

# Processi, Thread, IPC & Scheduling

## Sistemi Operativi (MZ)



[ComplexIntelligentSystems@gmail.com](mailto:ComplexIntelligentSystems@gmail.com)

<http://cis-lab.dmi.unict.it/>

**Mario Pavone**

[mario.pavone@unict.it](mailto:mario.pavone@unict.it)

<https://www.dmi.unict.it/mpavone/>

<https://www.dmi.unict.it/mpavone/so.html>

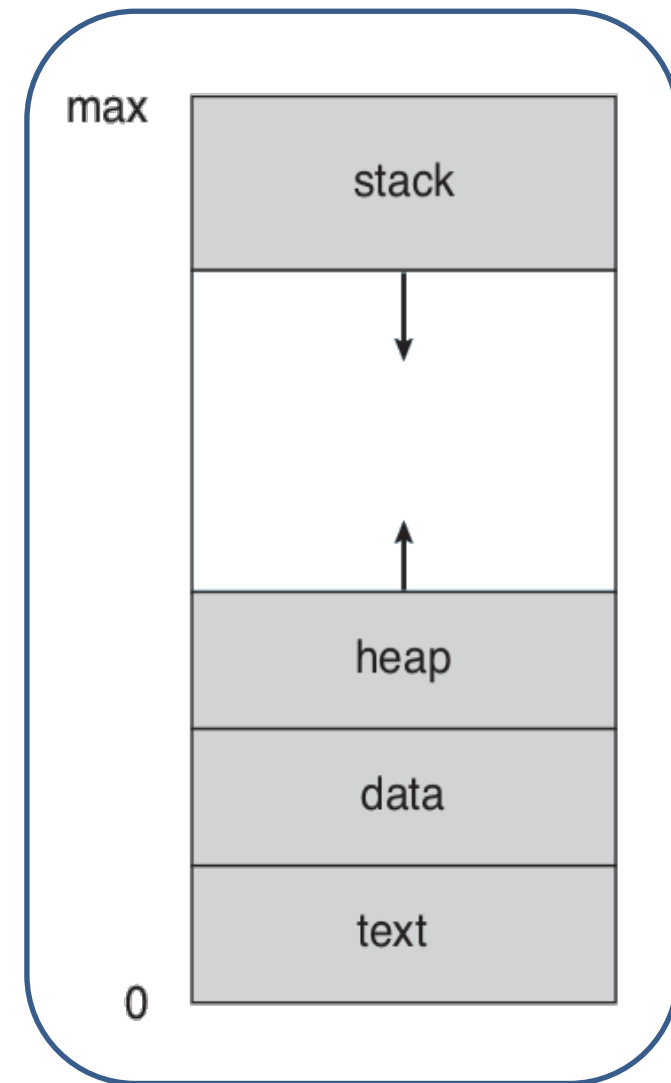
**C.d.L. in Informatica**

**(laurea triennale)**

AA 2025-2026

# Processo

- **Definizione:** una istanza di esecuzione di un programma.
- **Programma vs. Processo**
- Ad ogni processo è associato il suo **spazio degli indirizzi**:
  - codice eseguibile;
  - dati del programma;
  - stack;
  - copia dei **registri** della CPU;
  - **file aperti**;
  - **allarmi** pendenti;
  - processi imparentati.
- Tutte le informazioni relative al processo devono essere salvate (es. file aperti)
- **Tabella dei processi** con un **Process Control Block** (PCB) per ogni processo.



# Processo

- **Definizione:** una istanza di esecuzione di un programma

- Programmazione

- Ad ogni processo  
degli indirizzi

- codice

- dati

- stack

- copia dei

- file aperti

- allarmi

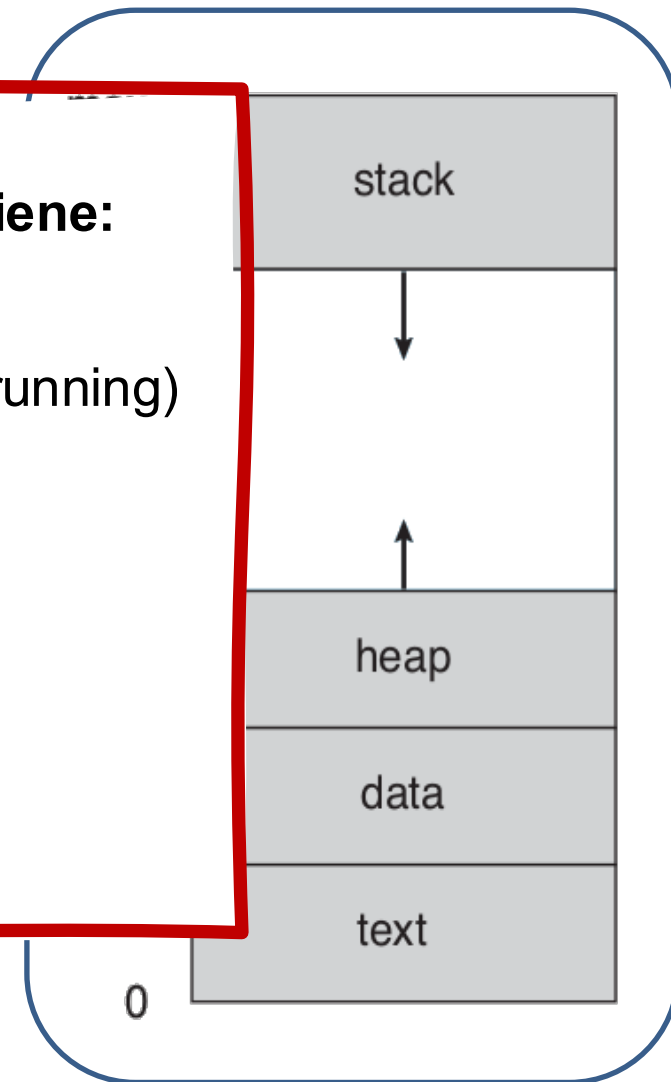
- processi

## Il PCB (Process Control Block) contiene:

- PID (Process ID)
- Stato del Processo (ready, blocked, running)
- Registri della CPU salvati
- Informazioni di scheduling
- Permessi e credenziali
- Puntatore allo spazio degli indirizzi
- Gestione risorse (file aperti, etc.)

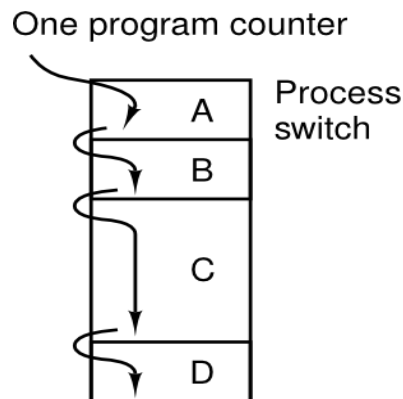
- Tutte le informazioni relative al processo devono essere salvate (es. file aperti)

- **Tabella dei processi** con un **Process Control Block** (PCB) per ogni processo.

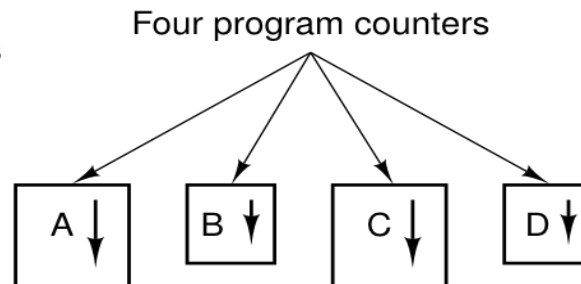


# Modello dei processi

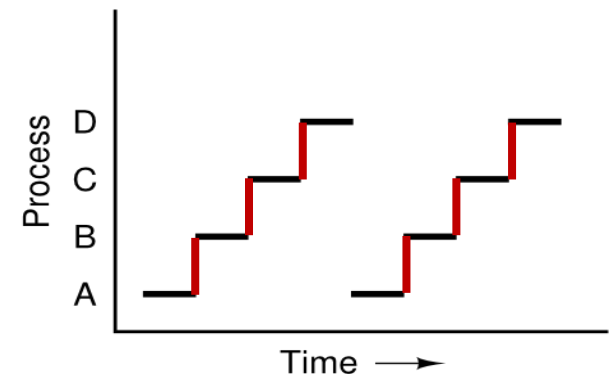
- **Multiprogrammazione e pseudo-parallelismo.**
- È più semplice ragionare pensando a **processi sequenziali** con una **CPU virtuale** dedicata.



(a)



(b)

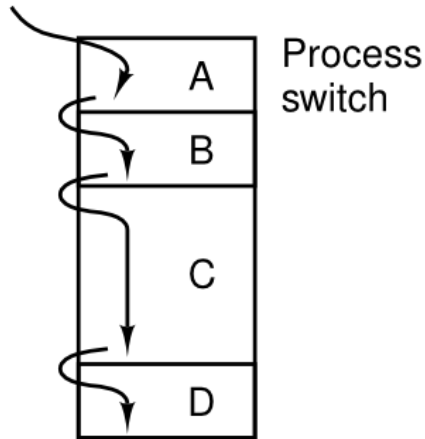


(c)



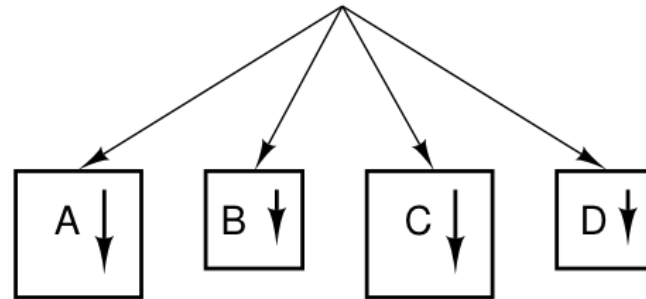
# Modello dei processi

One program counter

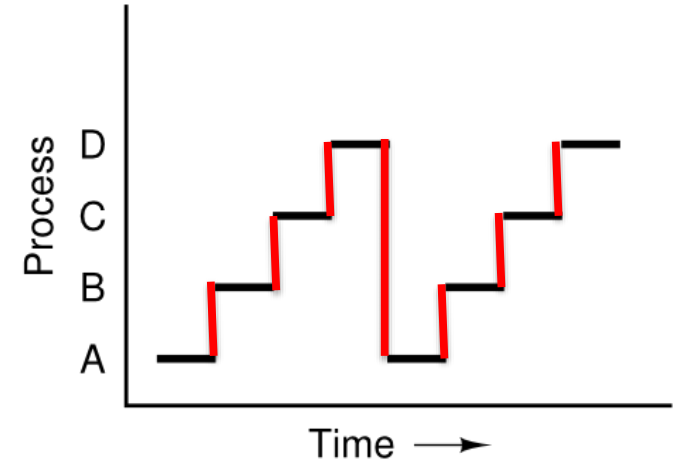


(a)

Four program counters



(b)



(c)

- Esiste un solo PC fisico:
  - **PC logico** viene caricato nel **PC fisico**
  - STOP => PC fisico viene memorizzato nel PC logico

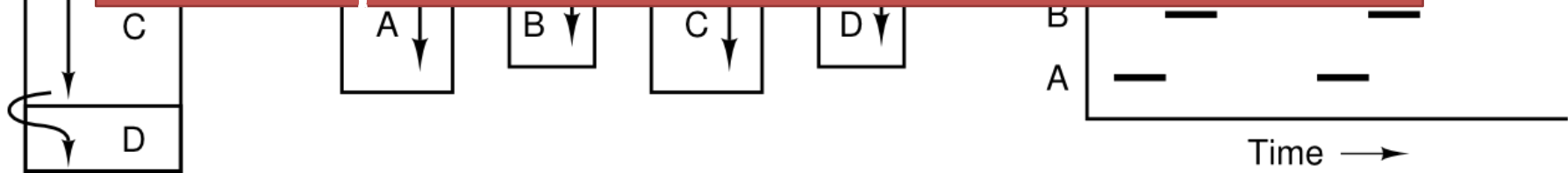
# Modello dei

- Multiprogrammazione e pseudo-parallelismo.

pr **NOTA:** due processi che eseguono lo stesso programma sono comunque distinti

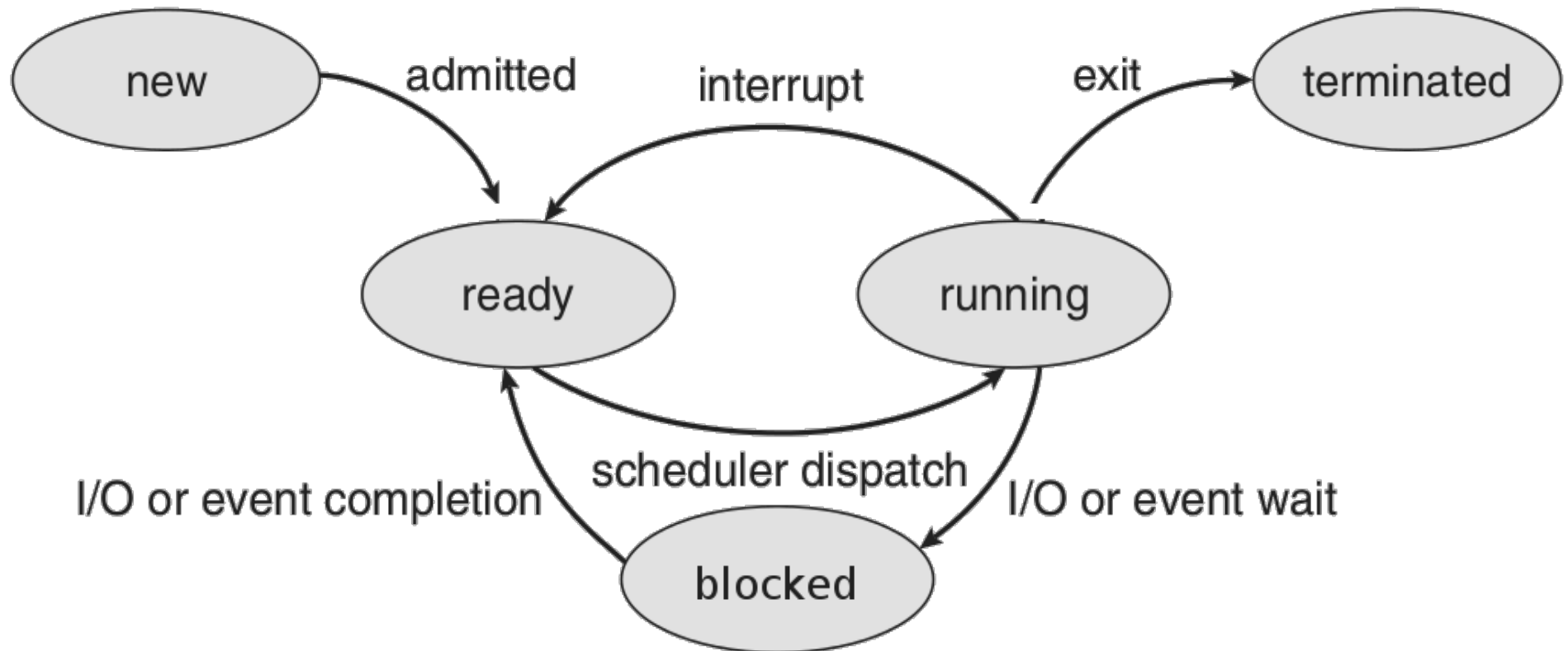
One prog

**ATTENZIONE:** i processi non devono essere programmati in base alle loro tempistiche (requisiti critici Real-Time)



# Stato di un processo

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



# Creazione dei processi

- **Creazione** di un processo:
  - in fase di inizializzazione del sistema;
    - processi attivi e in background (**daemon**)
  - esecuzione chiamate di sistema di creazione processo
  - richiesta dell'utente
- Creare nuovo processo da un altro processo:
  - **sdoppiamento del padre**: **fork** e **exec** (UNIX);
  - **nuovo processo per nuovo programma**: **CreateProcess** (Win32).
- **NOTA**: dopo la creazione del nuovo processo il padre e il figlio hanno il ***proprio spazio degli indirizzi***.
  - una modifica nello spazio di indirizzo di uno non è visibile all'altro



# terminazione dei processi

- **uscita normale:** *exit* (UNIX), *ExitProcess* (Win32);
- **uscita su errore causato dal processo;**
- **errore critico:** alcuni sono gestibili, altri no;
- **terminato da un altro processo:** *kill* (UNIX), *TerminateProcess* (Win32).

# Gerarchia dei Processi

Un processo può crearne un'altro

processo padre e processo figlio

processo figlio può a sua volta crearne altri

**UNIX:** process group

- processo speciale **init**

**In Windows:** NO

- Tutti i processi sono uguali
- **handle**: token per il processo padre

# Tabella dei processi

|                    |
|--------------------|
| process state      |
| process number     |
| program counter    |
| registers          |
| memory limits      |
| list of open files |
| ...                |

- SO mantiene una **Tabella dei processi**;
- **Process Control Block (PCB)**;
- Contiene tutte le informazioni importanti sullo stato del processo
- **Scheduler**

| Process management                                                                                                                                                                                                                                                                | Memory management                                                                             | File management                                                                |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| Registers<br>Program counter<br>Program status word<br>Stack pointer<br>Process state<br>Priority<br>Scheduling parameters<br>Process ID<br>Parent process<br>Process group<br>Signals<br>Time when process started<br>CPU time used<br>Children's CPU time<br>Time of next alarm | Pointer to text segment info<br>Pointer to data segment info<br>Pointer to stack segment info | Root directory<br>Working directory<br>File descriptors<br>User ID<br>Group ID |

Processes

|           |   |     |         |         |
|-----------|---|-----|---------|---------|
| 0         | 1 | ... | $n - 2$ | $n - 1$ |
| Scheduler |   |     |         |         |

- **NOTA:** i campi della tabella dipendono dal SO

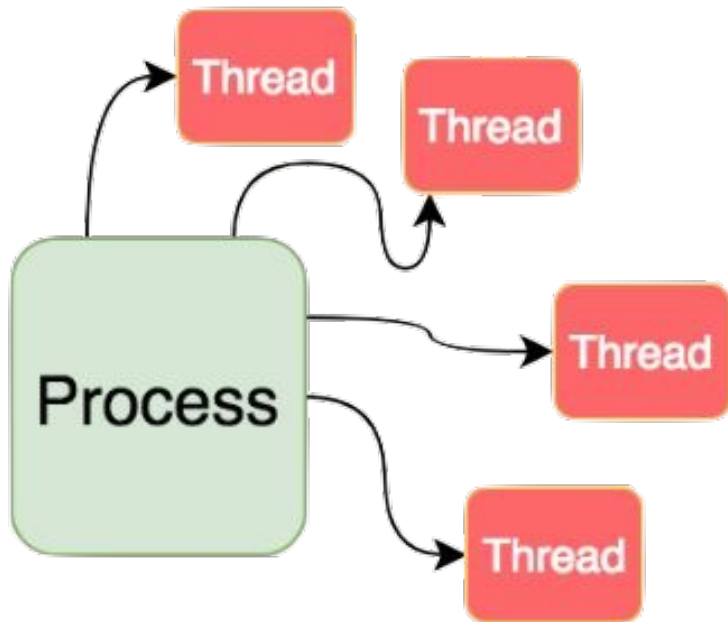
# Sistemi Operativi (M-Z)

## I Thread

C.d.L. in Informatica  
(laurea triennale)  
A.A. 2025-2026

**Prof. Mario F. Pavone**

**Dipartimento di Matematica e Informatica  
Università degli Studi di Catania  
[mario.pavone@unict.it](mailto:mario.pavone@unict.it)**

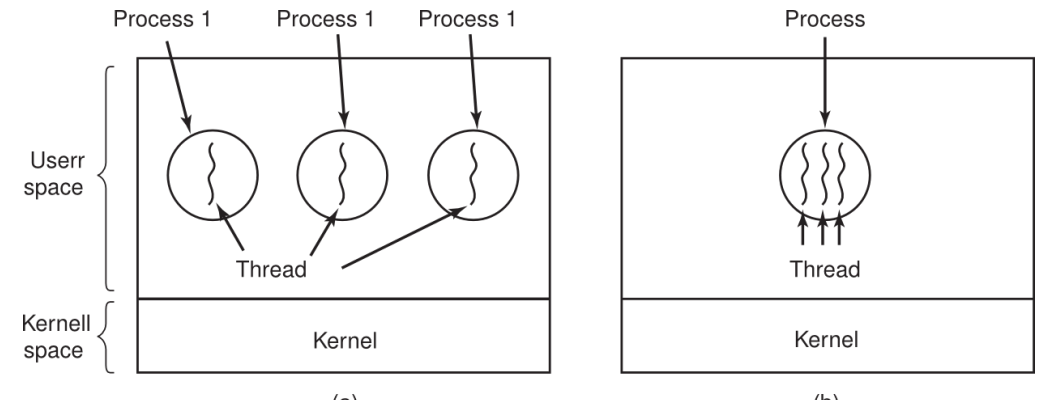


[ComplexIntelligentSystems@gmail.com](mailto:ComplexIntelligentSystems@gmail.com)

<http://cis-lab.dmi.unict.it/>



# 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, ...

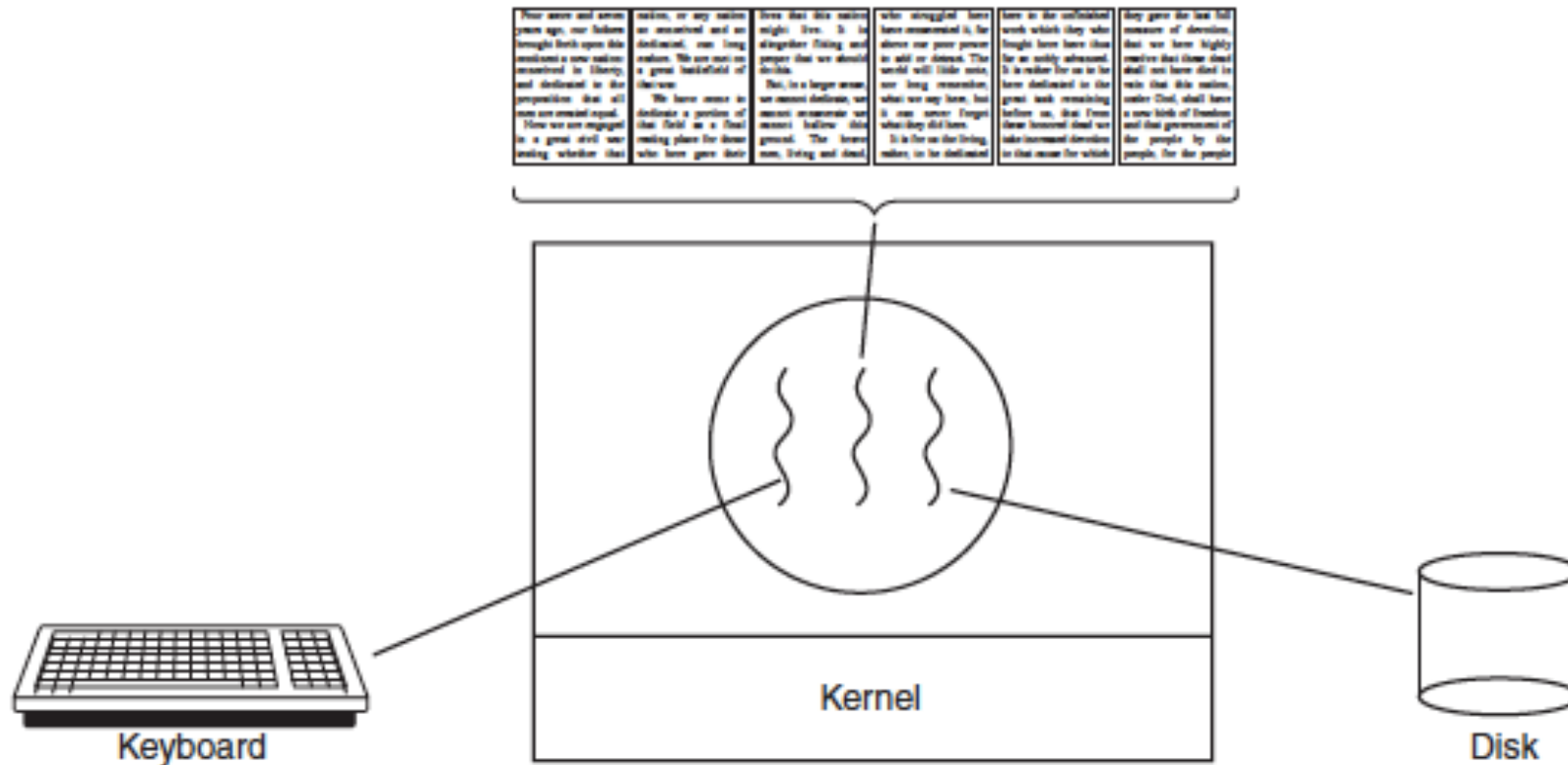


Figure 2-7. A word processor with three threads.

Thread: an  
example

• WORD PROCESSOR

# Thread: an example

## • WEB SERVER PROCESS

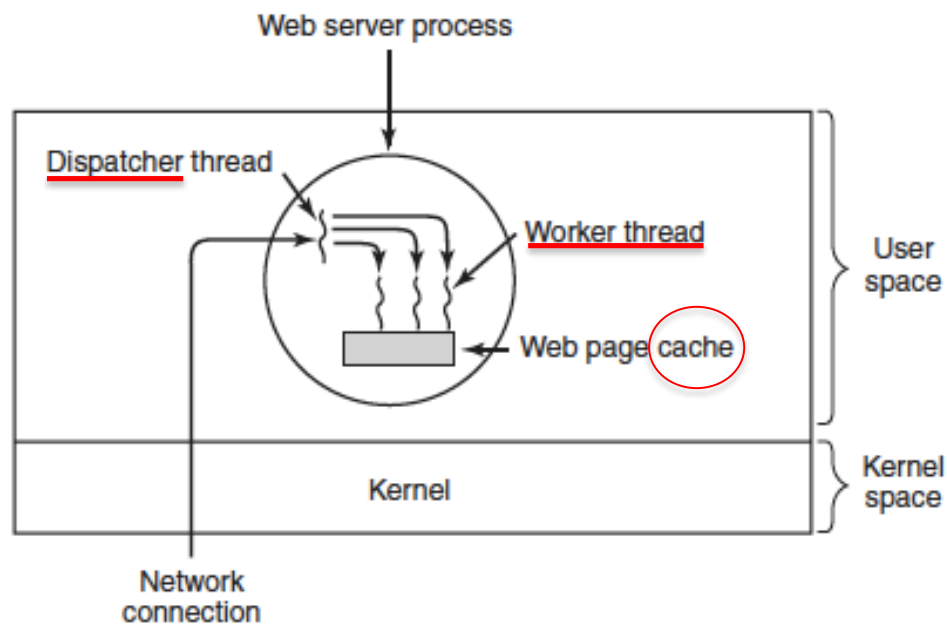
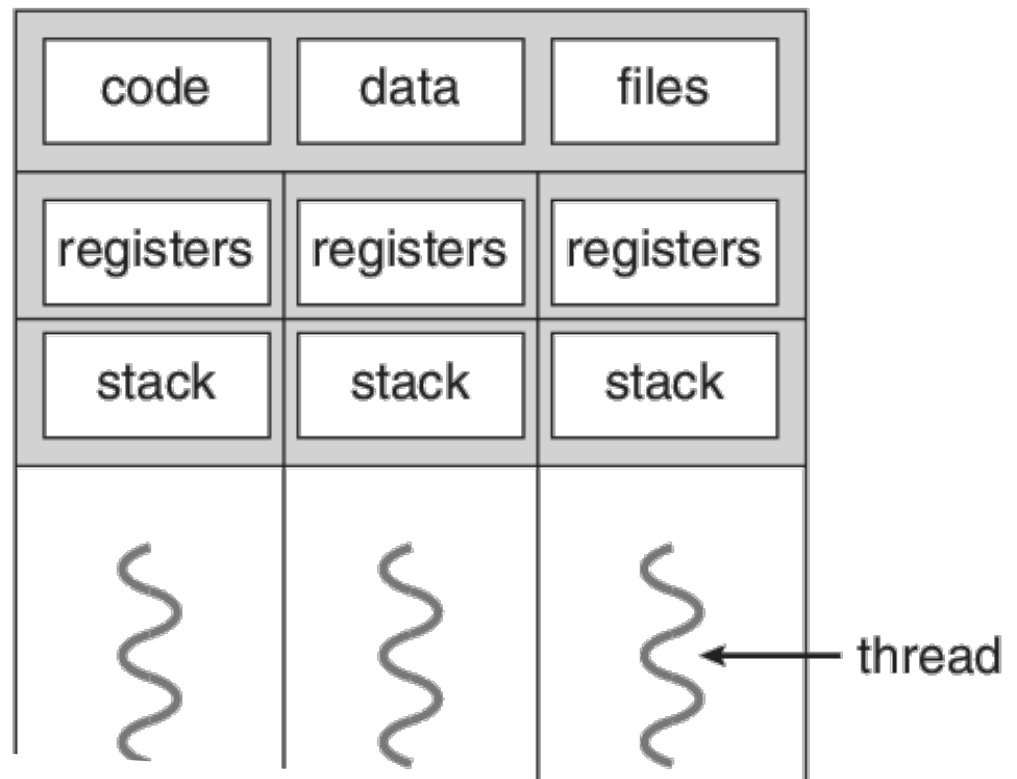
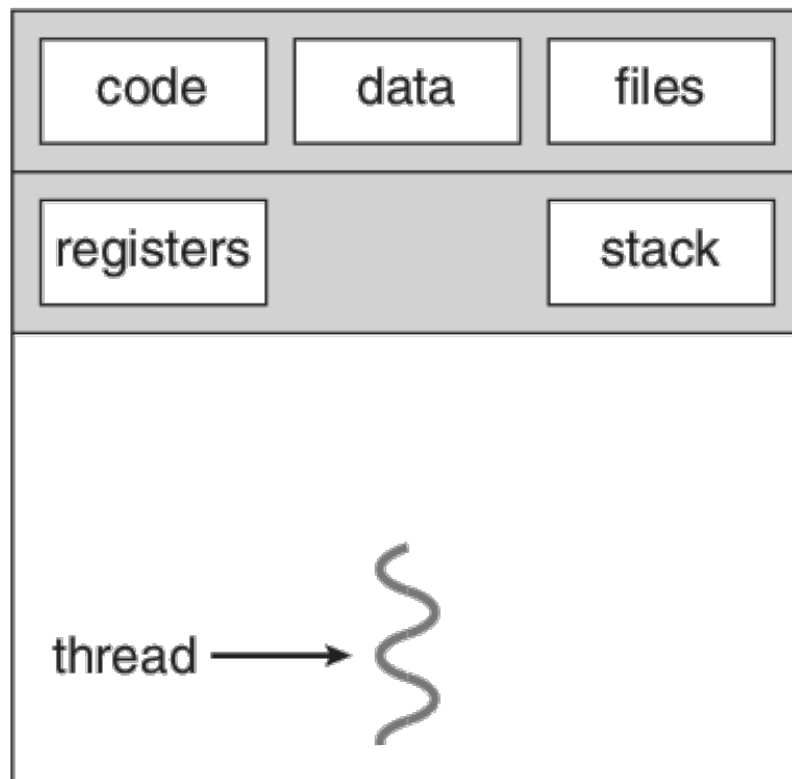


Figure 2-8. A multithreaded Web server.

*NOTA: il dispatcher è un ciclo infinito che acquisisce richieste di lavoro e le passa ad un worker thread...*

*... anche quello del worker thread è un ciclo infinito*

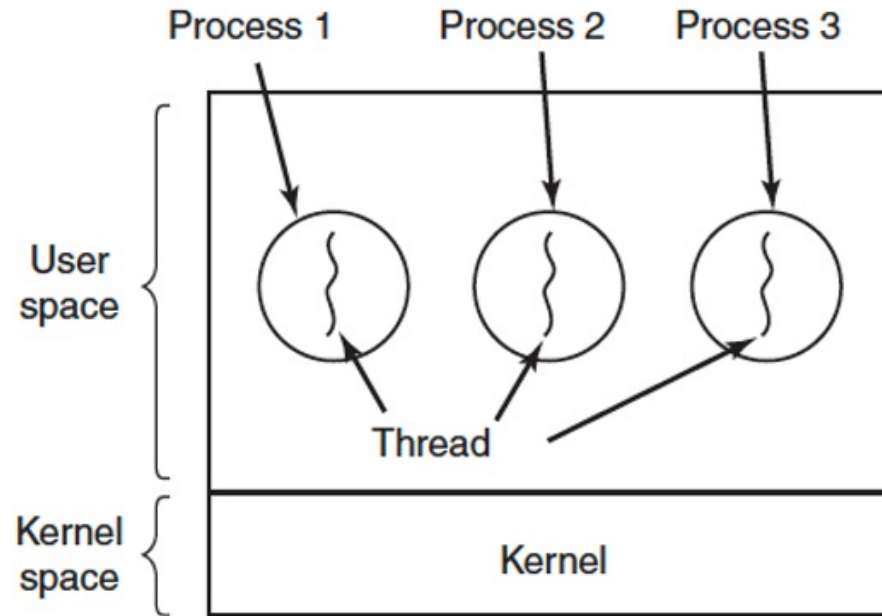


# Il Modello Thread

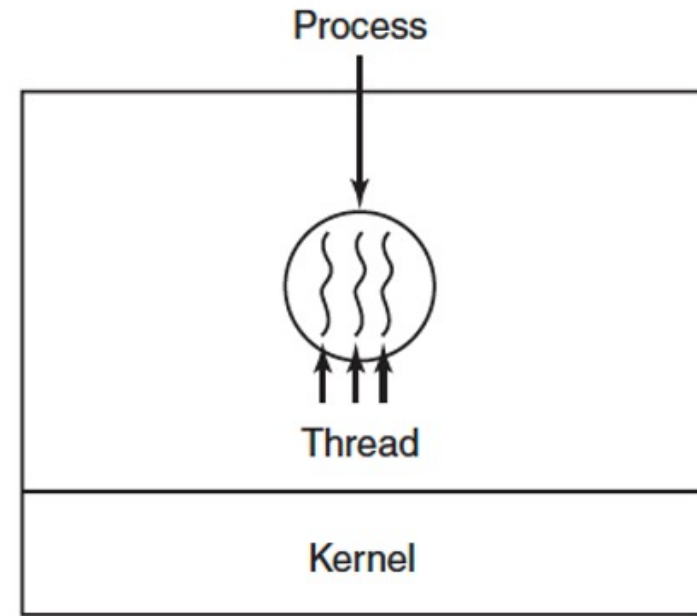
- ✓ **Processo**: raggruppare risorse
- ✓ **Thread**: esecuzione CPU

- 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;

*Sistemi Multithreading*



(a)



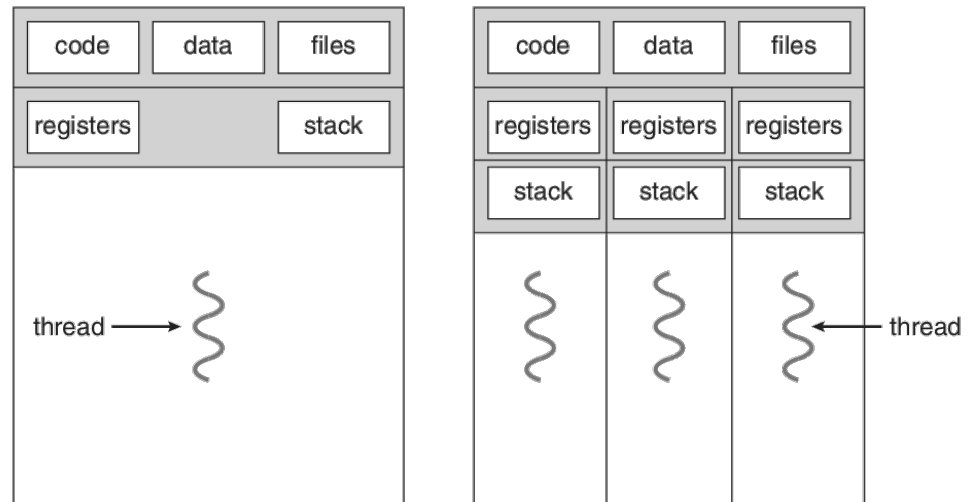
(b)

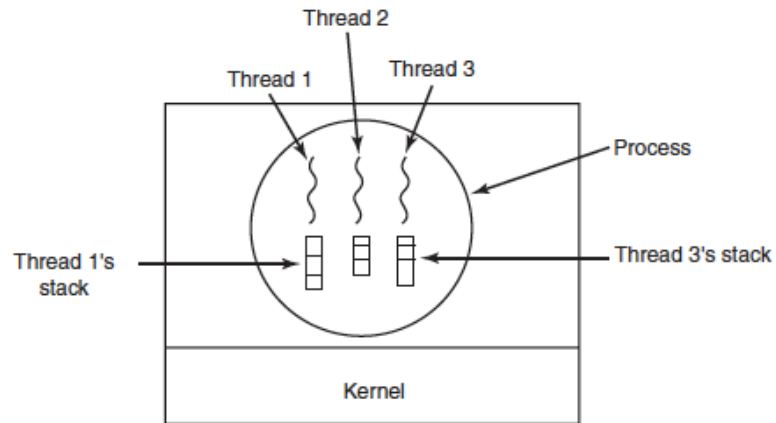
# Thread Vs. Processo

- Fig. (a): Ogni processo lavora in spazi degli indirizzi diversi
- Fig. (b): tutti i thread condividono lo stesso spazio degli indirizzi

# Thread

- Thread diversi nello stesso processo non sono indipendenti
  - Condivisione e **No protezione di memoria**
    - un thread può leggere, scrivere o cancellare lo stack di un altro thread
  - Stesso user -> Cooperano -> non entrano in conflitto
  - Thread apre file => è visibile ad ogni thread nel processo
- **Unità di gestione delle risorse => Processo**



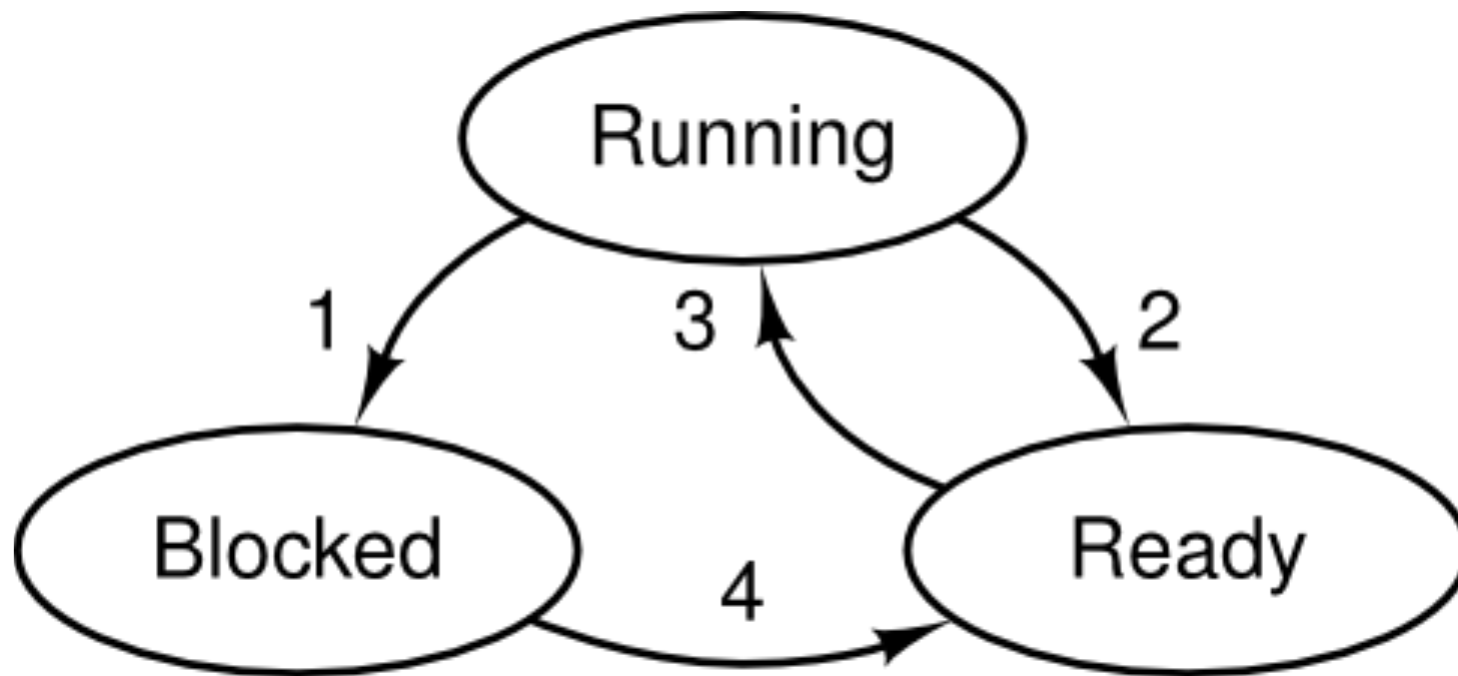


| Per-process Items           | Per-thread Items |
|-----------------------------|------------------|
| Address space               | Program counter  |
| Global variables            | Registers        |
| Open files                  | Stack            |
| Child processes             | State            |
| Pending alarms              |                  |
| Signals and signal handlers |                  |
| Accounting information      |                  |

# Thread

- Ogni Thread ha un proprio Stack
- **storia di esecuzione diversa**

NOTA: processo inizia con un solo thread  
=> altri thread verranno creati



# Thread

- Anche un thread può trovarsi in uno dei seguenti stati
- Transizioni di stato => analogo a quello dei processi.

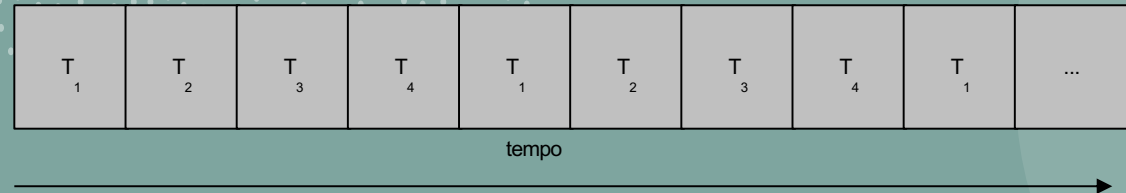


# Operazioni sui Thread

- Nei sistemi Multithreading i processi iniziano con un singolo thread
- **operazioni** tipiche sui thread:
  - **thread\_create**: crea un nuovo thread e specifica la procedura che deve eseguire;
  - **thread\_exit**: il thread chiamante termina;
  - **thread\_join**: un thread si sincronizza con la fine di un altro thread (attende che uno specifico thread termini);
  - **thread\_yield**: il thread chiamante rilascia **volontariamente** la CPU
    - **no clock interrupt**

**Libreria Pthread (sistemi Unix)**

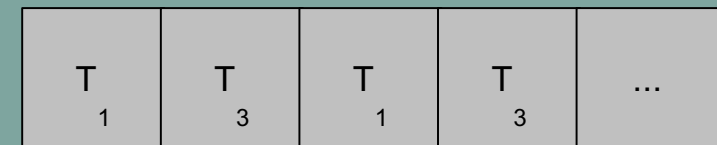
singola  
CPU



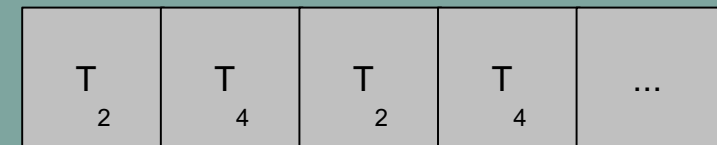
# Programmazione multicore

- I thread permettono una **migliore scalabilità** con core con hypertexting e soprattutto con sistemi multicore;
- con un sistema single-core abbiamo una esecuzione interleaved;
- su un sistema multi-core abbiamo parallelismo puro.

core 0



core 1



tempo



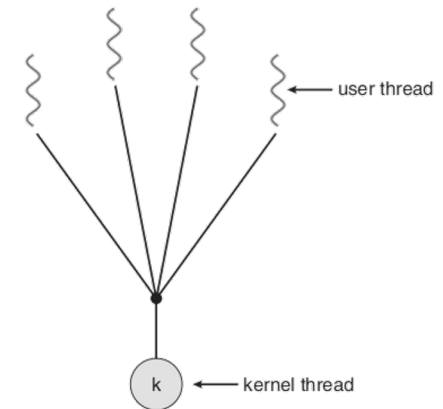
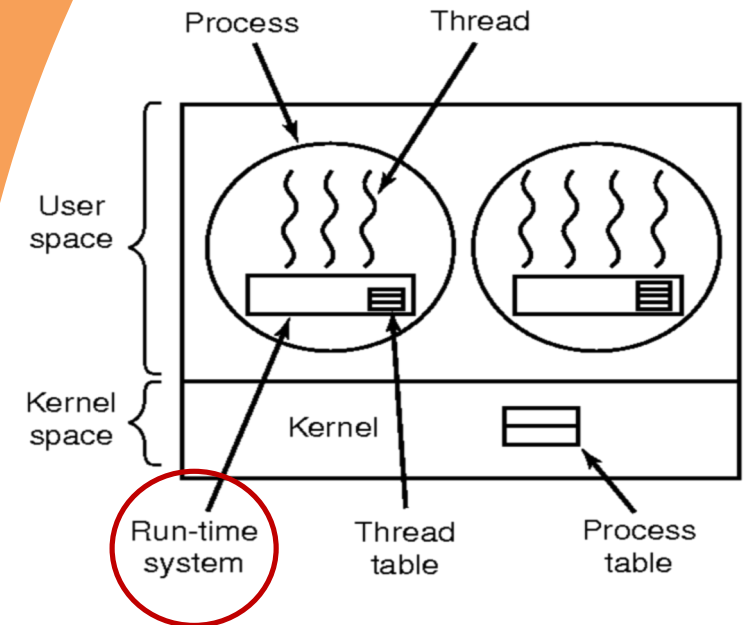


# Programmazione multicore

- Progettare programmi che sfruttino le moderne architetture multicore non è banale;
- **principi base:**
  - **separazione dei task;**
    - trovare aree per *attività separate e simultanee*
  - **bilanciamento;**
    - assicurarsi che eseguano lo *stesso lavoro di uguale valore*
  - **suddivisione dei dati;**
    - dividere i dati *per essere eseguiti su core separati*
  - **dipendenze dei dati;**
    - esecuzione delle *attività sia sincronizzata*
  - **test e debugging**

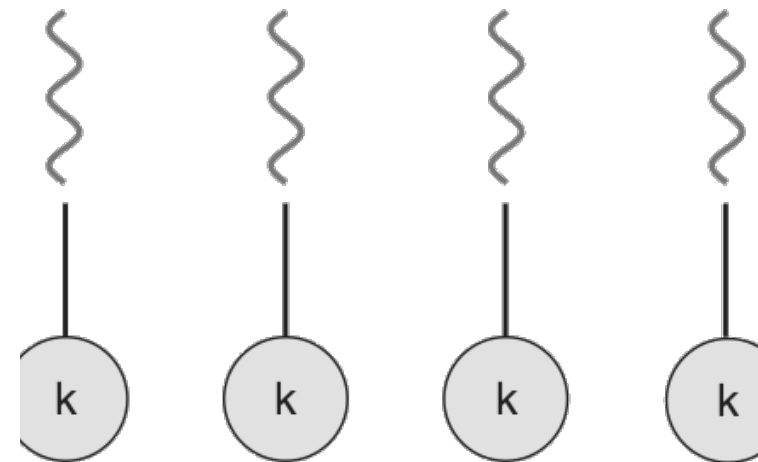
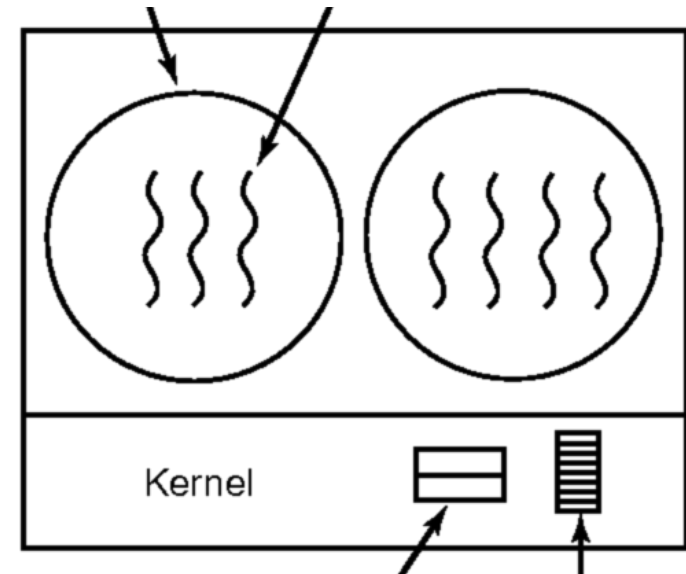
# Thread a livello utente

- 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.



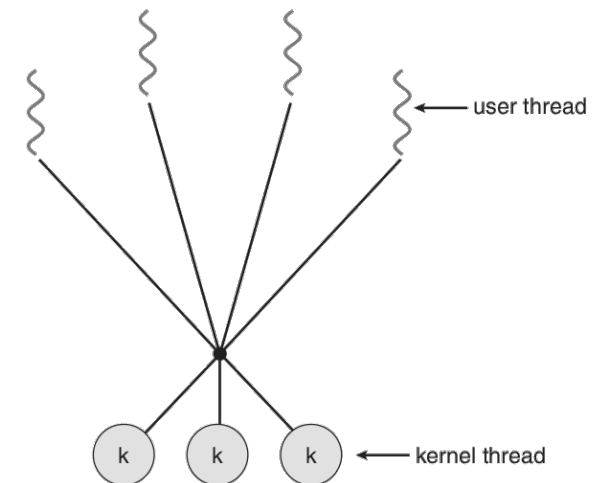
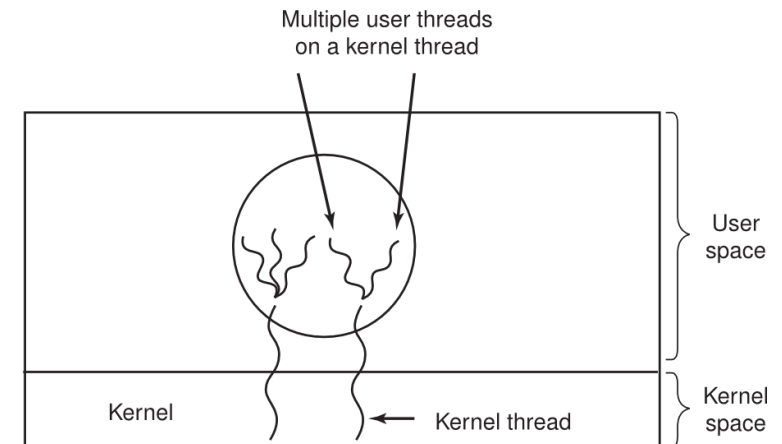
# Thread a livello kernel

- 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 X,...
- 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 X, anche Windows);
    - una specifica da implementare sui vari sistemi;
  - threads Win32;
  - thread in Java;
    - wrapper sulle API sottostanti.

# Sistemi Operativi (M-Z)

Prof. Mario F. Pavone



C.d.L. in Informatica  
(laurea triennale)  
A.A. 2025-2026



Dipartimento di Matematica e Informatica  
Università degli Studi di Catania  
[mario.pavone@unict.it](mailto:mario.pavone@unict.it)

## Comunicazione fra Processi (InterProcess Communication)

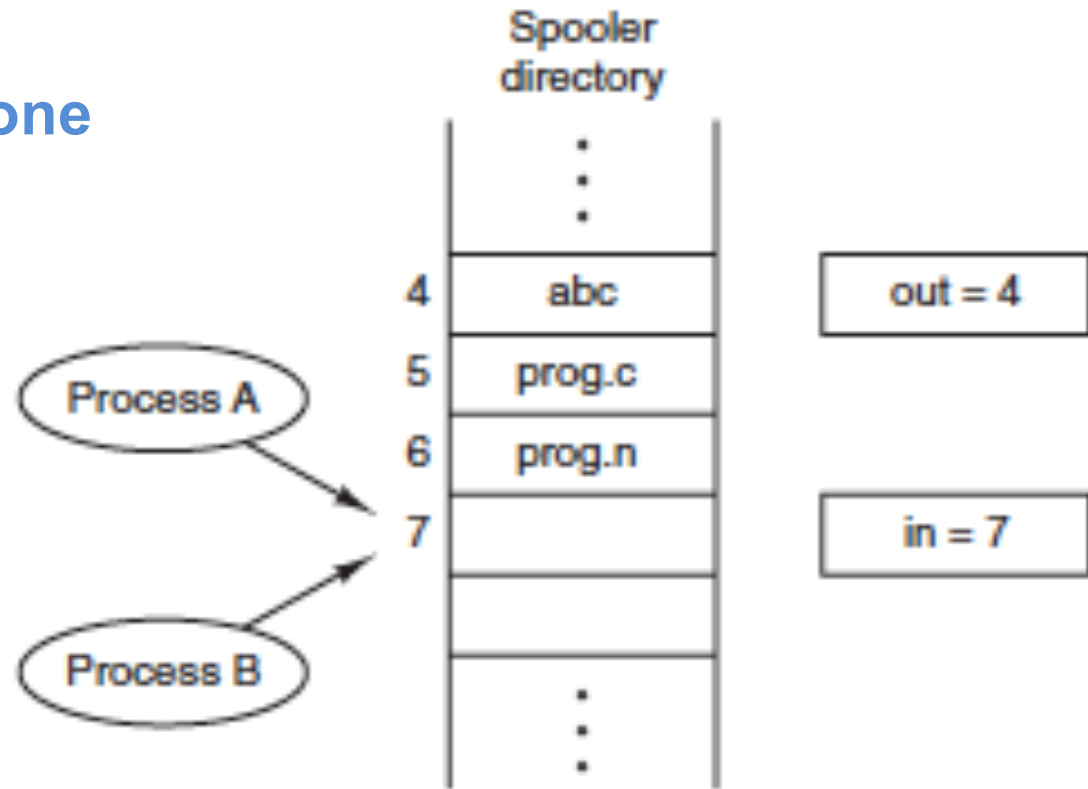


# Comunicazione fra processi

- 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**;
    - kernel **preemptive** vs. **non-preemptive**;
  - **soluzione**: **mutua esclusione** nell'accesso ai dati condivisi.

# Race Conditions

- **Spool di Stampa: demone di stampa** & directory di spool

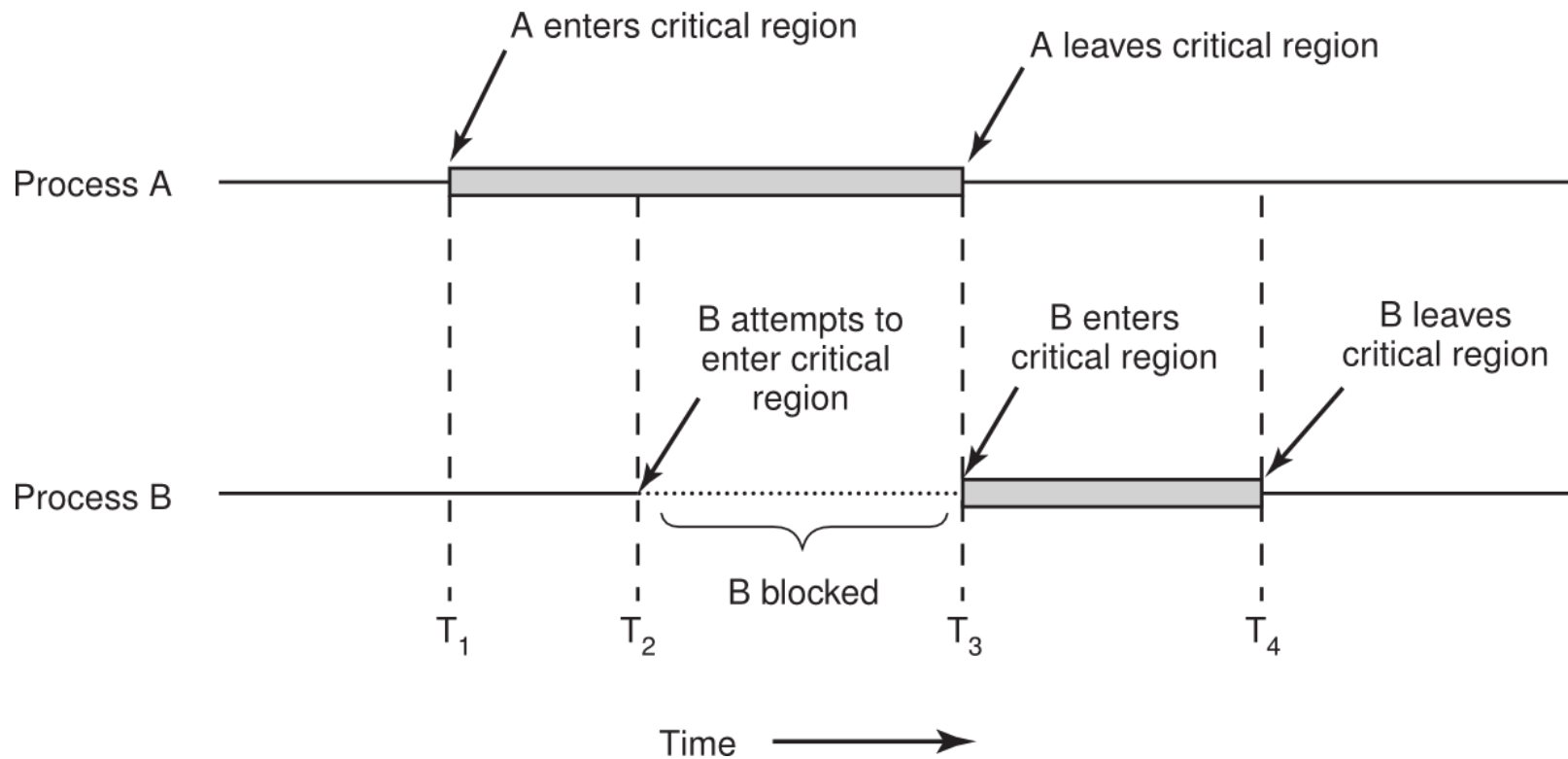


Evitare che più processi **accedino a dati condivisi contemporaneamente**

# Regioni Critiche

- **Mutua Esclusione**: garantire che il processo  $B$  utilizzi le risorse condivise **solo dopo** che il processo  $A$  termini il suo completo utilizzo
- **Regione/Sezione Critica**: parte di programma in cui si accede alla memoria condivisa

# Sezioni critiche



# 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.

# Come realizzare la mutua esclusione

- **Disabilitare gli interrupt**
  - scelta non saggia => blocco del sistema
  - non funziona su sistemi multiprocessore
- **Variabili di lock**
  - soluzione software
  - Variabile condivisa:
    - 0 -> nessun processo è nella regione critica
    - 1 -> qualche processo è nella sua regione critica
  - **stesso problema spool di stampa**

# Come realizzare la mutua esclusione

- **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**).

**fare a turno quando un processo è molto più lento dell'altro non è una buona soluzione**

# 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**.

# Istruzioni TSL e XCHG

enter\_region:

```
MOVE REGISTER,#1
XCHG 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**:
    - gruppo di operazioni eseguite insieme **senza interruzioni**
    - disabilitazione interrupt o spin lock TSL/XCHG;
  - tipicamente implementato **senza busy waiting** con una **lista di processi bloccati**.

# Produttore-consumatore con i semafori

---

- Utilizzo di 3 semafori:
  - *full, empty & mutex*

```
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)
```

- *Mutex*: mutua esclusione

- Diverso utilizzo dei Semafori:
  - **semaforo mutex**: **mutua esclusione**;
  - **semafori full & empty**: **sincronizzazione**.

```
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)
```

# Semafori



ATTENZIONE: L'ordine delle operazioni sui semafori è fondamentale...

# Produttore/Consumatore con i Semafori

- **Attenzione** nell'uso dei semafori

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

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

- I processi potrebbero **rimanere bloccati per sempre** (**deadlock**)

- Semaforo per la **gestione** della **Mutua Esclusione**
  - particolarmente valido in thread spazio utente
- Due stati: **locked** & **unlocked**



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

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

## Mutex e thread utente

- simili a **enter\_region/leave\_region** ma:
  - senza **spin lock**;
  - il **busy waiting** sarebbe problematico con i thread utente;
  - **no clock** per i thread



# Mutex e thread utente

- Qual'è la differenza tra `enter_region` & `mutex_lock`

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

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



Cede la CPU ad un altro thread

# Futex

---

- Osservazione: i mutex in user-space sono molto efficienti ma lo spin lock può essere lungo!
  - Spin lock: spreco cicli CPU
  - Blocco del processo e gestione al Kernel
- → **futex = fast user space mutex** (Linux)
- due componenti:
  - **servizio kernel**
    - coda di thread bloccati
  - **libreria utente**
    - variabile di **lock** (init = 1)
    - contesa in modalità utente (tipo con TSL/XCHG)
    - richiamo kernel solo in caso di bloccaggio

**NOTA: no contese lock => no Kernel call => veloce esecuzione**

# I Monitor

- primitiva di sincronizzazione
- costruito ad alto-livello disponibile su alcuni linguaggi
- **tipo astratto di dato: raccolta di variabili, strutture dati & procedure**
  - tipo di dati astratto (ADT – Abstract Data Type): dati privati con metodi pubblici
- processi: chiamare procedure ma no accesso strutture dati
- **garanzia mutua esclusione: 1 processo attivo alla volta**
  - organizzazione al compilatore
  - convertire regioni critiche in procedure monitor
- vincolo di **accesso ai dati** (interni ed esterni)
- **come bloccare un processo che non può proseguire?**

# I Monitor

```
monitor tipo_risorsa {  
    <dichiarazioni variabili locali>;  
    <inizializzazione variabili locali>;  
  
    public void op1 ( ) {  
        <corpo della operazione op1>;  
    }  
    ...  
    public void opn ( ) {  
        <corpo della operazione opn>;  
    }  
  
    <eventuali operazioni non public>  
}
```

Le operazioni **public** (o entry) sono le **sole operazioni** che possono essere utilizzate dai processi per accedere alle variabili locali. L'accesso avviene in **modo mutuamente esclusivo**.

Le operazioni **non** dichiarate **public** non sono accessibili dall'esterno. Sono invocabili solo all'interno del monitor (dalle funzioni public e da quelle non public).

Le variabili locali sono **accessibili solo all'interno del monitor**.

# I Monitor

- Meccanismo di sincronizzazione: **variabili condizione**
  - operazioni **wait** e **signal**
    - **wait**: blocca processo chiamante
    - **signal**: sveglia il processo in sleep
  - **variabili condizione: *non sono contatori***  
=> non accumulano segnali

- cosa accade dopo signal? (due processi attivi nel monitor)
  - **Hoare**: eseguire il processo svegliato, sospendendosi;
  - **Brinch-Hansen**: signal come ultima istruzione di una procedura monitor
  - continuare l'esecuzione del segnalatore

# I Monitor

La dichiarazione di una variabile cond di tipo condizione ha la forma:

```
condition cond;
```

## Operazioni sulle variabili condition:

- Le operazioni del monitor agiscono su tali variabili mediante le operazioni:

```
wait(cond) ;
```

```
signal(cond) ;
```

## wait:

- L'esecuzione dell'operazione **wait(cond)** sospende il processo, introducendolo nella coda individuata dalla variabile cond, e il monitor viene liberato. Al risveglio, il processo riprende l'esecuzione mutuamente esclusiva all'interno del monitor

## signal:

- L'esecuzione dell'operazione **signal(cond)** rende attivo un processo in attesa nella coda individuata dalla variabile cond; se non vi sono processi in coda, non produce effetti.

# I Monitor

- Meccanismo di sincronizzazione: **variabili condizione**
  - operazioni **wait** e **signal**
    - **wait**: blocca processo chiamante
    - **signal**: sveglia il processo in sleep
  - **variabili condizione: *non sono contatori***  
=> non accumulano segnali
  - cosa accade dopo signal? (due processi attivi nel monitor)
    - **Hoare**: eseguire il processo svegliato, sospendendosi;
    - **Brinch-Hansen**: signal come ultima istruzione di una procedura monitor
    - continuare l'esecuzione del segnalatore

```
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)
```

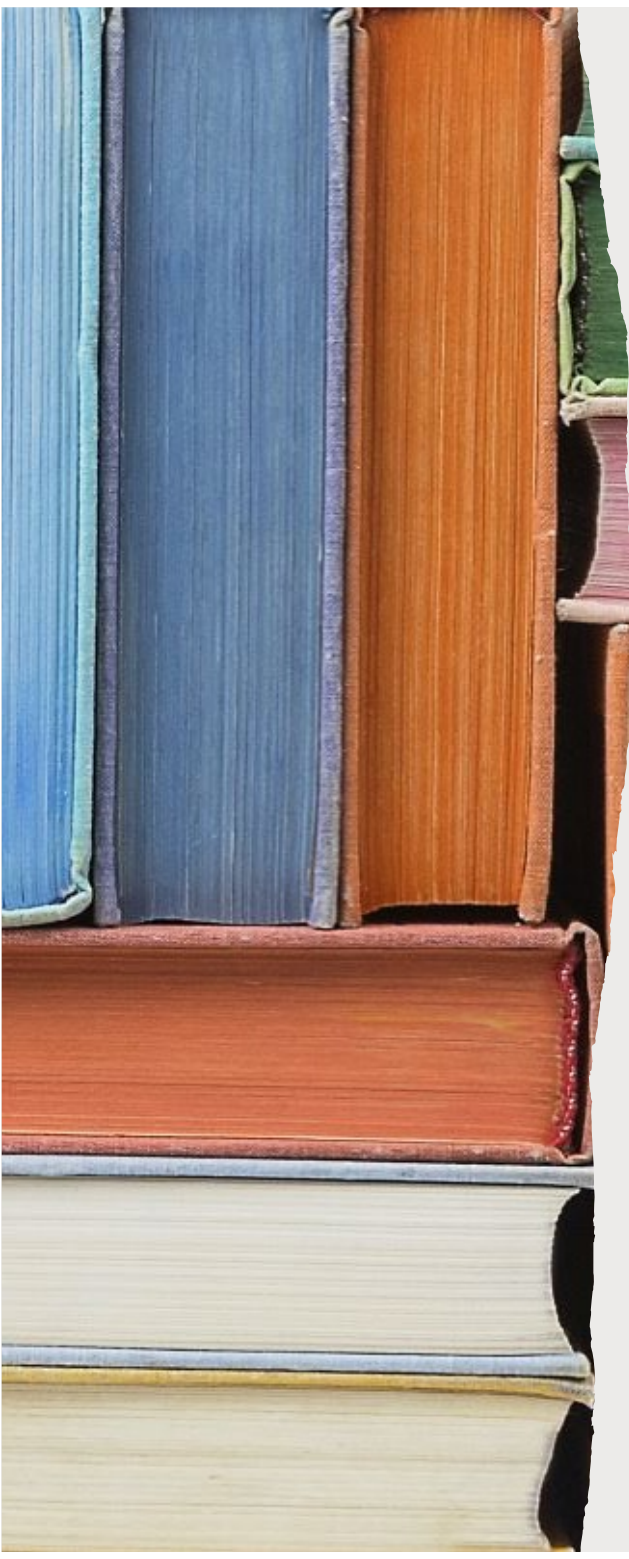
```
function consumer()
```

```
    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)*

} Chiamate di sistema

- **bloccante per il chiamante** (o può restituire un errore);
- estendibile al caso di più macchine (es., *libreria MPI*);
- **perdita messaggi: acknowledgement**
  - **sequenza di numeri**: gestire "vecchio o ritrasmesso"
- **metodi di indirizzamento: diretto** o tramite **mailbox**;



# Scambio messaggi tra processi: MailBox

- assumendo realisticamente l'esistenza di un **buffer** per i messaggi:
  - capienza finita  $N$
  - la mailbox contiene i messaggi spediti ma non ancora accettati
  - la send può essere bloccante
- Strategia **Rendezvous**:  
eliminazione uso buffer
  - **meno flessibile**

# Scambio Messaggi

## Comunicazione asincrona

- Il processo mittente continua la sua esecuzione immediatamente dopo che il messaggio è stato inviato.

- Il messaggio ricevuto contiene informazioni che non possono essere associate allo stato attuale del mittente (difficoltà nella verifica dei programmi).

## Comunicazione sincrona

- Il primo dei due processi comunicanti che esegue o l'invio o la ricezione si sospende in attesa che l'altro sia pronto ad eseguire l'operazione duale.

- Non esiste la necessità dell'introduzione di un buffer; un messaggio può essere inviato solo se il ricevente è pronto a riceverlo.

- Un messaggio ricevuto contiene informazioni corrispondenti allo stato attuale del processo mittente.

# Scambio Messaggi

## Comunicazione asincrona

- **Maggiore parallelismo:** mittente e destinatario possono procedere indipendentemente.
- **Migliori prestazioni:** meno attese, più throughput.
- **Più scalabile:** adatta a sistemi distribuiti e multiprocessore.

### Vantaggi

### Svantaggi

- **Maggiore complessità:** gestione di buffer, code e sincronizzazione.
- **Possibili problemi di overflow:** messaggi accumulati se il destinatario è lento.
- **Ordine meno controllabile:** più difficile garantire sequenze precise.

## Comunicazione sincrona

- **Semplicità concettuale:** sincronizzazione implicita (non servono lock).
- **Ordine e coerenza:** mittente e destinatario sono sincronizzati.
- **Facile da ragionare:** utile in protocolli request/response.

### Vantaggi

### Svantaggi

- **Blocco dei processi:** se il destinatario non è pronto, il mittente si ferma.
- **Bassa efficienza:** possibile spreco di CPU.
- **Scarsa scalabilità:** poco adatta a sistemi molto concorrenti o distribuiti.

# Produttore-Consumatore con Scambio di Messaggi



```
#define N 100                                /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                                /* message buffer */

    while (TRUE) {
        item = produce_item();                /* generate something to put in buffer */
        receive(consumer, &m);                /* wait for an empty to arrive */
        build_message(&m, item);              /* construct a message to send */
        send(consumer, &m);                    /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);                /* get message containing item */
        item = extract_item(&m);              /* extract item from message */
        send(producer, &m);                    /* send back empty reply */
        consume_item(item);                    /* do something with the item */
    }
}
```



# Sistemi Operativi (M-Z)

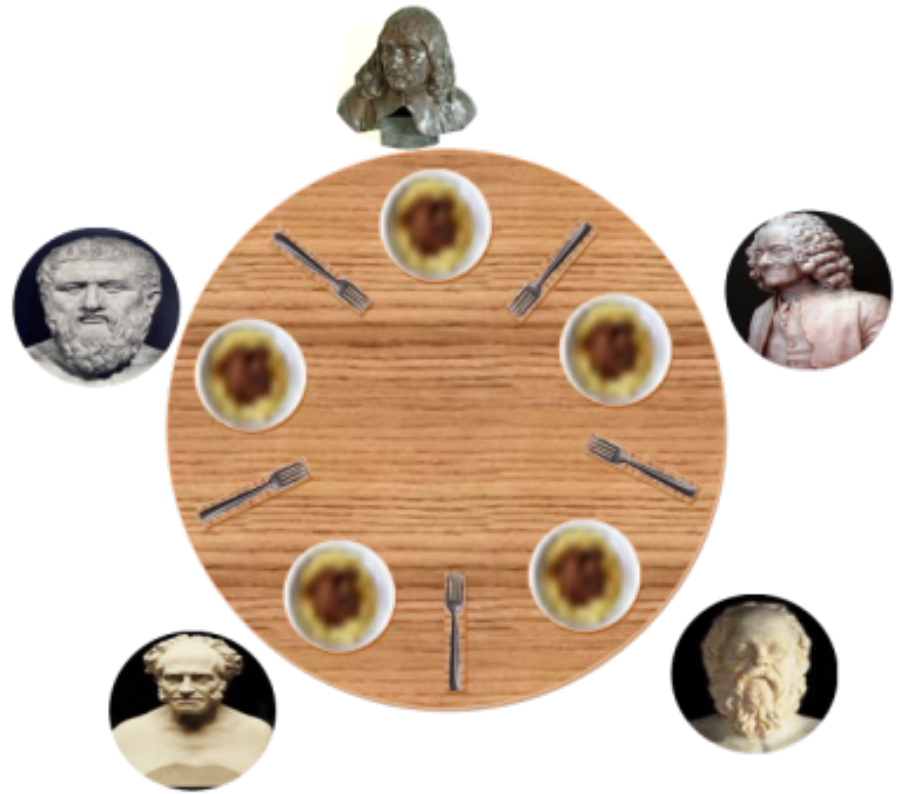
## Problema dei 5 Filosofi a Cena & Problema dei Lettori e Scrittori

C.d.L. in Informatica  
(laurea triennale)  
A.A. 2025-2026

**Prof. Mario F. Pavone**  
Dipartimento di Matematica e Informatica  
Università degli Studi di Catania  
[mario.pavone@unict.it](mailto:mario.pavone@unict.it)



# Problema dei 5 filosofi a cena

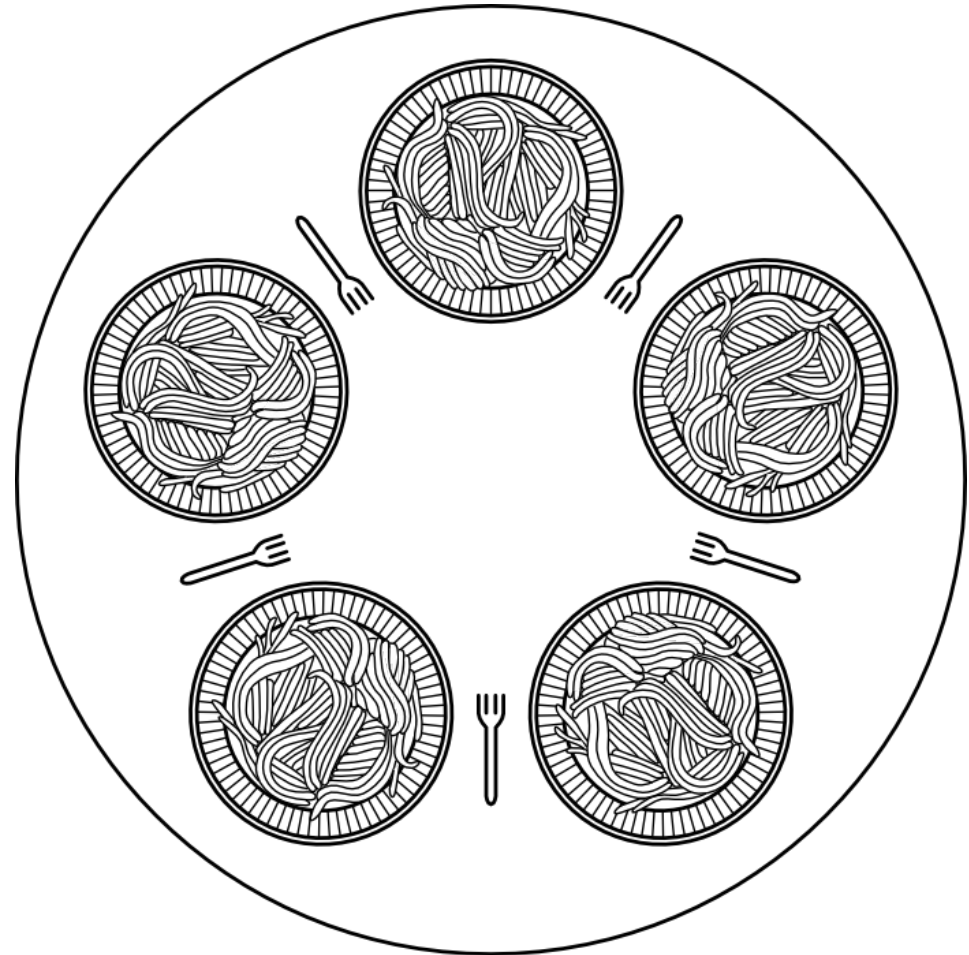


- Problema di **Sincronizzazione**
- Ogni filosofo: **pensa e mangia**
- Per mangiare **deve avere entrambe le forchette**: destra & sinistra
- **DOMANDA**: è possibile scrivere un programma per ogni filosofo in modo che non si blocchi mai?

# 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)
```

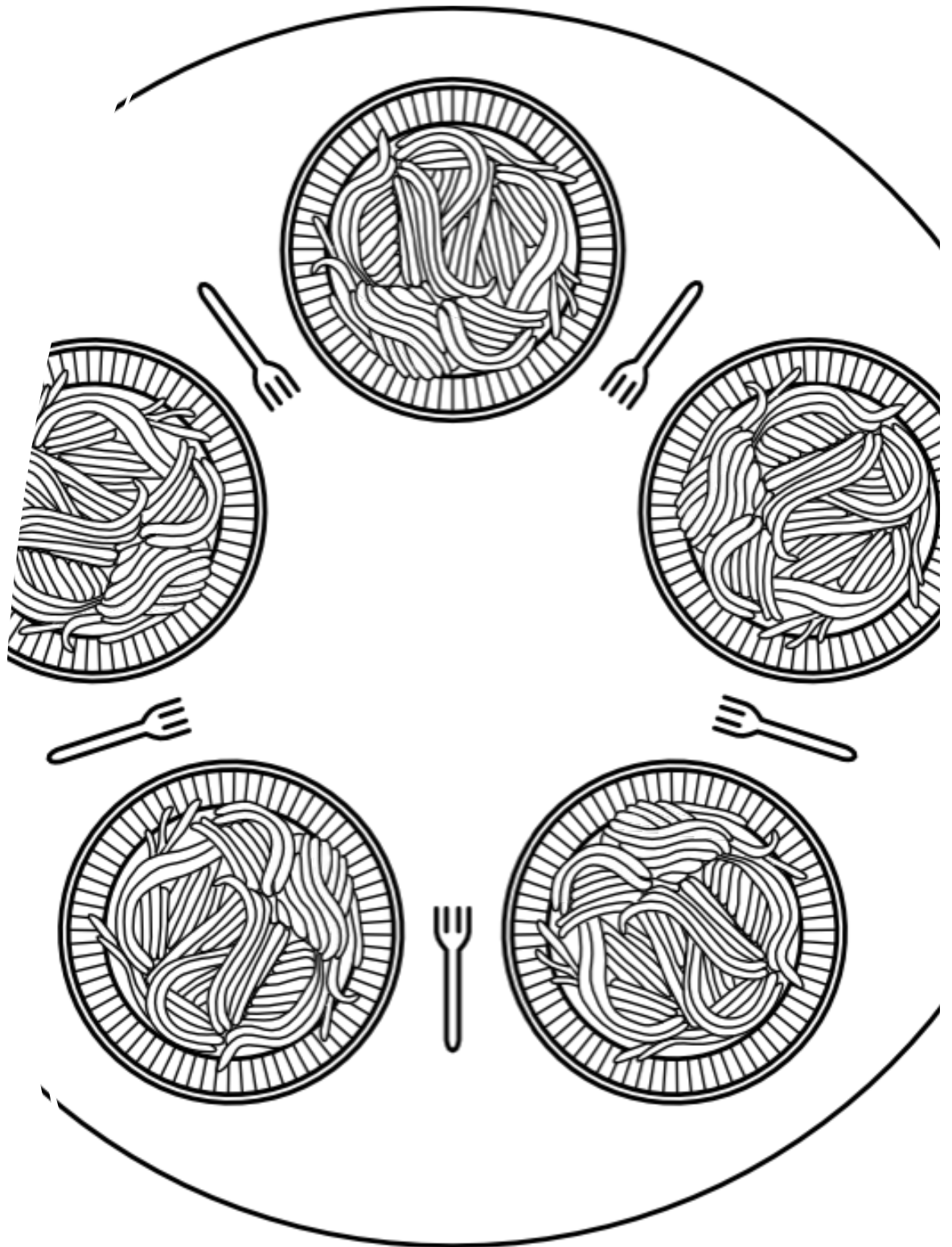




# Problema dei 5 filosofi

---

- ... altre soluzioni.....
- **soluzione 3**: controlla con **rilascio e riprova** aspettando un **tempo random**.
  - si vuole una funzione che funzioni sempre
- **soluzione 4**: utilizzo di un semaforo ***mutex***



# Problema dei 5 filosofi

- ... altre soluzioni.....

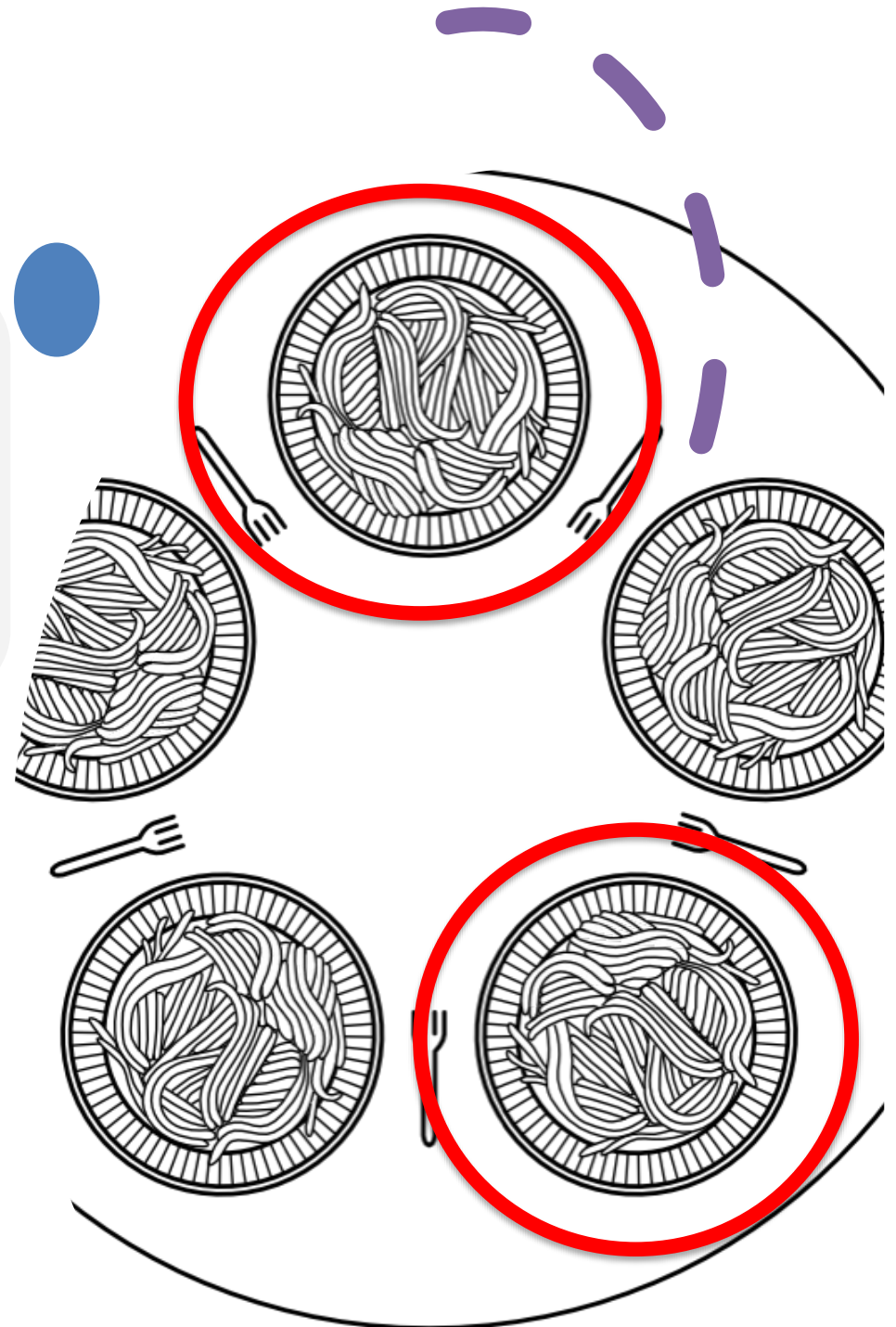
- **soluzione 3:** controlla con **rilascio e riprova** aspettando un **tempo random**.

- si vuole una funzione che funzioni sempre

- **soluzione 4:** utilizzo di un semaforo **mutex**

- Non del tutto efficiente

Con 5 forchette riescono a mangiare 2 filosofi contemporaneamente



# 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

- Problema classico che **modella l'accesso ad un data-base**
- Più processi leggono il DB => NO se si aggiorna il DB
- DOMANDA: **come programmare i lettori e gli scrittori?**





# Problema dei lettori e scrittori: soluzione basata sui semafori

```
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
- **Alternativa:** lo scrittore attende solo i lettori che lo precedono
  - **svantaggio:** minori prestazioni

# 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()
```



# Sistemi Operativi (M-Z)



ComplexIntelligentSystems@gmail.com  
<http://cis-lab.dmi.unict.it/>

## Scheduler & Algoritmi di Scheduling

Prof. Mario F. Pavone

Dipartimento di Matematica e Informatica  
Università degli Studi di Catania  
[mario.pavone@unict.it](mailto:mario.pavone@unict.it)

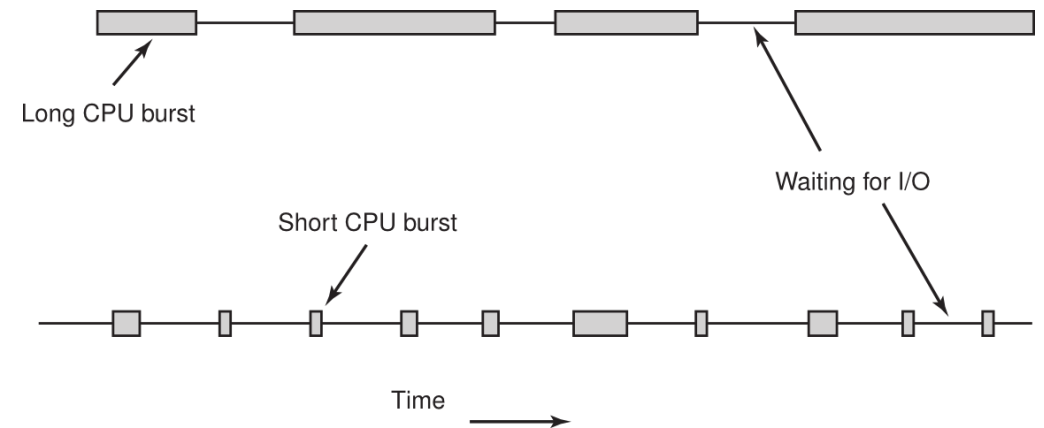
C.d.L. in Informatica (Laurea Triennale)

A.A. 2025-2026

# Scheduling

- Quale processo/thread eseguire successivamente?
- **Scheduler & Algoritmo di Scheduling**
- **Mainframe vs. PC vs. Server**
- Vecchi PC vs. Nuovi PC
  - risorse sono abbondanti
  - fortemente parallelizzate
- **Scheduler:**
  - effettua la scelta a secondo *tipo di macchina*
  - **Goal:** uso efficiente CPU

# Scheduling



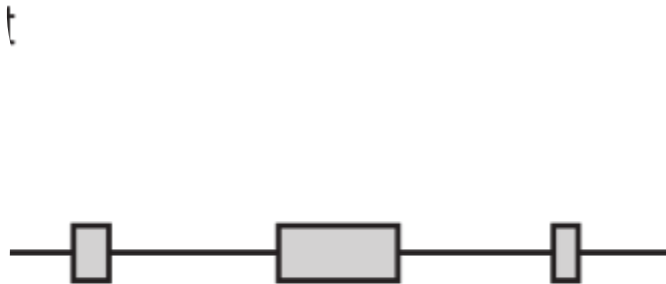
*CPU veloci  $\Rightarrow$  I/O bound*

- Esecuzione si alterna con richieste di I/O
- tipologie di processi:
  - **CPU-bounded**  $\Rightarrow$  burst CPU lunghi e attese I/O poco frequenti
  - **I/O-bounded**  $\Rightarrow$  burst CPU brevi e attese I/O frequenti
- Corretto bilanciamento delle attività Disco & CPU

# Scheduling



- **quando** viene attivato lo scheduler:
  1. **terminazione e creazione** di processi;
  2. **chiamata bloccante** (es., I/O) e arrivo del relativo interrupt;
  3. **blocco I/O** => può influire sulla scelta del successivo



- interrupt periodici:
  - sistemi **non-preemptive** (senza prelazione);
  - sistemi **preemptive** (con prelazione);
- collabora con il **dispatcher**:
  - **il più veloce possibile**
  - **latenza di dispatch.**





# Obiettivi degli algoritmi di scheduling

- Ambienti differenti:
  - **Batch**: non-preemptive
  - **Interattivi**: preemptive
  - **Real-Time**: preemptive non sempre necessaria
- **Obiettivi comuni**:
  - **equità nell'assegnazione della CPU**;
    - processi comparabili devono avere servizi comparabili
  - **bilanciamento nell'uso delle risorse**;
    - tutte le parti del sistema devono essere impegnate

# Obiettivi degli algoritmi di scheduling (metriche prestazioni)

---

NOTA: lo scheduler influisce solo sulle attese del processo  
NON per attesa I/O o tempo della CPU per eseguire il processo

- Obiettivi tipici dei **sistemi batch**:
  - massimizzare il **throughput** (o produttività);
  - minimizzare il **tempo di turnaround** (o tempo di completamento);
  - minimizzare il **tempo di attesa**;
- **Massimizzare throughput non necessariamente minimizza il tempo di turnaround**
- 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

In generale:  $(4a+3b+2c+d)/4$

## Esempio

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

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

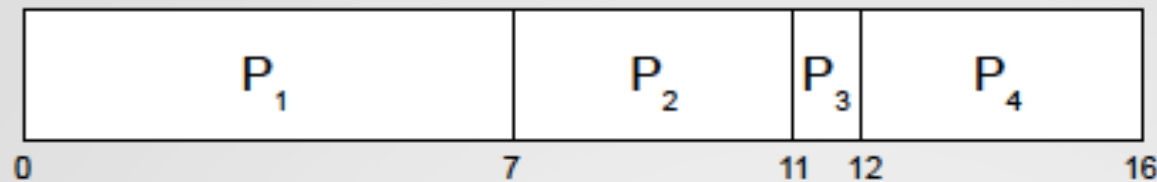
SJF  $(0+2+3+4+5)/5 = 2.8$   
altern.  $(7+0+1+2+3)/5 = 2.6$

| 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      |

## Scheduling nei sistemi batch

FCFS vs. SJF vs. SRTN

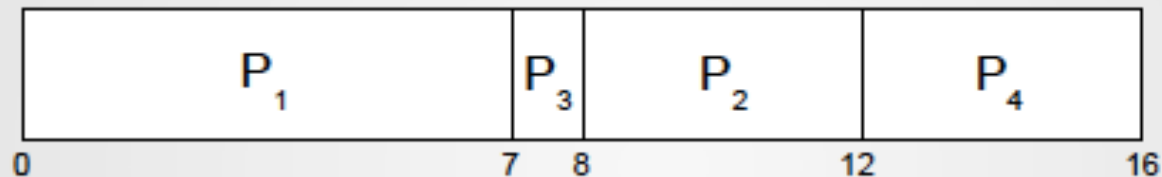
- **FCFS:**



- tempi di attesa:  $P_1=0$ ;  $P_2=5$ ;  $P_3=7$ ;  $P_4=7$  (media 4.75);
- tempi di completamento:  $P_1=7$ ;  $P_2=9$ ;  $P_3=8$ ;  $P_4=11$  (media 8.75);

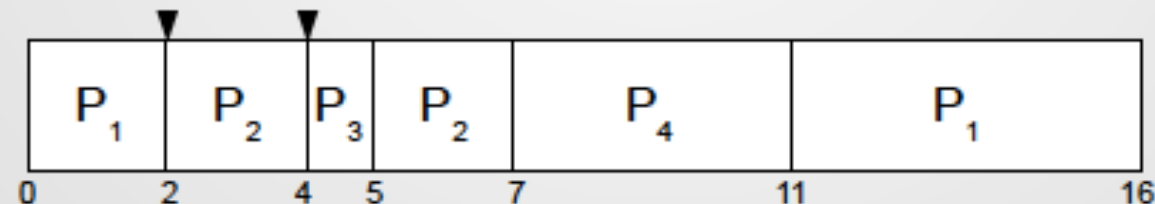
| Processo       | Arrivo | Durata |
|----------------|--------|--------|
| P <sub>1</sub> | 0      | 7      |
| P <sub>2</sub> | 2      | 4      |
| P <sub>3</sub> | 4      | 1      |
| P <sub>4</sub> | 5      | 4      |

- **SJF:**



- tempi di attesa:  $P_1=0$ ;  $P_2=6$ ;  $P_3=3$ ;  $P_4=7$  (media 4);
- tempi di completamento:  $P_1=7$ ;  $P_2=10$ ;  $P_3=4$ ;  $P_4=11$  (media 8);

- **SRTN:**

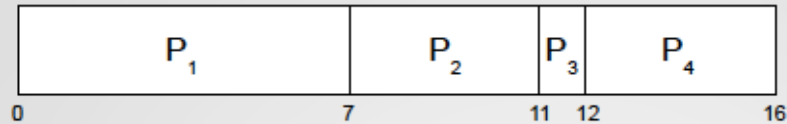


## Scheduling nei sistemi batch

FCFS vs. SJF vs. SRTN

# Diagrammi di Gantt

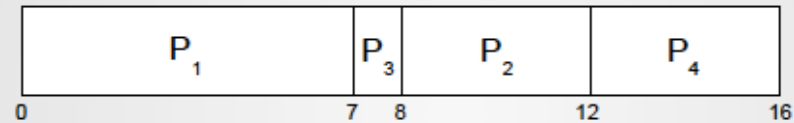
## FCFS:



- tempi di attesa:  $P_1=0$ ;  $P_2=5$ ;  $P_3=7$ ;  $P_4=7$  (media 4.75);
- tempi di completamento:  $P_1=7$ ;  $P_2=9$ ;  $P_3=8$ ;  $P_4=11$  (media 8.75);

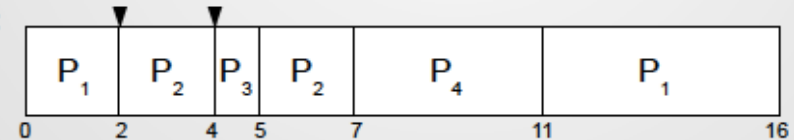
| Processo       | Arrivo | Durata |
|----------------|--------|--------|
| P <sub>1</sub> | 0      | 7      |
| P <sub>2</sub> | 2      | 4      |
| P <sub>3</sub> | 4      | 1      |
| P <sub>4</sub> | 5      | 4      |

## SJF:



- tempi di attesa:  $P_1=0$ ;  $P_2=6$ ;  $P_3=3$ ;  $P_4=7$  (media 4);
- tempi di completamento:  $P_1=7$ ;  $P_2=10$ ;  $P_3=4$ ;  $P_4=11$  (media 8);

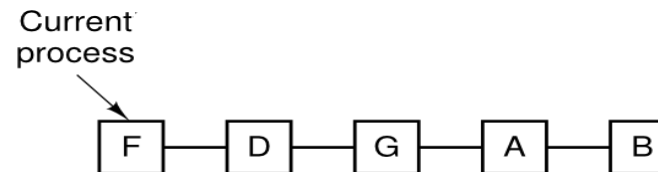
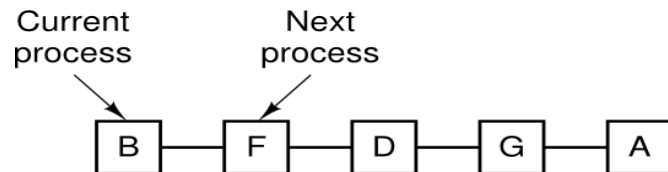
## SRTN:



- tempi di attesa:  $P_1=9$ ;  $P_2=1$ ;  $P_3=0$ ;  $P_4=2$  (media 3);
- tempi di completamento:  $P_1=16$ ;  $P_2=5$ ;  $P_3=1$ ;  $P_4=6$  (media 7).

# Scheduling nei sistemi interattivi

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



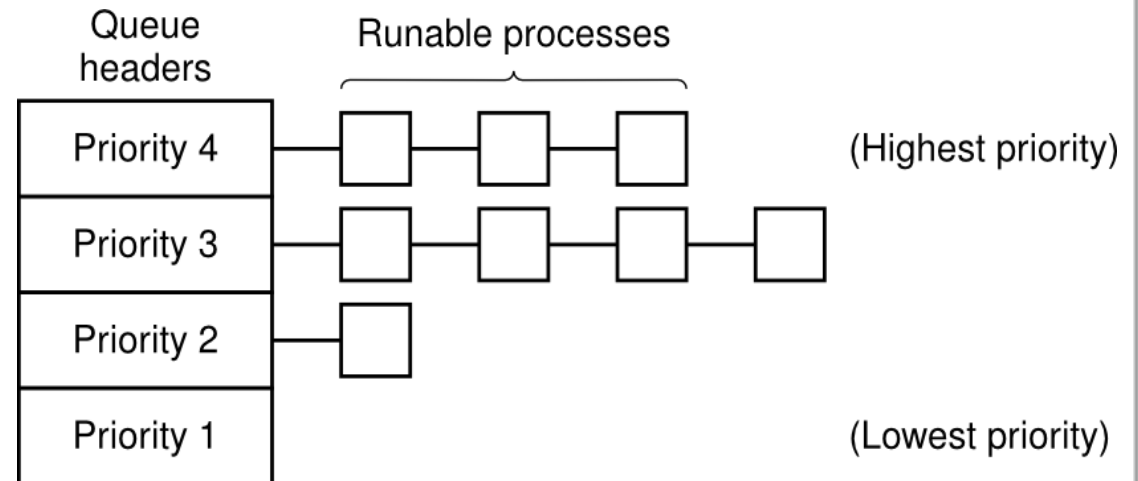
- quanto deve essere lungo il timeslice?
  - **process switch** or **context switch**
  - valori tipici sono 20-50ms;
- con  $n$  processi e un quanto di  $q$  **millisecondi**, ogni processo avrà diritto a circa  $1/n$  della CPU e attenderà al più  $(n-1)q$  ms

# Scheduling nei sistemi interattivi

- **Vincolo RR: tutti i processi con uguale importanza**
- **Scheduling a priorità:**
  - **regola di base:** si assegna la CPU al processo con più alta priorità;
  - **assegnamento delle priorità:**
    - **statiche** o **dinamiche**
      - comando *nice* in Unix
    - **favorire processi I/O bounded**
      - priorità  $1/f$ , con  $f$  = frazione quanto utilizzato
    - **SJF come sistema a priorità**
      - priority: inverse task length

# Scheduling nei sistemi interattivi

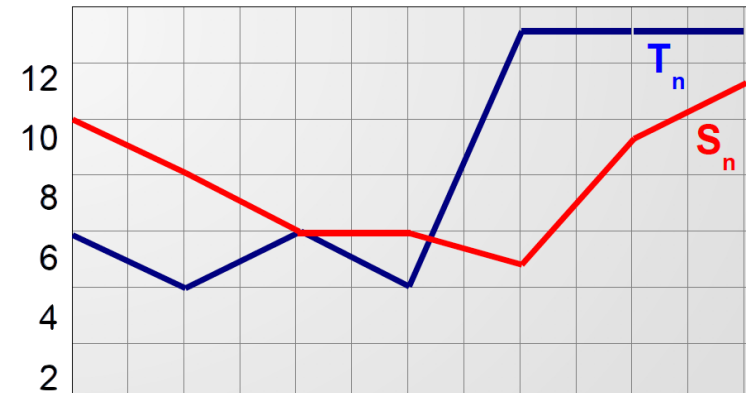
- **prelazione vs. non-prelazione** (ex. SJF vs. SRTN)
- **possibile problema:** **starvation**
- **possibile soluzione:** **aging**
  - **Incremento graduale della priorità**
- **Classi di Priorità**
  - **Scheduling di priorità tra le classi**
  - **Scheduling Round-Robin all'interno delle classi**



- **NOTA:** priorità non corrette => problemi con classi priorità basse



# 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;

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

$$0 \leq a \leq 1$$

- $a$  determina la stima lungo, media o corta

- 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:**
  - fare promesse reali e mantenerle
  - stabilire una **percentuale di utilizzo e rispettarla**.
  - $n$  utenti connessi avranno  $1/n$  della potenza CPU (idem processi)
  - tenere traccia quanta CPU ricevuta
  - calcolare quantità CPU spettante
    - $(T_c / n)$  = tempo dalla creazione diviso  $n$
  - l'algoritmo esegue il **processo con rapporto minore**

# Scheduling nei sistemi interattivi

- **Scheduling a lotteria:**

- idea base: **biglietti con estrazioni a random**
  - **non lo stesso numero di biglietti**
  - **processo da eseguire con estrazione**
- processi importanti: **biglietti extra**
- **criterio semplice e chiaro**
- reagisce velocemente ai cambiamenti
- possibilità di avere **processi cooperanti**
- **risolvere problemi difficili da gestire con altri metodi**

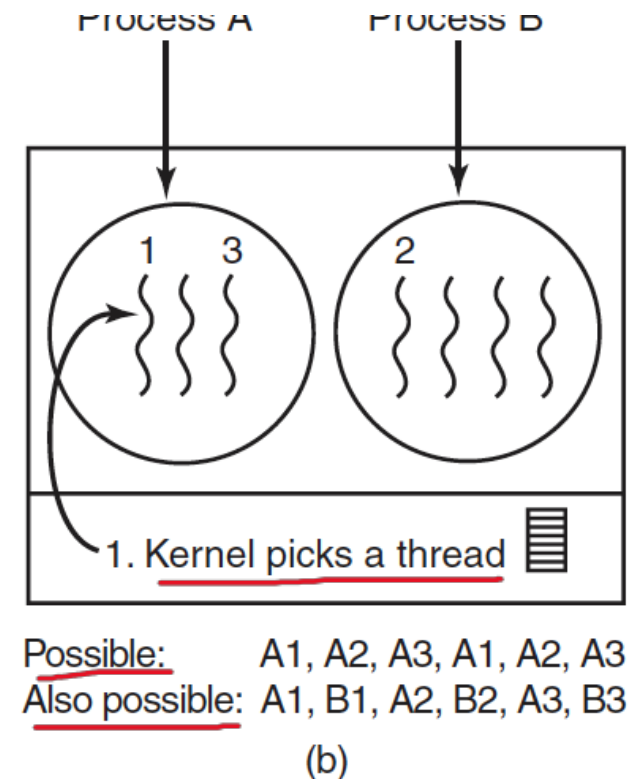
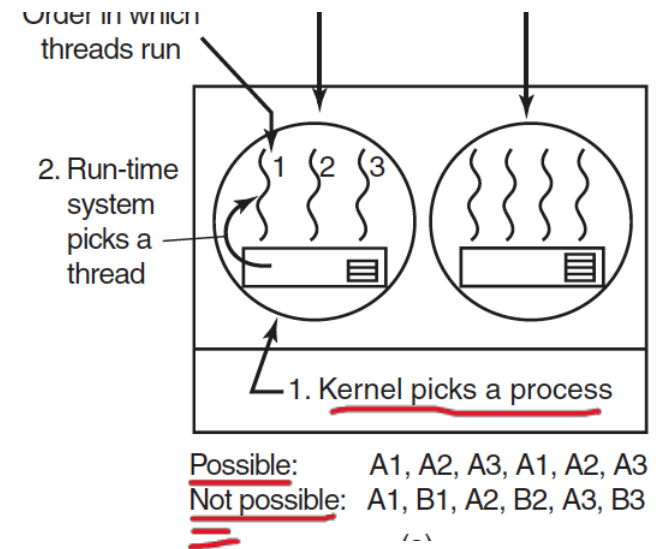


# Scheduling nei sistemi interattivi

- Scheduling fair-share:
  - considerare a chi appartiene un processo
  - realizza un equo uso tra gli utenti del sistema
  - un utente con più processi avrà più "attenzione"
  - considerare la proprietà dei processi durante la schedulazione
    - a prescindere dal numero di processi

# 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
    - unica restrizione: assenza interrupt clock
  - possibilità di utilizzo di scheduling personalizzato
- Thread del kernel:
  - o si considerano tutti i thread uguali, oppure;
  - si pesa l'appartenenza al processo;
    - switch su un thread di un processo diverso implica anche la riprogrammazione della MMU e, in alcuni casi, l'azzeramento della cache della CPU.



# Thread Utente vs. Thread Kernel (differenze)

- Prestazioni:
  - cambio di contesto più lento
  - livello kernel: thread bloccato non sospende il processo
    - no livello utente
  - informazione su processo proprietario thread
    - thread processo B più costoso secondo thread processo A
  - thread utente: scheduler specifico dell'applicazione.



# Scheduling su sistemi multiprocessore

- **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:
  - **Gestione della cache**
  - presenza o assenza di **predilezione per i processori**:
    - **predilezione debole**: supporta ma non garantisce
    - **predilezione forte**: forza un processo ad un dato processore



# Scheduling su sistemi multiprocessore

- Bilanciamento del carico:
  - Avvantaggiarsi di **avere più processori**
  - **Ugualmente distribuito**
  - necessaria solo in **presenza di code distinte** per i processi pronti;
  - **migrazione guidata** (*push migration*) o **migrazione spontanea** (*pull migration*);
    - 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 con varie;
  - 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.