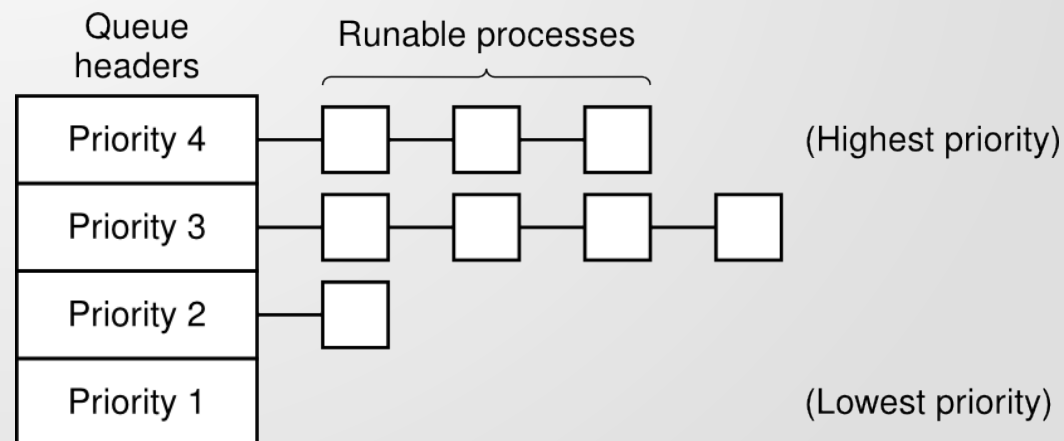
- **Scheduling Round-Robin (RR):**
  - versione con prelazione del FCFS;
  - preemptive e basato su un quanto di tempo (timeslice);



- quanto deve essere lungo il timeslice?
  - valori tipici sono 20-50ms;
- con  $n$  processi e un quanto di  $q$  ms, ogni processo avrà diritto a circa  $1/n$  della CPU e attenderà al più  $(n-1)q$  ms.

# Scheduling nei sistemi interattivi

- **Scheduling a priorità:**
  - **regola di base:** si assegna la CPU al processo con più alta priorità;
  - **assegnamento delle priorità:**
    - statiche, dinamiche;
    - ♦ favorire processi I/O bounded;
    - SJF come sistema a priorità;
  - **prelazione vs. non-prelazione;**
  - **starvation, aging;**
- **Variante: scheduling a code multiple (classi di priorità);**
  - priorità fisse;
  - con feedback (o retroazione).



# Scheduling nei sistemi interattivi

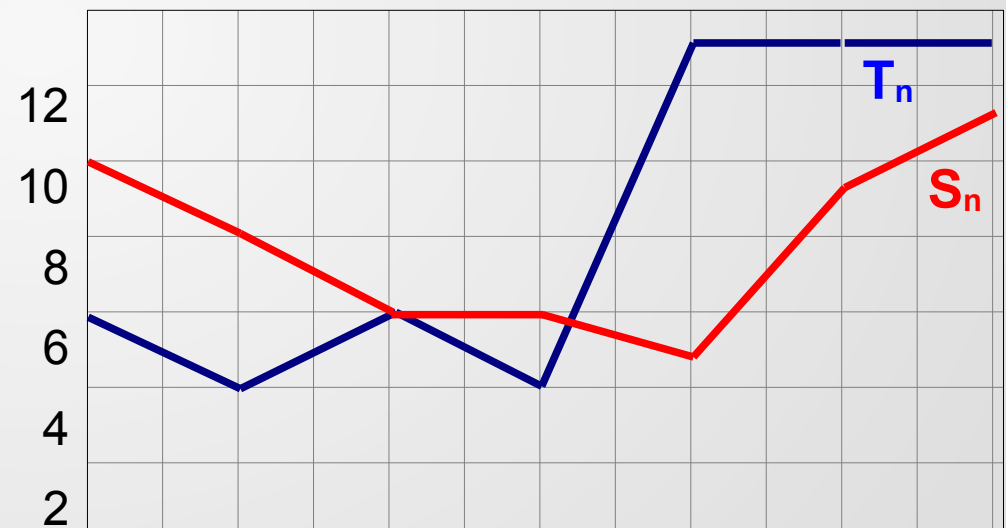
- **Shortest Process Next (SPN):**

- idea: applicare lo SJF ai processi interattivi;
- problema: identificare la durata del prossimo burst di CPU;
- soluzione: stime basate sui burst precedenti;

$$S_{n+1} = S_n(1 - a) + T_n a$$

- esempio:  $a=1/2$

$T_n$	6	4	6	4	13	13	13
$S_n$	10	8	6	6	5	9	11



# Scheduling nei sistemi interattivi

- **Scheduling garantito:**
  - viene stabilita una percentuale di utilizzo e viene fatta rispettare.
- **Scheduling a lotteria:**
  - biglietti con estrazioni a random;
  - criterio semplice e chiaro;
  - possibilità di avere processi cooperanti.
- **Scheduling fair-share:**
  - realizza un equo uso tra gli utenti del sistema.

# Scheduling dei thread

- **Thread utente:**
  - ignorati dallo scheduler del kernel;
  - per lo scheduler del sistema run-time vanno bene tutti gli algoritmi non-preemptive visti;
  - possibilità di utilizzo di scheduling personalizzato.
- **Thread del kernel:**
  - o si considerano tutti i thread uguali, oppure;
  - si pesa l'appartenenza al processo;
    - lo switch su un thread di un processo diverso implica anche la riprogrammazione della MMU e, in alcuni casi, l'azzeramento della cache della CPU.

# Scheduling su sistemi multiprocessore

- Possibili approcci:
  - **multielaborazione asimmetrica;**
    - uno dei processori assume il ruolo di **master server**;
  - **multielaborazione simmetrica (SMP);**
    - **coda unificata** dei processi pronti o **code separate** per ogni processore/core.
- Politiche di scheduling:
  - presenza o assenza di **predilezione per i processori**:
    - **predilezione debole** o **predilezione forte**;
  - **bilanciamento del carico**:
    - necessaria solo in presenza di code distinte per i processi pronti;
    - **migrazione guidata** o **migrazione spontanea**;
      - possibili **approcci misti** (Linux e FreeBSD);
  - bilanciamento del carico vs. predilezione del processore.

# Cosa usano i nostri Sistemi Operativi?

- elementi comuni:
  - thread, SMP, gestione priorità, predilezione per i processi IO-bounded
- **Windows:**
  - scheduler basato su code di priorità;
  - euristiche per migliorare il servizio dei processi interattivi e in particolare di foreground;
  - euristiche per evitare il problema dell'inversione di priorità.
- **Linux:**
  - scheduling basato su task (generalizzazione di processi e thread);
  - Completely Fair Scheduler (CFS): moderno scheduler garantito;
- **MacOS:**
  - Mach scheduler basato su code di priorità con euristiche.