



SAPIENZA  
UNIVERSITÀ DI ROMA

UNIVERSITÀ "SAPIENZA" DI ROMA  
FACOLTÀ DI INFORMATICA

---

## Sistemi Operativi

---

Appunti integrati con il libro "Modern Operating Systems",  
A. S. Tanenbaum

*Author*  
Simone Bianco

26 gennaio 2023

# Indice

<b>0 Introduzione</b>	<b>1</b>
<b>1 Introduzione ai Sistemi Operativi</b>	<b>2</b>
1.1 Sistema operativo e Macchina fisica . . . . .	5
1.2 Servizi del sistema operativo . . . . .	7
1.2.1 Kernel/User mode e Protezione della memoria . . . . .	7
1.2.2 System Calls . . . . .	8
1.2.3 Scheduling e Sincronizzazione . . . . .	10
1.2.4 Virtualizzazione della memoria . . . . .	10
<b>2 Gestione dei processi</b>	<b>11</b>
2.1 Stato dei processi e PCB . . . . .	14
2.2 Creazione dei processi . . . . .	15
2.3 Terminazione dei processi . . . . .	20
2.4 Scheduling dei processi . . . . .	20
2.5 Comunicazione tra processi . . . . .	21
2.6 Scheduler della CPU . . . . .	23
2.6.1 First Come First Served (FCFS) . . . . .	26
2.6.2 Round Robin (RR) . . . . .	30
2.6.3 Shortest Job First (SJF) . . . . .	33
2.6.4 Priority Scheduling . . . . .	37
2.6.5 Multilevel Queueing (MLQ e MLFQ) . . . . .	38
2.6.6 Lottery scheduling . . . . .	41
2.7 Thread e Multi-threading . . . . .	41
2.7.1 Modelli di multi-threading . . . . .	45
<b>3 Sincronizzazione tra Processi/Thread</b>	<b>48</b>
3.0.1 Lock . . . . .	49
3.0.2 Semafori . . . . .	53

# Capitolo 0

## Introduzione

# Capitolo 1

## Introduzione ai Sistemi Operativi

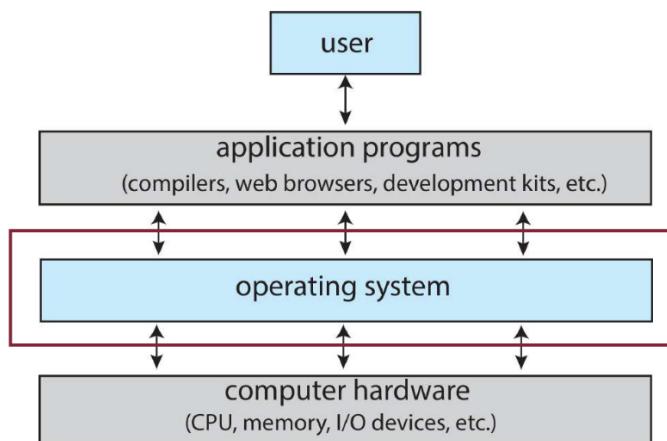
Un **sistema operativo**, abbreviato con OS, è un software di base in grado di gestire le risorse hardware e software della macchina, fornendo servizi di base ai software applicativi.

In particolare, a seconda delle necessità (ad esempio general-purpose, real-time, mobile, ...) e delle scelte di progettazione vengono decisi i compiti e i componenti che costituiscono un sistema operativo. Nonostante ogni OS tenda ad essere diverso dagli altri, ognuno di essi può essere suddiviso in:

- **Kernel**, ossia il "cuore" del sistema operativo, il quale è sempre attivo
- **Programmi di sistema**, ossia ogni altro software facente parte del sistema operativo

Tra i vari compiti svolti da un OS, in particolare troviamo:

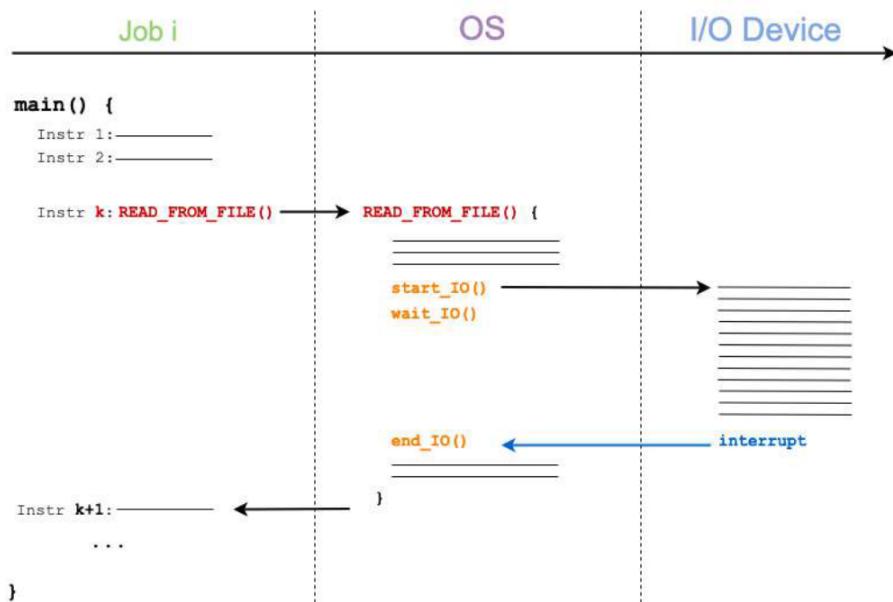
- **Gestione delle risorse fisiche e logiche** utilizzate dai vari software
- **Virtualizzazione delle risorse**, dando l'"illusione" ad ogni software di aver a disposizione "infinite risorse"
- **Interfacciamento tra Hardware e Software**, fornendo un insieme di servizi comuni, ossia delle API, che permettono ai software e agli utenti di interagire con le risorse hardware senza doverle controllare in modo diretto



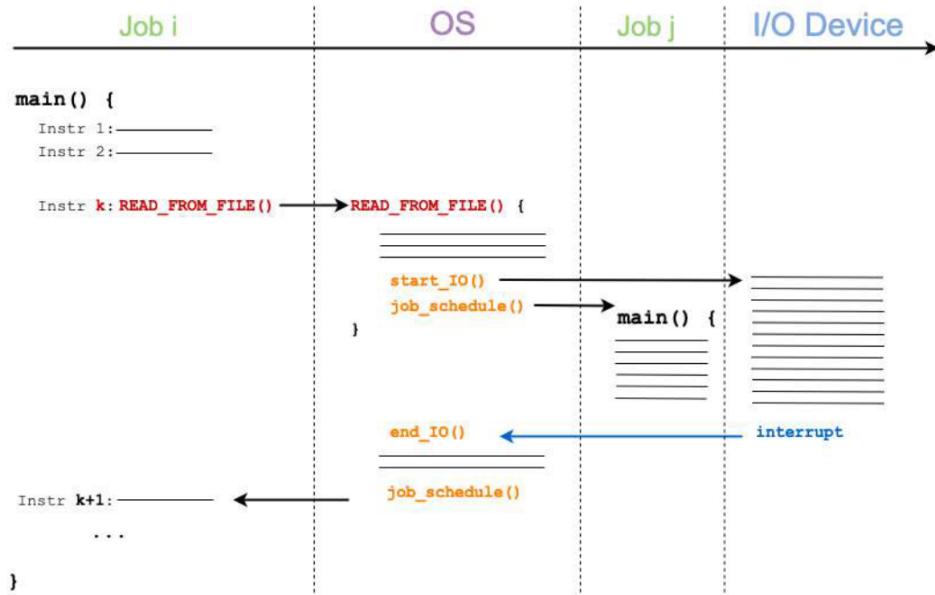
## Tipologie di OS

- **Sistemi programmati a mano**, utilizzati da una cerchia ristretta di persone esperte in grado di programmarli manualmente in linguaggio macchina, dunque sprovvisti di sistema operativo agevolante. Il problema principale risultò essere il poco utilizzo e l'eccessivo costo dell'hardware
- **Sistemi con console a singolo utente (Mainframe)**, utilizzati da un utente per volta durante l'esecuzione di un programma, i quali venivano scritti su schede perforate. Ogni operazione veniva eseguita sequenzialmente, prevedendo la presenza di una versione primitiva di OS, ossia un avviatore di programmi, e l'assenza di sovrapposizione tra computazione e interfacciamento con le periferiche di I/O. Per via della sua natura single-user oriented, il problema principale risultò essere l'inefficienza per più utenti.
- **Sistemi di Batch**, in grado di eseguire più "compiti", detti **jobs**, in gruppo, detti **batch**. Ogni utente inseriva un programma tramite schede o nastri, i quali venivano poi periodicamente caricati da un tecnico, per poi essere avviati ed eseguiti da un sistema operativo, risultando in un uso più efficiente dei calcolatori. Tuttavia, ogni job veniva ancora eseguito in modo sequenziale, ossia uno per volta.
- **Sistemi multi-programmabili**, in grado di mantenere caricati in memoria più job contemporaneamente, i quali vengono eseguiti da una CPU in grado di alternare tra di essi. In questi sistemi, l'OS si occupava dell'organizzazione dell'esecuzione dei programmi, detto **job scheduling**, delle operazioni di I/O e della protezione della memoria, impedendo ai vari job di accedere ad aree utilizzate da altri. L'unica problematica di tali sistemi risultò essere la necessità di mantenere la CPU inattiva durante l'esecuzione di operazioni di I/O bloccanti.

### Esempio di Sistema Bloccante:

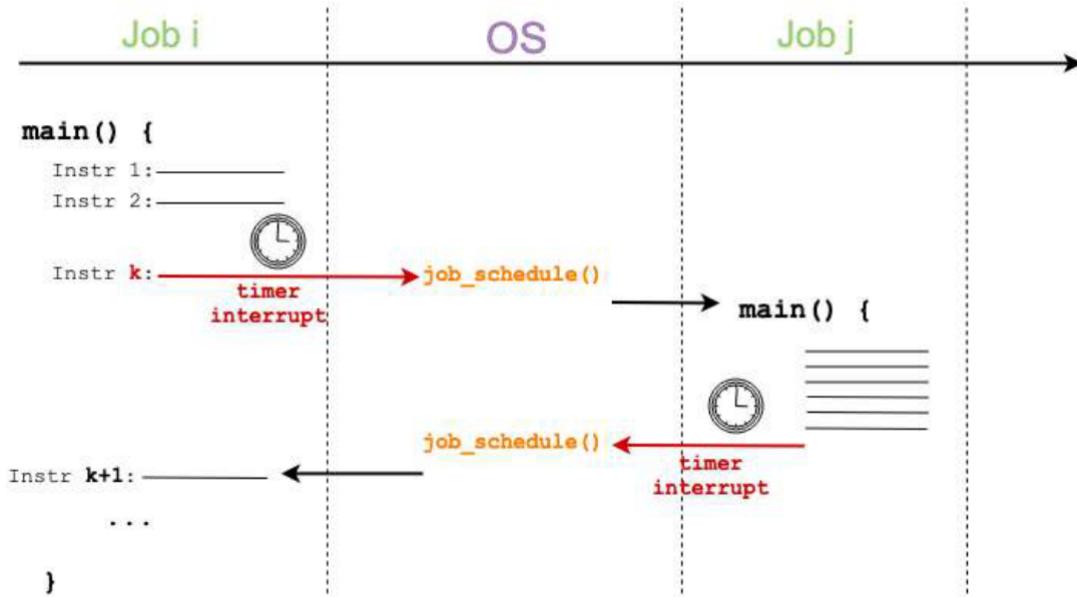


### Esempio di Sistema Non-bloccante:



- **Sistemi time-sharing**, dove più utenti utilizzano la stessa CPU tramite console semplici. Viene utilizzato un'interruzione programmata a tempo per alternare la CPU tra i vari job, dando l'illusione di esecuzione parallela (pseudo-parallelismo). In particolare, è famoso il sistema operativo ideato da Ken Thompson e Dennis Ritchie, ossia UNIX, precursore di sistemi operativi come MacOS e le varie distribuzioni di Linux

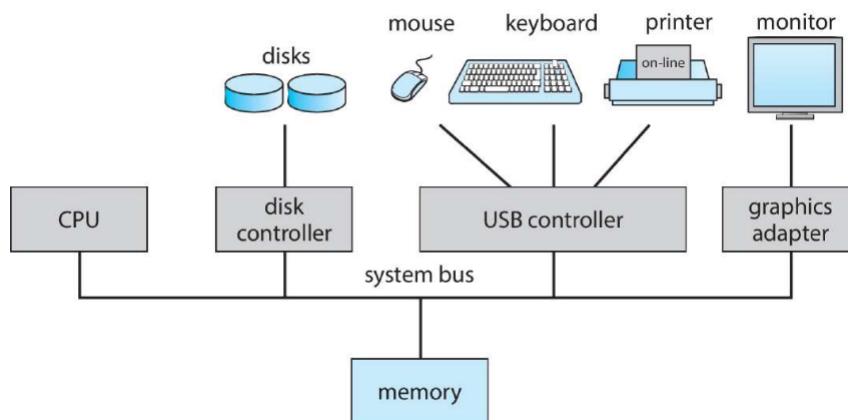
### Esempio di Pseudo-parallelismo:



## 1.1 Sistema operativo e Macchina fisica

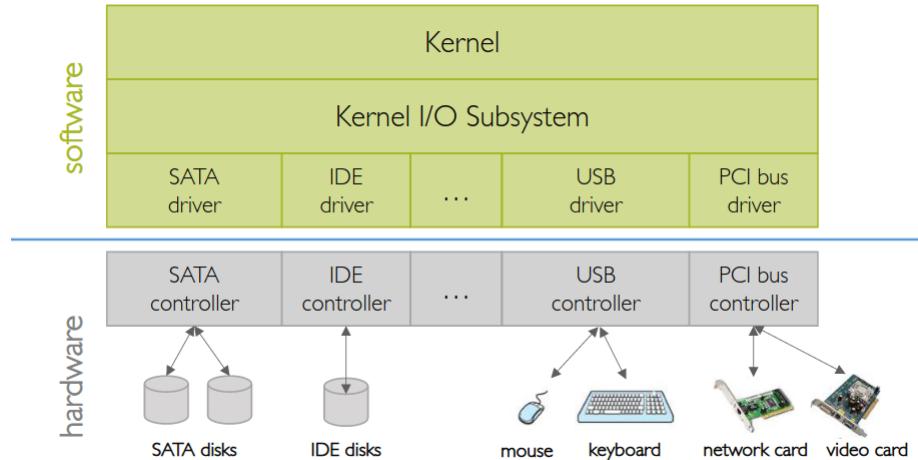
Una moderna architettura è *necessariamente*, composta da:

- **CPU**, il processore che effettua i veri calcoli necessari, operando in base ad un **ciclo di istruzioni**, ossia tre fasi ripetute ciclicamente a velocità elevate:
  1. Fase di **Fetch**: viene prelevata dalla memoria la prossima istruzione da eseguire
  2. Fase di **Decode**: viene interpretata l'istruzione appena recuperata
  3. Fase di **Execute**: viene eseguita l'istruzione appena interpretata
- **Memoria principale**, contiene i dati e le istruzioni utilizzati ed eseguite dalla CPU, la quale può essere immaginata come una sequenza di celle, ognuna composta da una **word**, ossia una sequenza di bit (solitamente 32 o 64), ed ognuna avente un proprio **indirizzo**
- **Dispositivi di I/O**, ossia le periferiche di input ed output (tastiera, mouse, memorie esterne, ...)
- **Bus di Sistema**, ossia un mezzo di comunicazione tra CPU, memoria e dispositivi di I/O, composto a sua volta da:
  - **Data Bus**, il quale trasporta i dati effettivi
  - **Address Bus**, il quale determina dove i dati debbano essere inviati
  - **Control Bus**, il quale indica quale operazione deve essere effettuata



Le **istruzioni** eseguite dalla CPU vengono descritte tramite il **linguaggio macchina**, rappresentato da sequenze di bit, dove ogni word corrisponde ad una singola istruzione appartenente ad un insieme di istruzioni elementari, le quali interagiscono con i vari **componenti di calcolo**, in particolare l'**ALU**, e i **registri** della CPU, in grado di immagazzinare temporaneamente una word.

Riguardo ai dispositivi di I/O, è necessario sottolineare come ognuno di essi sia composto da due parti, ossia il **dispositivo fisico** in se e il **device controller**, un insieme di piccoli chip che lo gestisce. Il sistema operativo interagisce con tali dispositivi tramite dei **driver**, ossia dei particolari software adattati per ogni dispositivo che permettono di "tradurre" le richieste dell'OS.



Ogni device controller possiede una serie di registri dedicati con cui comunicare:

- **Registri di stato**, i quali forniscono informazioni alla CPU riguardo il dispositivo
- **Registri di controllo**, i quali vengono utilizzati dalla CPU per configurare e controllare il dispositivo
- **Registri dati**, i quali vengono utilizzati per lo scambio di dati con il dispositivo

Per poter comunicare con i dispositivi di I/O, la CPU deve essere in grado di poter distinguere tra indirizzi della memoria principale ed indirizzi rappresentanti i dispositivi stessi. A tal fine, il control bus è dotato di una **linea speciale** detta **M#IO** in grado di stabilire se la CPU desideri comunicare con la memoria principale o con i dispositivi di I/O.

Quindi, esistono in particolare due modi con cui la CPU può comunicare con i device controller:

- **Port-mapped I/O**, dove i riferimenti ai controller vengono effettuati utilizzando uno spazio di indirizzi aggiuntivo dedicato solo all'accesso ai dispositivi di I/O, necessitando l'implementazione di istruzioni aggiuntive del tipo IN ed OUT che vadano a specificare se lavorare sullo spazio di indirizzi della memoria e dei dispositivi I/O

```
MOV DX, 1234h //carica nel registro DX il valore 0x1234
```

```
MOV AL, [DX] //carica nel registro DX il valore contenuto  
//nell'indirizzo 0x1234 della memoria
```

```
IN AL, DX //carica nel register DX il valore letto dal  
//dispositivo connesso alla porta 0x1234
```

- **Memory-mapped I/O**, dove parte degli indirizzi della memoria viene "sprecato" per poter essere utilizzato per effettuare i riferimenti ai dispositivi I/O, non necessitando tuttavia istruzioni aggiuntive e permettendo alla CPU di trattare i registri dei control device come se fosse dei suoi registri aggiuntivi

## 1.2 Servizi del sistema operativo

### 1.2.1 Kernel/User mode e Protezione della memoria

Alcune istruzioni eseguite dalla CPU risultano essere più sensibili di altre, ad esempio l'istruzione HLT, in grado di arrestare il sistema, e l'istruzione INT X, in grado di generare un interrupt di sistema. Affinché tali **istruzioni privilegiate** vengano utilizzate esclusivamente dal sistema operativo, la CPU può essere impostata in due modalità specifiche a seconda del programma in esecuzione.

#### Definition 1. Kernel mode e User mode

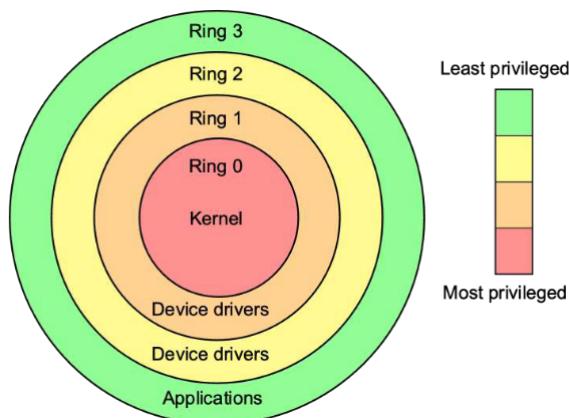
La CPU può essere impostata in:

- **Kernel mode**, ossia in modalità senza alcuna restrizione, permettendo l'esecuzione di qualsiasi istruzione (utilizzata dal sistema operativo)
- **User mode**, dove **non** è possibile:
  - Accedere agli indirizzi riservati ai dispositivi di I/O
  - Manipolare il contenuto della memoria principale
  - Arrestare il sistema
  - Passare alla Kernel Mode
  - ...

Per poter impostare una delle due modalità, viene utilizzato un **bit speciale** salvato in un registro protetto: se impostato su 0 la CPU sarà in Kernel mode, mentre se impostato su 1 la CPU sarà in User mode

#### Observation 1. Protection rings

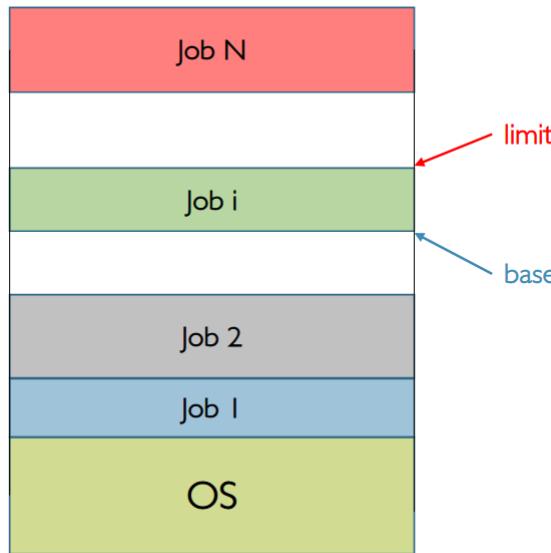
Nelle architetture più moderne, vengono implementati più livelli di protezione, detti **protection rings**. Ogni ring aggiunge restrizioni sulle istruzioni eseguibili dalla CPU.



Oltre alle protezioni imposte sulle istruzioni eseguite, è necessario imporre delle protezioni anche sulla **memoria**, in modo da proteggere reciprocamente gli spazi di memoria riservati ai singoli programmi degli utenti e in modo da proteggere gli spazi di memoria riservati all'OS dai programmi utente.

La tecnica più semplice per ottenere tale protezione è l'utilizzo di due registri dedicati, i quali vengono caricati dall'OS all'avvio di un programma l'OS, per poi controllare che ogni indirizzo richiesto durante l'esecuzione del programma ricada nell'intervallo specificato

- **Registro base**, contenente l'indirizzo iniziale dello spazio utilizzabile
- **Registro limite**, contenente l'indirizzo finale dello spazio utilizzabile



## 1.2.2 System Calls

### Definition 2. Trap di sistema

Definiamo come **trap di sistema** un qualsiasi evento che richiede il passaggio da Kernel mode ad User mode.

In particolare, individuiamo tre tipi di trap:

- **System calls**, ossia la richiesta di un servizio dell'OS, svolte in modo sincrono e innescate dai software
- **Exception**, ossia la gestione di errori dovuti ad eventi inattesi, svolte in modo sincrono e innescate dai software
- **Interrupt**, ossia il completamento di una richiesta in attesa, svolte in modo asincrono e innescate dall'hardware

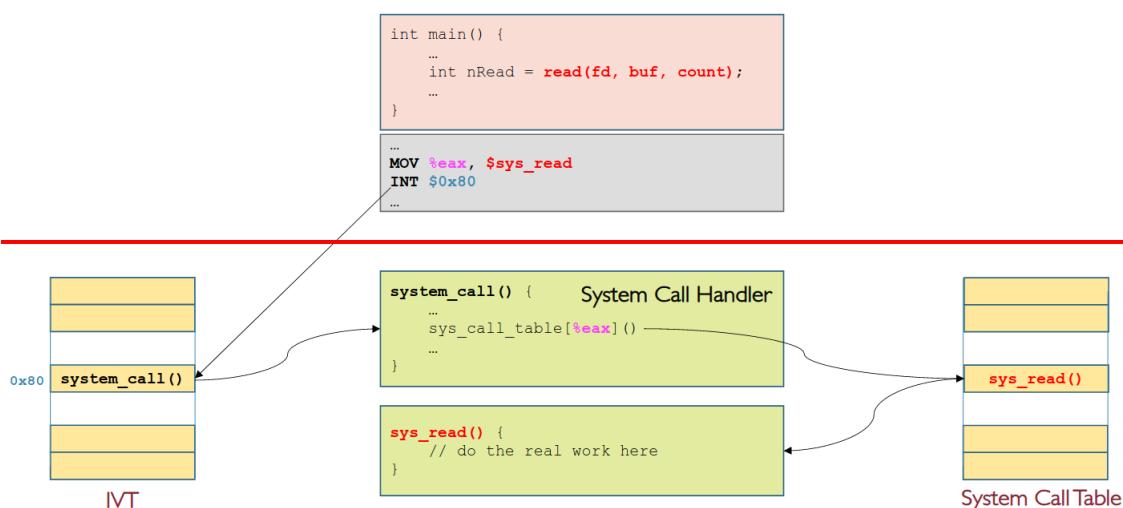
### Definition 3. System Calls

Le **System calls** sono delle procedure messe a disposizione dal sistema operativo per permettere agli altri software di poter accedere ai servizi messi a disposizione dal sistema operativo, in particolare l'esecuzione di alcune istruzioni privilegiate (ad esempio l'accesso ad un dispositivo I/O).

Le System call ricadono in **sei categorie**:

- **Process control**, include procedure come end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, allocate and free memory
- **File management**, include procedure come create file, delete file, open, close, read, write, reposition, get and set file attributes
- **Device management**, include procedure come request device, release device, read, write, reposition, get and set device attributes, logically attach or detach devices
- **Information maintenance**, include procedure come get and set time, get and set date, get and set system data, get and set process, get and set file information, get and set device attributes
- **Communications**, include procedure come create and delete communication connection, send and receive messages, transfer status information, attach or detach remote devices

Quando un programma utente richiede l'esecuzione di una syscall tramite un'API fornita dal sistema operativo, tale richiesta viene prima convertita in linguaggio macchina, per poi venir gestita da **System call Handler**, il quale si occuperà di salvare lo stato precedente dei registri, i quali verranno poi alterati durante l'esecuzione della syscall, per poi essere ripristinati



### 1.2.3 Scheduling e Sincronizzazione

Per poter gestire lo scheduling delle operazioni, viene implementato un **timer** nell'hardware, il quale genera un interrupt dopo una breve quantità di tempo, permettendo al **CPU scheduler** di prendere il controllo e decidere quale sia la prossima operazione da eseguire, impedendo che la CPU venga monopolizzata da un job "egoista"

Inoltre, poiché le interrupt sono **asincrone**, esse potrebbero essere innescate in qualsiasi momento, interferendo con i programmi in esecuzione. Il sistema operativo, quindi, deve essere in grado di **sincronizzare le operazioni** di processi concorrenti e cooperanti. Affinché ciò sia possibile, è necessario che l'hardware si assicuri che piccole sequenze di istruzioni vengano eseguite in modo **atomico**, disabilitando le interruzioni durante la loro esecuzione

### 1.2.4 Virtualizzazione della memoria

La **virtualizzazione della memoria** è un'astrazione logica della memoria fisica, dando ad ogni processo l'illusione di una memoria fisica strutturata come uno spazio di indirizzi contiguo, permettendo inoltre l'esecuzione di programmi senza la necessità che essi siano completamente caricati nella vera memoria principale (restando ovviamente caricati completamente nella memoria virtuale).

Può essere implementata sia **tramite hardware**, in particolare tramite una Memory Management Unit (MMU), sia **tramite software**, dove l'OS stesso si occupa di tradurre gli indirizzi virtuali in quelli fisici.

# Capitolo 2

## Gestione dei processi

### Definition 4. Programmi e Processi

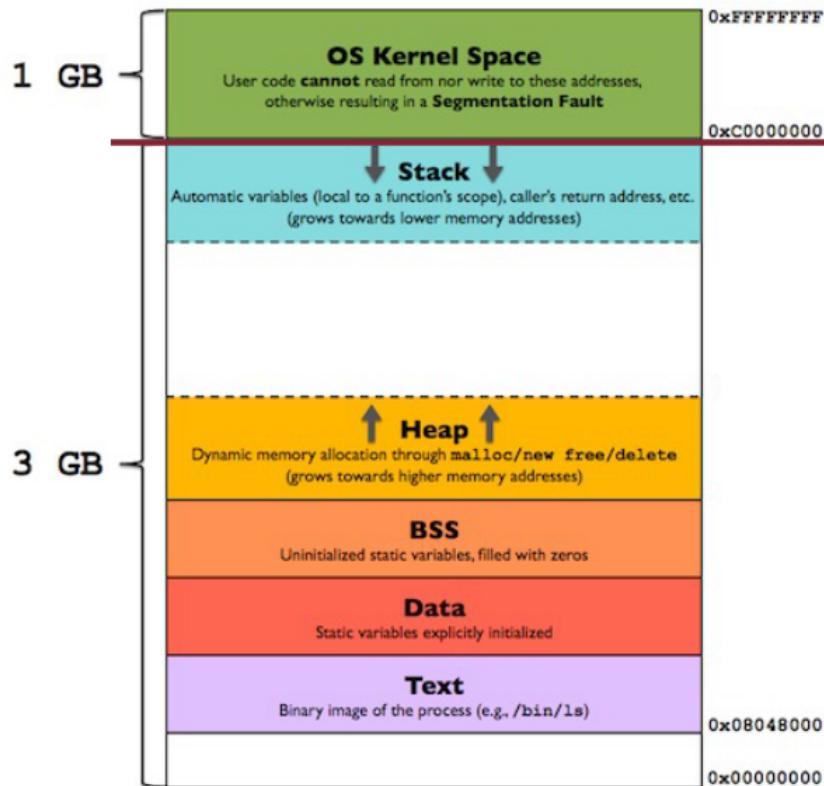
Un **programma** è un file eseguibile salvato in memoria secondaria. Contiene l'insieme di istruzioni necessarie a svolgere un compito richiesto.

Un **processo** è una particolare istanza attiva di un programma caricata in memoria principale, eseguendo sequenzialmente le istruzioni descritte nel programma stesso. Più processi potrebbero corrispondere allo stesso programma, tuttavia ognuno di essi possiede un proprio stato attuale.

Il sistema operativo fornisce la stessa quantità di **virtual address space** ad ogni processo, la quale è suddivisa in più sezioni:

- **Text**, contenente le istruzioni dell'eseguibile
- **Data**, contenente i dati statici già inizializzati all'avvio
- **BSS**, contenente i dati statici non inizializzati all'avvio
- **Stack**, contenente tutti i dati utilizzati da ogni funzione, ossia i vari **stack frame**
- **Heap**, utilizzato per allocazioni dinamiche, ossia di dimensione non fissa
- **Spazio libero**, interposto tra stack ed heap, utilizzato da entrambi per "espandersi"

Di seguito vi è un esempio di suddivisione del virtual address space di un'architettura a 32bit:



### Esempio:

- Consideriamo il seguente programma:

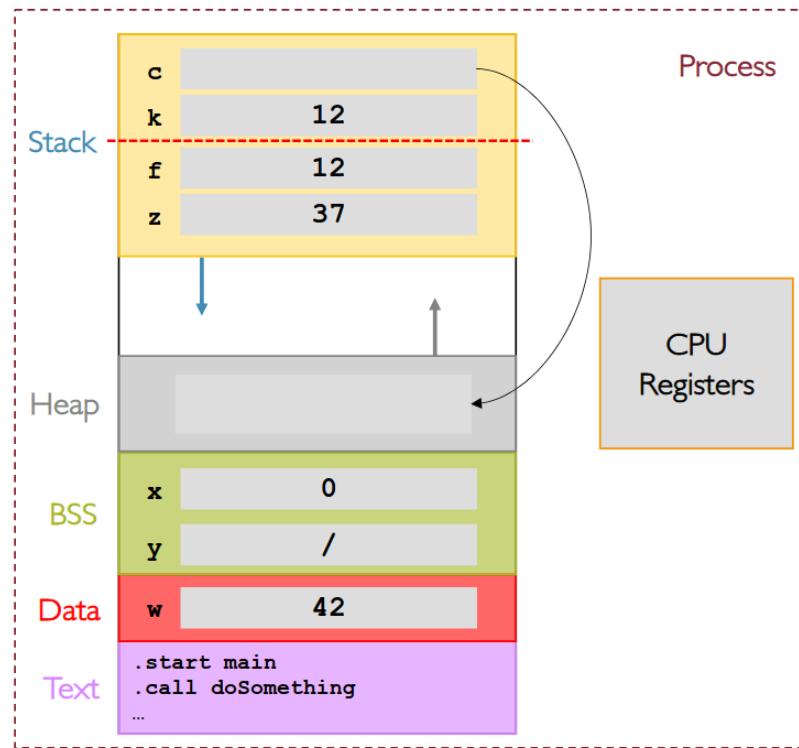
```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    ...
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    ...
}
```

- La variabile w sarà salvata all'interno della sezione Data, poiché essa è statica ed è già definita all'avvio

- Le variabili `x` e `y` verranno salvate all'interno della sezione BSS, poiché entrambe statiche ma non definite all'avvio (di default, il valore di una variabile di tipo int è già 0, dunque l'assegnamento iniziale inserito verrebbe semplicemente ignorato dal compilatore)
- Le istruzioni presenti nelle funzioni `main()` e `doSomething()` vengono inserite nella sezione Text
- Gli stack frame delle funzioni `main()` e `doSomething()` vengono salvati nello Stack.
- In particolare, la variabile `char* c` è un puntatore salvato nello Stack facente riferimento ad una zona dell'Heap di 128 bit, corrispondente quindi a 16 caratteri.

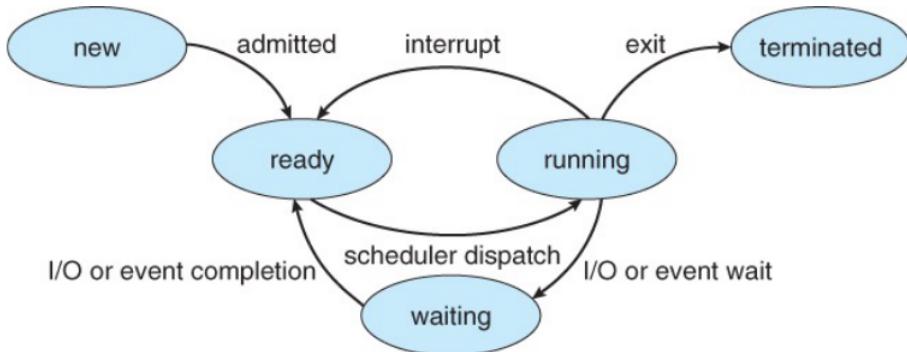


## 2.1 Stato dei processi e PCB

### Definition 5. Stato di un processo

In qualsiasi momento, uno processo può assumere uno dei seguenti stati:

- **New**, ossia il processo è appena stato avviato
- **Ready**, il processo è pronto ad essere eseguito ma è in attesa di essere schedulato nella CPU
- **Running**, le istruzioni del processo stanno venendo eseguite dalla CPU
- **Waiting**, il processo viene sospeso in attesa che una risorsa necessaria sia disponibile o che un evento sia completato o avvenga (ad esempio input da tastiera, accesso al disco, ...)
- **Terminated**, il processo ha completato la sua esecuzione e l'OS può eliminarlo



### Observation 2

La maggior parte delle syscalls è **bloccante**:

- Dal lato **user space**, il processo chiamante non può svolgere operazioni finché la syscall termina la richiesta
- Dal lato **kernel space**, l'OS imposta il processo corrente nello stato waiting e schedula il primo processo in stato di ready presente nella **state queue**
- Una volta che la syscall termina, il precedente processo bloccato passa in stato di ready, rimanendo in attesa di essere schedulato nuovamente

NB: solo il processo che ha effettuato la chiamata viene bloccato, non l'intero sistema operativo

**Definition 6. Process Control Block (PCB)**

Il **Process Control Block (PCB)** è la struttura dati principale utilizzata dall'OS per tenere traccia dello stato e della posizione in memoria di un processo. All'avvio di un processo, l'OS alloca un nuovo PCB, aggiungendolo alla state queue, per poi de-allocarlo una volta che il processo associato è terminato.

Ogni PCB è composto da:

- **Process state**, contenente lo stato del processo
- **Process number**, contenente un'identificatore univoco per il processo
- **Program counter, Stack Pointer e registri vari** relativi al processo in esecuzione
- **Informazioni per lo scheduling**, contenente l'indice di priorità e il puntatore alla state queue
- **Informazioni per la gestione della memoria e contabilità**, contenenti le page table relative al processo e il tempo in user e kernel mode del processo
- **Status I/O**, contenente la lista dei file aperti dal processo

## 2.2 Creazione dei processi

**Definition 7. Processo padre e figlio**

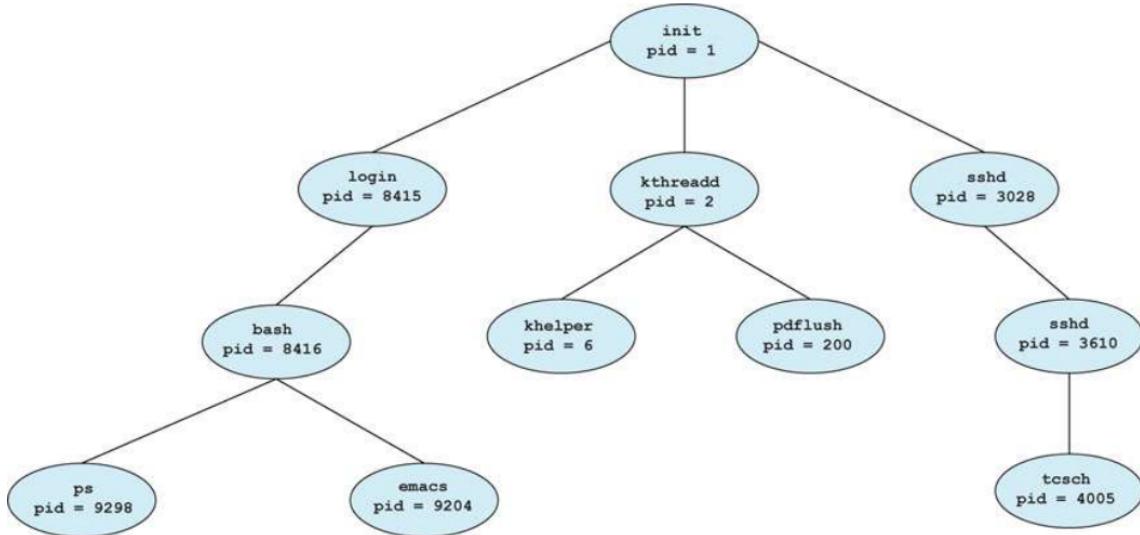
Un processo può creare altri processi tramite specifiche syscalls. Il processo creatore viene detto **processo padre**, mentre il processo creato viene detto **processo figlio**.

Ad ogni processo è associato un **process identifier (PID)**, che lo identifica univocamente, ed un **parent process identifier (PPID)**, che identifica univocamente il suo processo padre.

**Definition 8. Processo sched e init**

In un tipico sistema UNIX, il processo scheduler è chiamato **sched** e il suo PID è 0. La prima operazione eseguita dal processo **sched** all'avvio del sistema operativo è avviare il processo **init**, avente PID pari a 1. Il processo **init** si occupa di avviare tutti i **processi daemon**, ossia processi eseguiti in background, e i processi relativi ai login degli utenti, diventando quindi l'antenato di tutti i processi che verranno avviati in seguito.

Esempio:



### Method 1. Fork e Spawn

Per poter creare i processi figli relativi ad un parente vengono utilizzate due syscall:

- **fork()**, dove il figlio creato è una copia esatta del padre, condividendo con esso le stesse risorse ed ognuno avente il proprio PCB
- **exec()**, dove il figlio creato è un processo legato ad un programma diverso da quello del padre e avente uno spazio d'indirizzamento diverso, dunque con istruzioni e dati diversi dal padre

Poiché utilizzando la syscall **fork()** i due processi condividono lo stesso codice e le stesse risorse, nel processo **padre** essa **ritornerà un PID maggiore di 0**, corrispondente al PID effettivo del figlio, mentre nel processo **figlio**, essa **ritornerà 0 come PID**, corrispondente al PID di **sched**

Una volta creato un processo figlio, il processo padre ha due opzioni:

- Attendere che il processo figlio termini l'esecuzione utilizzando la syscall **wait()**
- Continuare la sua esecuzione in parallelo con il figlio senza essere bloccato

**Esempi:**

- Consideriamo il seguente codice:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(){
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) {
        /* if the returned PID is < 0, an error occurred */

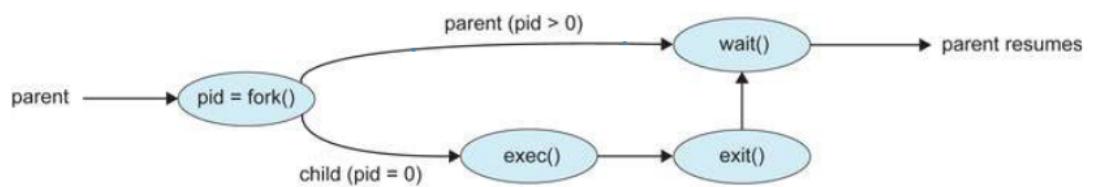
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) {
        /* execute child process code */

        execlp("/bin/ls", "ls", NULL);
    }
    else {
        /* execute parent process code */

        ...
        /* wait for child to terminate */
        wait(NULL)
        printf("Child terminated");

        exit(0);
    }
}
```

- L'albero decisionale del programma si sviluppa come tale:



2. • Consideriamo il seguente codice:

```
int pid = fork();

if(pid == 0) {      // A's child (B)
    pid = fork();

    if(pid == 0) {      // B's child (C)
        ...
        execlp(...);
    }
    else { // B
        ...
    }
}
else { // A
    ...
}
```

- La gerarchia dei processi generata corrisponde a:



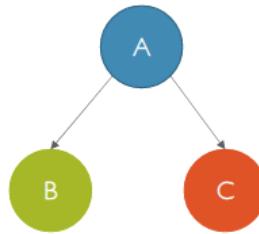
3. • Consideriamo il seguente codice:

```
int pid = fork();

if(pid == 0) {      // A's child (B)
    ...
    execlp(...);
}
else { // A
    pid = fork();

    if(pid == 0) {      // A's child (C)
        ...
        execlp(...);
    }
}
```

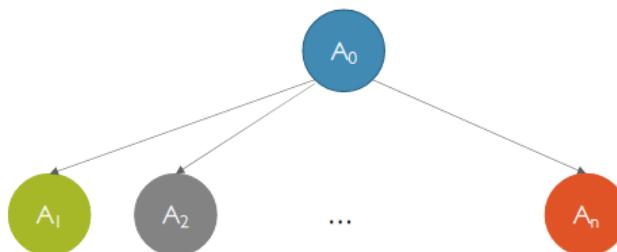
- La gerarchia dei processi generata corrisponde a:



4. • Consideriamo il seguente codice:

```
for(int i=0;i<n;i++) {  
    if(fork() == 0) {           // A0's child  
        ...  
        execlp(...);  
    }  
}  
  
// back in the parent A0  
// wait for all children to terminate  
for(int i=0;i<n;i++) {  
    wait(NULL);  
}
```

- La gerarchia dei processi generata corrisponde a:



## 2.3 Terminazione dei processi

Ogni processo può richiedere la sua **terminazione immediata** eseguendo la syscall **exit()**, la quale, tipicamente, ritorna un intero. L'intero ritornato viene passato al processo padre nel caso in cui esso stia eseguendo la syscall **wait()**.

I processi possono anche essere terminati immediatamente dal sistema operativo stesso, ad esempio nel caso in cui sia necessario liberare risorse o nel caso in cui venga eseguito un comando **kill**. Inoltre, un padre potrebbe uccidere un suo figlio se il compito ad esso assegnato non è più necessario.

Quando un processo termina, tutte le sue **risorse** di sistema vengono **liberate** (i dati vengono de-allocate, i file aperti vengono chiusi, ...) e viene ritornato lo **stato di processo terminato**, il **tempo di esecuzione** e un **codice di esito dell'esecuzione** (solitamente, viene ritornato 0 se l'esecuzione viene completata senza problemi, altrimenti viene ritornato un valore diverso da 0), i quali verranno passati al processo padre nel caso in cui esso stia eseguendo la syscall **wait()**

### Definition 9. Processo orfano e processo zombie

Se un processo padre esegue la syscall **exit()** e un suo figlio non è ancora terminato, quest'ultimo viene detto **processo orfano**, venendo solitamente ereditato da **init**, il quale poi si occuperà di ucciderli tramite **kill**.

Se invece un processo figlio tenta di terminare ma il suo processo padre non sta eseguendo la syscall **wait()**, allora tale processo viene detto **processo zombie**, il quale eventualmente verrà ereditato da **init** per poi essere ucciso.

## 2.4 Scheduling dei processi

### Definition 10. Process State Queue

Per poter gestire tutti i PCB dei vari processi attivi, il sistema operativo è dotato di **queue**, ossia code all'interno delle quali essi vengono inseriti. In particolare, vengono utilizzate **5 code per gestire gli stati dei processi** (una per ogni stato) e **una coda per ogni dispositivo I/O**.

Quando l'OS modifica lo stato di un processo, il PCB associato viene spostato da una coda all'altra, a seconda delle politiche di gestione dell'OS stesso.

La quantità di PCB che possono essere inseriti all'interno della **running queue** è strettamente legato al numero di core utilizzabili dall'OS, mentre per tutte le altre queue non vi è alcuna restrizione.

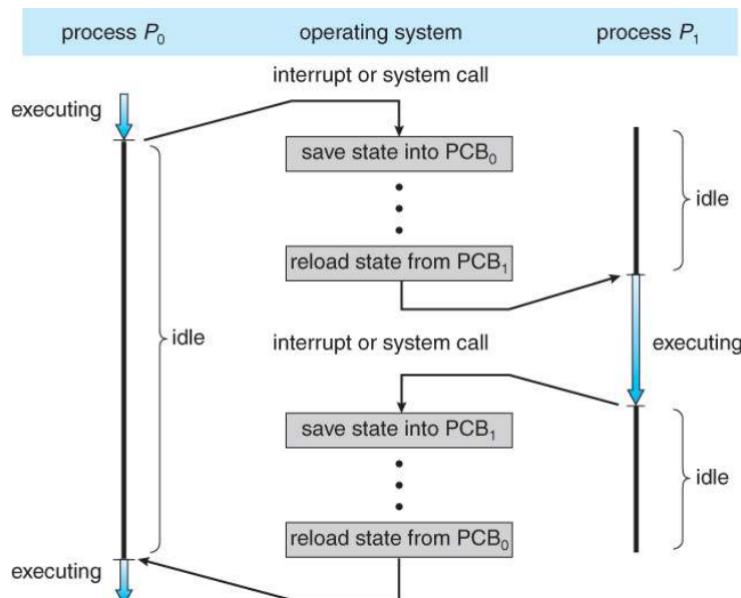
### Definition 11. Context Switch

Il **context switch** è la procedura utilizzata dalla CPU per sospendere il processo attualmente in esecuzione per poter eseguirne un altro in stato di ready.

Il context switch viene eseguito **quando viene attivata una trap di sistema**, dunque quando viene eseguita una syscall, viene gestita un'eccezione o eseguito un interrupt hardware, oppure una volta **superato il time slice**, ossia la massima quantità di tempo attesa dalla CPU prima di eseguire un context switch.

Poiché per passare da un processo all'altro è necessario salvare lo stato attuale del processo nel suo PCB per poi caricare lo stato salvato nel PCB del processo da eseguire, l'operazione di context switch risulta essere un'operazione molto **costosa**.

Un time slice minore implica una reattività massimizzata, sottraendo tuttavia tempo all'esecuzione dei processi per via del tempo impiegato per completare un context switch, richiedendo quindi un **compromesso** tra i due



## 2.5 Comunicazione tra processi

### Definition 12. Processi indipendenti e cooperanti

Un processo può essere **indipendente**, ossia ininfluente e non influenzato da altri processi, oppure **cooperante**, ossia influente o influenzato da altri al fine di svolgere un compito comune.

I processi cooperanti possono comunicare tra loro tramite una zona di **memoria condivisa** oppure tramite lo **scambio di messaggi**.

- **Memoria condivisa**

- Più veloce una volta inizializzata, poiché non richiede syscall
- Più complessa da inizializzare e da usare tra processi distribuiti su più computer
- Preferibile per lo scambio di grosse quantità di informazioni tra processi sullo stesso computer
- La memoria da condividere è inizialmente all'interno dell'address space di un particolare processo, richiedendo di eseguire alcune syscall per poter rendere accessibile la memoria ad altri processi, i quali eseguiranno delle syscall per poter connettere tale memoria condivisa al loro spazio

- **Scambio di messaggi**

- Più lento poiché richiedente una syscall per ogni scambio
- Più semplice da inizializzare e da usare tra processi su più computer
- Preferibile quando la quantità e la frequenza delle informazioni da trasferire sono basse o quando più computer sono coinvolti
- Affinché sia utilizzabile, è necessario che l'OS sia provvisto di syscall per poter permettere ai processi di scambiare e ricevere messaggi, richiedendo che venga preventivamente stabilito un canale di comunicazione tra i due processi cooperanti

Nello scambio di messaggi è necessario:

- Scegliere l'utilizzo di una **comunicazione diretta**, dove il mittente deve sapere preventivamente il nome del processo destinatario, il quale a sua volta necessita di sapere il nome del mittente, oppure una **comunicazione indiretta**, tramite l'uso di mailbox o porte
- Scegliere se utilizzare o meno una queue per i messaggi:
  - **Zero capacity**, dove i messaggi non possono essere salvati in una queue, dunque il mittente rimane bloccato finché il destinatario non riceve il messaggio
  - **Bounded capacity**, dove vi è una queue di capienza predefinita, permettendo ai mittenti di non bloccarsi a meno che la queue non sia piena
  - **Unbounded capacity**, dove la queue ha *teoricamente* capienza infinita, implicando che i mittenti non debbano mai bloccarsi

## 2.6 Scheduler della CPU

### Definition 13. CPU burst e I/O burst

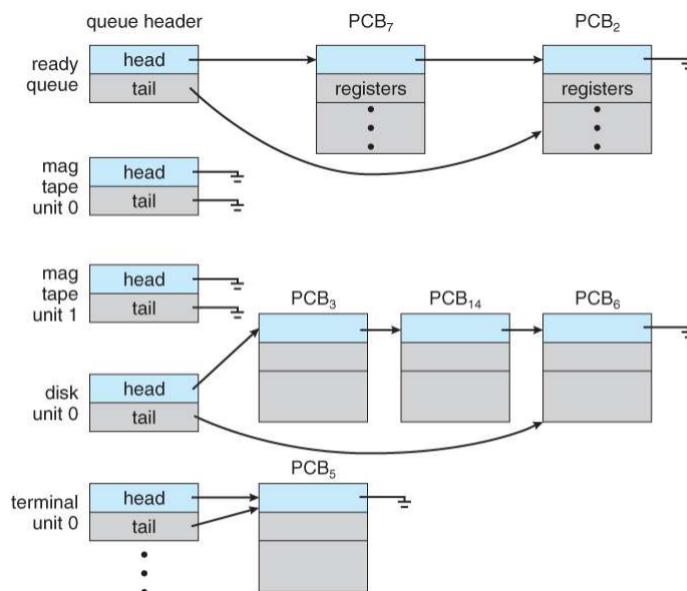
Definiamo come **CPU burst** lo stato in cui un processo è eseguito dalla CPU, mentre definiamo come **I/O burst** lo stato in cui un processo è in attesa che i dati vengano trasferiti dentro o fuori dal sistema.

In generale, ogni processo alterna costantemente tra CPU burst e I/O burst.

### Definition 14. Scheduler della CPU

Lo **scheduler della CPU** è un processo che, a seconda di una policy di scheduling stabilita, si occupa di selezionare un processo dalla ready queue da eseguire in ogni momento in cui la CPU è inattiva.

La struttura dati utilizzata per la ready queue e l'algoritmo usato per selezionare il processo successivo è basato su una queue FIFO (First In, First Out)



Le **decisioni di scheduling** della CPU vengono prese in una delle seguenti **quattro condizioni**:

1. Quando un processo passa **dal running state al waiting state** (ad esempio quando viene effettuata una richiesta I/O oppure viene invocata la syscall `wait()`)
2. Quando un processo passa **dal running state al ready state** (ad esempio in risposta ad un interrupt)
3. Quando un processo passa **dal waiting state al ready state** (ad esempio quando viene completata una richiesta I/O oppure viene terminata una syscall `wait()`)
4. Quando un processo viene **creato o terminato**

**Definition 15. Scheduling Preemptive e Non-Preemptive**

Uno scheduling viene detto **non-preventivo (non-preemptive)** se lo scheduler non ha alcuna scelta se non il selezionare un nuovo processo (condizioni 1 e 4). Una volta che un processo viene avviato, esso continua ad essere eseguito finché esso non decide volontariamente di bloccarsi oppure finché esso non termina.

Uno scheduling viene detto **preventivo (preemptive)** se lo scheduler deve scegliere se continuare ad eseguire il processo attuale oppure selezionarne uno nuovo (condizioni 2 e 3)

Lo scheduling preemptive potrebbe causare **problematiche** nel caso in cui esso avvenga mentre il kernel è occupato ad effettuare una syscall (ad esempio l'aggiornamento di strutture dati critiche per il kernel) o nel caso in cui due processi condividono dati (uno dei due potrebbe essere interrotto nel mezzo dell'aggiornamento di una struttura dati condivisa tra i due).

Le contromisure applicabili per prevenire tali problematiche sono:

- Far aspettare il processo finché la syscall del kernel non viene completata o bloccata prima di procedere con lo scheduling
- Disabilitare gli interrupt prima di entrare nella sezione di codice critica, per poi riabilitarle subito dopo.

**Definition 16. Dispatcher**

Il **dispatcher** corrisponde al modulo che cede il controllo della CPU al processo selezionato dallo scheduler. Le sue funzioni includono l'esecuzione del **context switch**, l'esecuzione del **passaggio alla user mode** e il saltare alla posizione corretta nel programma appena caricato. Il tempo impiegato dal dispatcher viene detto **dispatch latency**.

**Definition 17. Tempistiche dello scheduling**

- **Arrival time:** l'istante di arrivo del processo nella ready queue
- **Start time:** l'istante in cui la CPU esegue la prima istruzione di un processo
- **Completion time:** l'istante in cui il processo completa la sua esecuzione
- **Burst time:** il tempo richiesto da un processo per l'esecuzione della CPU
- **Tournaround time:** il tempo trascorso tra il completion time e l'arrival time ( $T_{turnaround} = T_{completion} - T_{arrival}$ )
- **Waiting time:** la differenza di tempo tra il turnaround time e il burst time ( $T_{waiting} = T_{turnaround} - T_{burst}$ )

*Attenzione:* il tempo di attesa per l'I/O non viene considerato, poiché i processi in waiting state non sono sotto il controllo dello scheduler della CPU

Durante la scelta dell'algoritmo di scheduling è necessario considerare più criteri, in particolare:

- La **massimizzazione dell'utilizzo della CPU**, ossia la percentuale di tempo in cui la CPU è occupata (idealmente 100%, su sistemi concreti è sufficiente che sia tra il 40% e il 90%)
- La **massimizzazione del throughput**, ossia il numero di processi completati in una determinata unità di tempo
- La **minimizzazione del turnaround time**
- La **minimizzazione del waiting time**
- La **minimizzazione del response time**, ossia il tempo impiegato dall'emissione del comando all'inizio della risposta alla richiesta, principalmente per processi interattivi

Tipica dei **sistemi interattivi** è una politica di scheduling portata alla minimizzazione del response time medio, minimizzazione del response time massimo e minimizzazione della varianza del response time.

Tipica dei **sistemi batch**, invece, è una politica di scheduling portata alla massimizzazione del throughput e alla minimizzazione del waiting time.

### Observation 3

Per le politiche di scheduling assumiamo che:

- Vi è un singolo processo per utente
- I processi sono indipendenti tra di loro, dunque non vi è comunicazione
- Ogni processo è costituito da un singolo thread

### Definition 18. Job, Processo e Thread

Un **job** è l'unità generica di esecuzione della CPU, differenziandosi in **processi** e **thread**, ossia una sotto-istanza di un processo facente parte di una sua suddivisione in più thread

### 2.6.1 First Come First Served (FCFS)

#### Method 2. First Come First Served (FCFS)

L'algoritmo **First Come First Served (FCFS)** è un algoritmo di scheduling **non-preemptive** dove, appena creati, i job vengono inseriti in una queue FIFO e lo scheduler esegue il primo job della queue fino al suo completamento, per poi procedere con il job successivo.

Lo scheduler prende il controllo della CPU solo nel caso in cui il processo in esecuzione richieda un'operazione di I/O oppure termina la sua esecuzione.

#### Pro:

- Estremamente semplice da implementare

#### Contro:

- Un job potrebbe essere eseguito indefinitamente (ad esempio finché esso non si blocca)
- Il waiting time medio è molto variabile poiché job con pochi CPU burst potrebbero essere in coda dopo job con molti CPU burst
- Può crearsi un **effetto convoglio**, dovuto alla sovrapposizione tra CPU e I/O poiché i processi strettamente legati alla CPU forzeranno i processi strettamente legati all'I/O ad aspettare

#### Esempi:

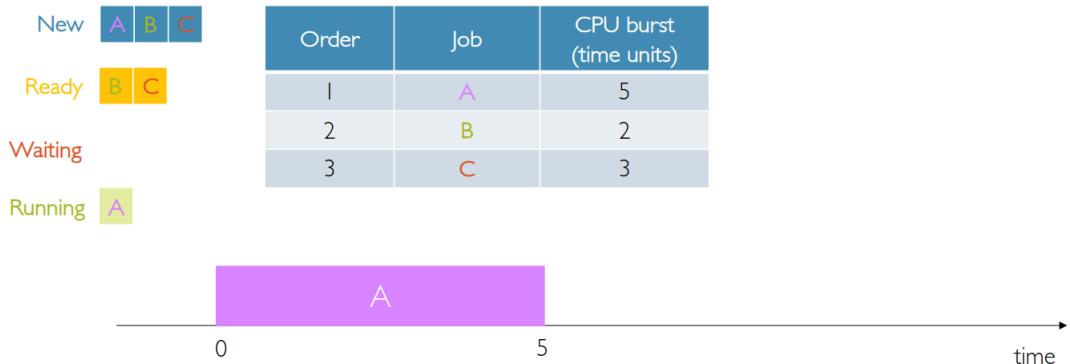
- Consideriamo la seguente coda di processi:

New	A   B   C	Order	Job	CPU burst (time units)
		1	A	5
		2	B	2
		3	C	3

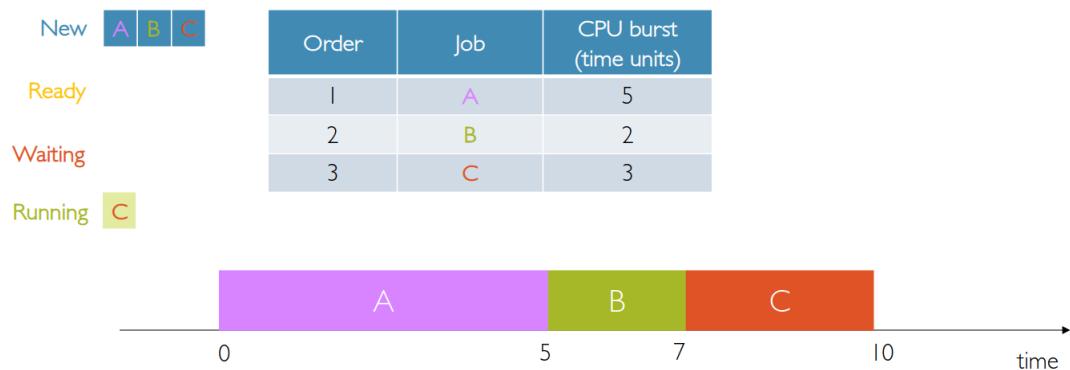
- I processi vengono creati allo stesso istante, dunque l'**arrival time** per tutti i processi è l'istante 0. Successivamente, tutti e tre i processi vengono messi nella ready queue

New	A   B   C	Ready	A   B   C	Waiting	Running	Order	Job	CPU burst (time units)
						1	A	5
						2	B	2
						3	C	3

- Di seguito, lo scheduling FCFS seleziona il primo processo della ready queue, ossia il processo A, spostandolo in stato di ready e completando la sua esecuzione



- Una volta terminato il processo A, lo scheduler ripeterà le stesse operazioni fino a che tutti i processi non saranno terminati



- Una volta terminati tutti i processi, calcoliamo il waiting time medio:

– Per il processo A si ha:

$$T_{waiting} = T_{tournaround} - T_{burst} = T_{completion} - T_{arrival} - T_{burst} = 5 - 0 - 5 = 0$$

– Per il processo B si ha:

$$T_{waiting} = T_{tournaround} - T_{burst} = T_{completion} - T_{arrival} - T_{burst} = 7 - 0 - 2 = 5$$

– Per il processo C si ha:

$$T_{waiting} = T_{tournaround} - T_{burst} = T_{completion} - T_{arrival} - T_{burst} = 10 - 0 - 3 = 7$$

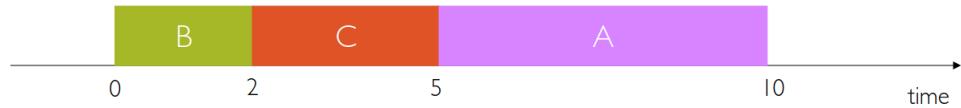
– Il waiting time medio, quindi, sarà:

$$\bar{T}_{waiting} = \frac{1}{n} \sum_{i=0}^n T_i^{waiting} = \frac{0 + 5 + 7}{3} = 4$$

2. • Nel seguente scenario, utilizzando sempre uno scheduling FCFS, si ha:

New [B | C | A]

Order	Job	CPU burst (time units)
1	B	2
2	C	3
3	A	5



- Il waiting time medio, quindi, sarà:

$$\bar{T}_{waiting} = \frac{1}{n} \sum_{i=0}^n T_i^{waiting} = \frac{5 + 0 + 2}{3} \approx 2.3$$

3. • Consideriamo il seguente scenario:

New [A | B | C]

Ready [A | B | C]

Order	Job	CPU burst (time units)
1	A	5
2	B	2
3	C	3

- Supponiamo che dopo 2 unità temporali il processo A esegua una richiesta I/O, entrando quindi in stato di waiting

New [A | B | C]

Ready [B | C]

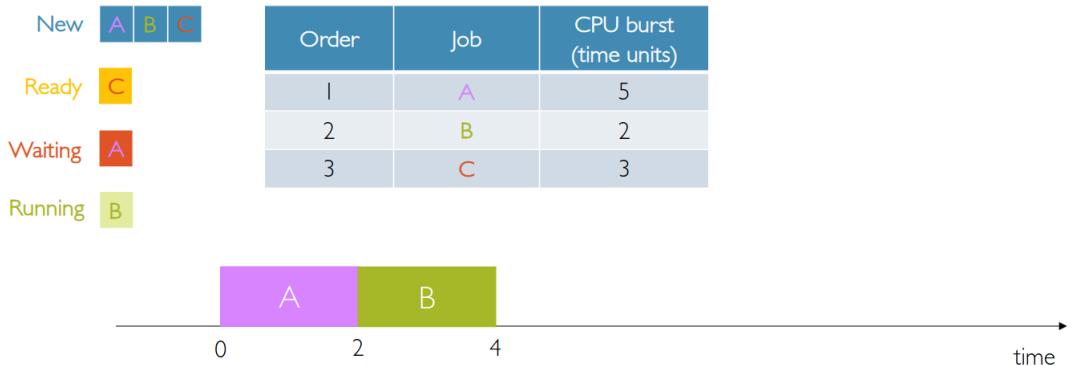
Waiting

Running [A]

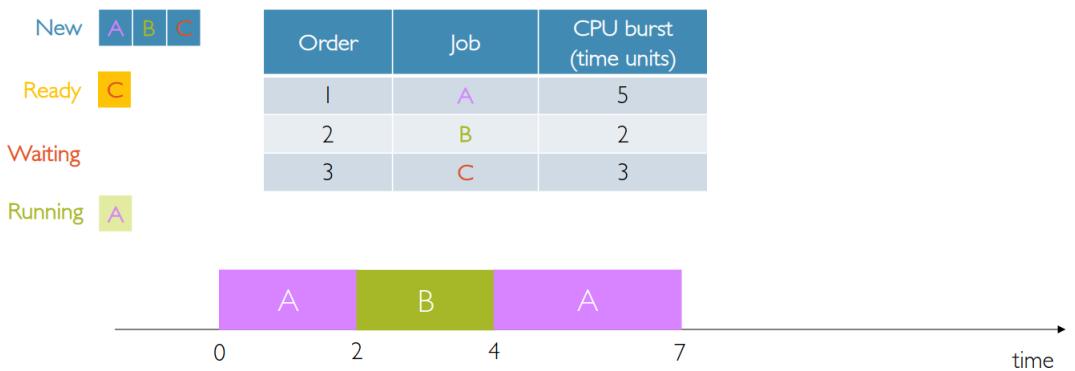
Order	Job	CPU burst (time units)
1	A	5
2	B	2
3	C	3



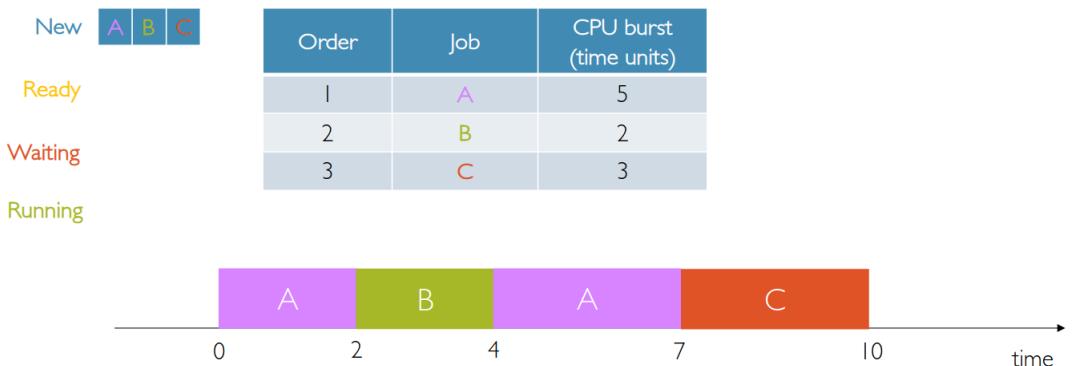
- Una volta entrato in stato di waiting, lo scheduler selezionerà il processo seguente nella coda, ossia il processo B, portandolo a termine



- Una volta terminata l'esecuzione del processo B, il processo A verrà selezionato per riprendere l'esecuzione. **Attenzione:** viene selezionato il processo A e non il processo C poiché l'algoritmo FCFS utilizza una queue basata sull'ordine di creazione dei processi, dunque sul loro arrival time



- Infine, il waiting time medio sarà:



$$\bar{T}_{\text{waiting}} = \frac{1}{n} \sum_{i=0}^n T_i^{\text{waiting}} = \frac{2 + 2 + 7}{3} \approx 3.7$$

## 2.6.2 Round Robin (RR)

### Method 3. Round Robin (RR)

L'algoritmo **Round Robin (RR)** è un algoritmo di scheduling **preemptive** simile al FCFS, con l'aggiunta di un timer che funga da limite al CPU burst dei job, detto **time slice (o time quantum)**:

- Ogni volta che un job prende controllo della CPU, il timer del time slice viene avviato
- Se il job termina prima che il time slice esaurisca, allora lo scheduler si comporterà come un normale algoritmo FCFS
- Se il time slice esaurisce prima che il job termini, allora lo scheduler sposterà il job in esecuzione alla fine della ready queue

Per via della natura del RR, la ready queue viene gestita come una **coda circolare**, distribuendo il tempo di utilizzo della CPU equamente tra tutti i processi.

#### Pro:

- Più equo rispetto all'FCFS, poiché viene fornita la stessa quantità di tempo di utilizzo della CPU ad ogni job

#### Contro:

- Un time slice troppo ampio rischia di far degenerare un RR in un FCFS
- Un time slice troppo stretto rischia di generare un elevato numero di context switch, diminuendo l'uso effettivo della CPU

### Observation 4

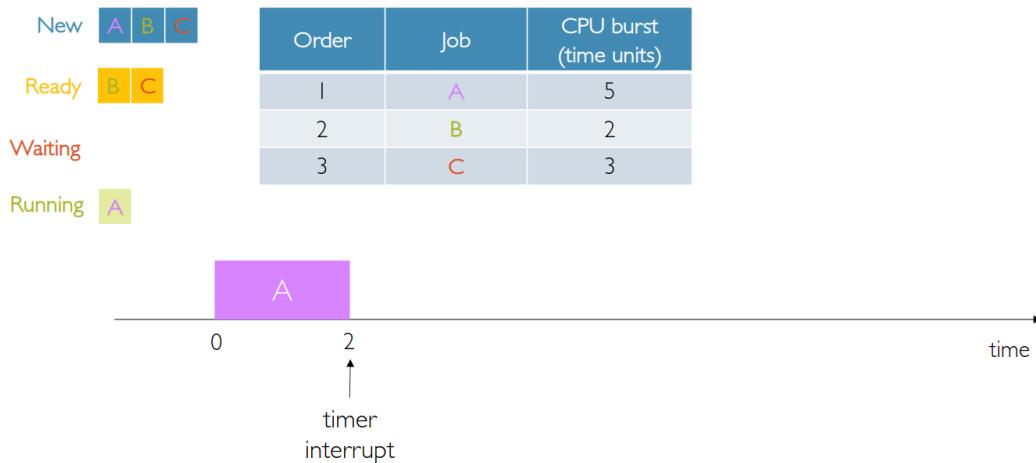
Il limite superiore per lo **start time** di un processo in una coda di  $n$  processi utilizzando uno scheduler RR corrisponde a  $\sup\{T_i^{start}\} = \delta \cdot (i - 1)$ , dove  $i \in [1, n]$  e dove  $\delta$  è il time slice

#### Esempio:

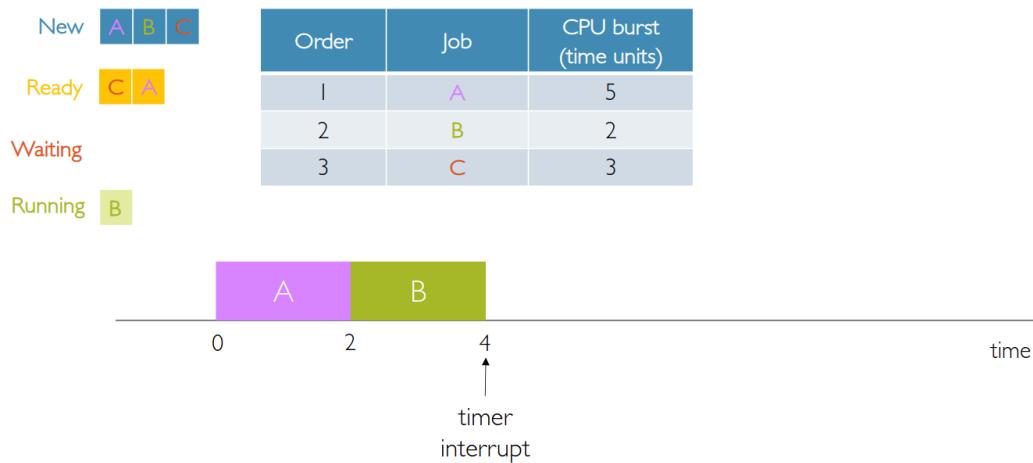
- Consideriamo la seguente coda dei processi utilizzando un algoritmo RR con un time slice di 2 e un context switch trascurabile:

New	A   B   C	Ready	A   B   C	Waiting	A   B   C	Running	A   B   C
		Order	Job	CPU burst (time units)			
		1	A	5			
		2	B	2			
		3	C	3			

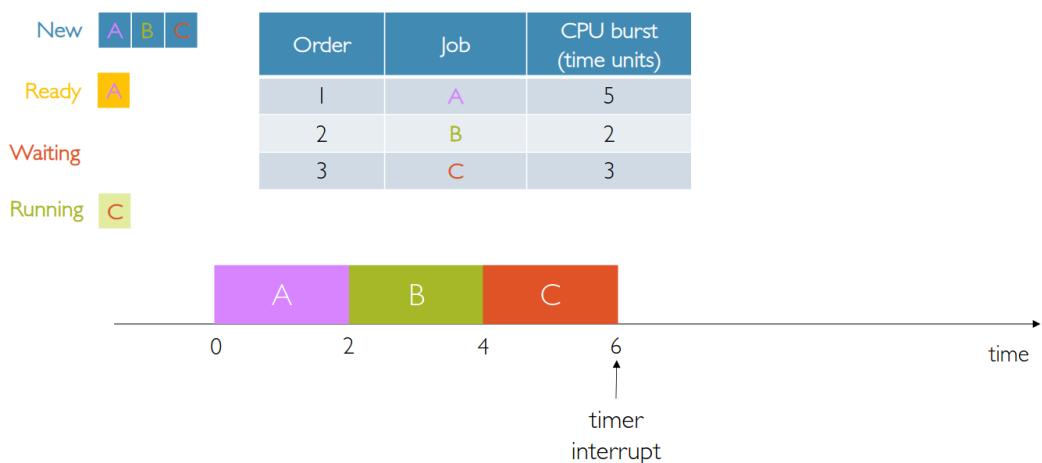
- Il primo processo a prendere il controllo della CPU è il processo A, venendo bloccato ed aggiunto alla ready queue dopo 2 unità temporali per via del time slice



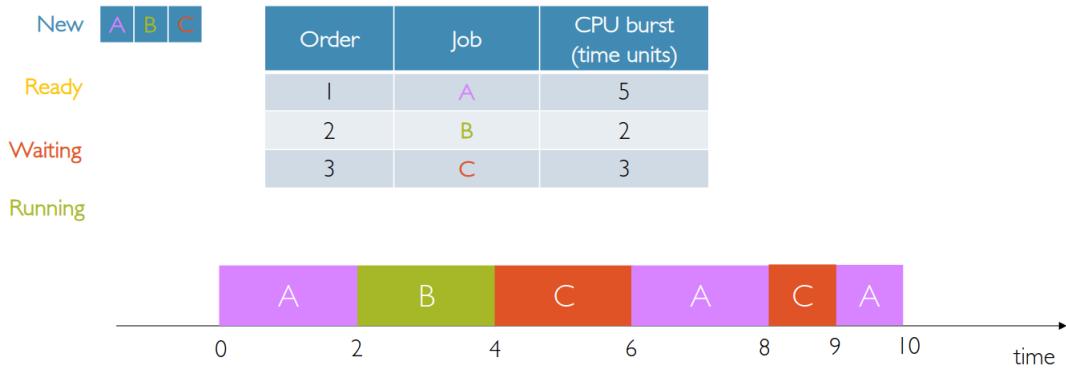
- Successivamente, il processo B prende controllo della CPU, venendo anch'esso bloccato dopo 2 unità temporali. In questo caso, tuttavia, il processo B non verrà aggiunto alla fine della ready queue poiché la sua esecuzione è terminata



- Di seguito, il processo C prenderà il controllo, venendo bloccato dopo 2 tempi



- Infine, vengono eseguiti i processi A e C finché essi non verranno completati



- Il waiting time medio, quindi, sarà:

$$\bar{T}_{waiting} = \frac{1}{n} \sum_{i=0}^n T_i^{waiting} = \frac{5 + 2 + 6}{3} \approx 4.3$$

### Observation 5. FCFS vs RR

Confrontando il turnaround time e il waiting time medio tra uno scheduler FCFS e uno scheduler RR, il primo sembra essere a primo occhio più performante del secondo.

Tuttavia, considerando la varianza tra i waiting time di ogni processo, notiamo come il RR risulta essere **più equo**, fornendo ad ogni processo la stessa quantità di tempo di utilizzo della CPU.

### Esempio:

Job	CPU burst	turnaround time		waiting time	
		FCFS	RR	FCFS	RR
A	100	100	496	0	396
B	100	200	497	100	397
C	100	300	498	200	398
D	100	400	499	300	399
E	100	500	500	400	400
Avg.		300	498	200	398

### 2.6.3 Shortest Job First (SJF)

#### Method 4. Shortest Job First (SJF)

L'algoritmo **Shorted Job First (SJF)** è un algoritmo di scheduling basato sull'esecuzione prioritaria del job con la **quantità minore stimata di CPU burst**.

L'algoritmo SJF può essere:

- **non-preemptive**, dove ogni job che ottiene il controllo della CPU viene eseguito fino al suo completamento
- **preemptive**, dove ogni volta che un nuovo job arriva nella ready queue e il suo CPU burst stimato è minore di quello rimanente del job attualmente in esecuzione, tale job prende controllo della CPU. Questa versione del SJF viene anche detta **Shortest Remaining Time First (SRTF)**

**Pro:**

- Ottimale nel caso in cui l'obiettivo sia minimizzare il waiting time medio
- Funzionale con scheduler sia non-preemptive sia preemptive

**Contro:**

- È quasi impossibile stimare esattamente il tempo di esecuzione della CPU di un job
- I processi strettamente legati alla CPU attivi da molto tempo potrebbero andare in *starvation*

Per stimare la lunghezza del prossimo CPU burst, viene utilizzato l'**exponential smoothing**:

- Sia  $x_t$  la lunghezza effettiva del  $t$ -esimo CPU burst
- Sia  $S_{t+1}$  la lunghezza stimata del  $(t + 1)$ -esimo CPU burst
- Sia  $\alpha \in R$  dove  $0 \leq \alpha < 1$
- Si ha che:

$$s_1 = x_0$$

$$s_{t+1} = \alpha x_t + (1 - \alpha) s_t$$

**Esempi:**

- Consideriamo la seguente coda di processi gestita da uno scheduler SJF non-preemptive. In tal caso, si ha che:

Job	CPU burst (time units)
A	6
B	8
C	7
D	3



$$T_{avg.\ waiting} = \frac{3 + 16 + 9 + 0}{4} = 7$$

- Consideriamo la seguente coda di processi gestita da uno scheduler SJF preemptive, ossia uno scheduler SRTF

Job	Arrival time	CPU burst (time units)
A	0	8
B	1	4
C	2	9
D	3	5



- Il primo job ad essere eseguito è il processo A, poiché il primo ad essere creato ed inserito nella ready queue.

Job	Arrival time	CPU burst (time units)
A	0	8
B	1	4
C	2	9
D	3	5



- Una volta che il processo B viene creato, il suo CPU burst stimato, ossia 4, viene comparato con il CPU burst rimanente del processo A, ossia 7. Poiché  $4 < 7$ , allora il processo B prende controllo della CPU.

Analogamente, nell'istante in cui il processo C viene creato, il suo CPU burst stimato, ossia 9, viene comparato con quello rimanente del processo A, ossia 6, e quello del processo B, ossia 3. Siccome  $3 < 7 < 9$ , allora il processo B mantiene il controllo della CPU.

Lo stesso ragionamento viene effettuato anche dopo la creazione del processo D, dove il processo B mantiene il controllo ( $2 < 7 < 5 < 9$ ). Di conseguenza, il processo B verrà eseguito fino al suo completamento.

Job	Arrival time	CPU burst (time units)
A	0	8
B	1	4
C	2	9
D	3	5



- Una volta completato il processo B, verrà eseguito il processo avente il CPU burst stimato minore:
  - Il processo A ha un CPU burst rimanente pari a 7
  - Il processo C ha un CPU burst rimanente pari a 9
  - Il processo D ha un CPU burst rimanente pari a 5

Di conseguenza, verrà eseguito il processo D fino al suo completamento

job	Arrival time	CPU burst (time units)
A	0	8
B	1	4
C	2	9
D	3	5



- Analogamente al processo D, verranno eseguiti il processo A e il processo C, ognuno di essi fino al loro completamento (poiché, nel frattempo, nessun altro processo viene inserito nella ready queue, dunque non viene mai attivato l'algoritmo di scheduling)

job	Arrival time	CPU burst (time units)
A	0	8
B	1	4
C	2	9
D	3	5



$$T_{avg.\ waiting} = \frac{(17 - 0 - 8) + (5 - 1 - 4) + (26 - 2 - 9) + (10 - 3 - 5)}{4} = 6.5$$

### Confronto tra FCFS, RR e SJF

Job	CPU burst	turnaround time			waiting time		
		FCFS	RR	SJF	FCFS	RR	SJF
A	50	50	150	150	0	100	100
B	40	90	140	100	50	100	60
C	30	120	120	60	90	90	30
D	20	140	90	30	120	70	10
E	10	150	50	10	140	40	0
Avg.		110	110	70	80	80	40

## 2.6.4 Priority Scheduling

### Method 5. Priority Scheduling

Gli algoritmi basati sul **priority scheduling** sono algoritmi di scheduling dove ad ogni job viene assegnato valore, il quale determina la sua **priorità**. Il job con priorità maggiore viene schedulato per primo.

Le priorità possono essere assegnate in due modi:

- **Internamente**, ossia assegnate dall'OS in base a determinati criteri (ad esempio il CPU burst medio, il rateo di attività tra CPU e I/O, risorse utilizzate, ...)
- **Esteriormente**, ossia assegnate dall'utente in base all'importanza del job o di altri criteri

Il priority scheduling può essere sia **non-preemptive** sia **preemptive**.

#### Contro:

- **Starvation** (o bloccaggio infinito), dove un job di bassa priorità rimane in attesa perenne poiché altri job hanno sempre una priorità maggiore.

Tali job potrebbero essere eseguiti eventualmente quando il carico del sistema è minore o dopo che il sistema stesso vada in crash, venga spento o venga riavviato

Come contromisura alla starvation viene utilizzato l'**aging**, ossia l'incremento della priorità di un job in base al suo tempo in attesa, finché essi non verranno eventualmente schedulati.

### Observation 6

L'algoritmo SJF è un algoritmo con **priority scheduling** dove la priorità è dettata dal CPU burst minore

## 2.6.5 Multilevel Queueing (MLQ e MLFQ)

### Method 6. Multilevel Queue (MLQ)

L'algoritmo **Multilevel Queue (MLQ)** è un algoritmo di scheduling basato sull'utilizzo di queue multiple separate tra loro, ognuna per ogni **categoria** di job, dove ogni queue utilizza l'algoritmo di scheduling più appropriato per una determinata categoria di job. Una volta inserito all'interno di una queue, **nessun job può essere spostato in un'altra queue**.

Le due opzioni più comuni per l'implementazione di tale algoritmo sono:

- **String priority**, dove nessun job all'interno di una queue di priorità più bassa viene eseguito finché esiste almeno un job delle queue di priorità più alta
- **Round robin**, dove ogni queue utilizza un proprio time slice, il quale aumenta esponenzialmente al diminuire della priorità della coda

### Method 7. Multilevel Feedback Queue (MLFQ)

L'algoritmo **Multilevel Feedback Queue (MLFQ)** segue la stessa idea dell'algoritmo MLQ, con l'aggiunta della **possibilità per ogni job di essere spostato da una queue all'altra**.

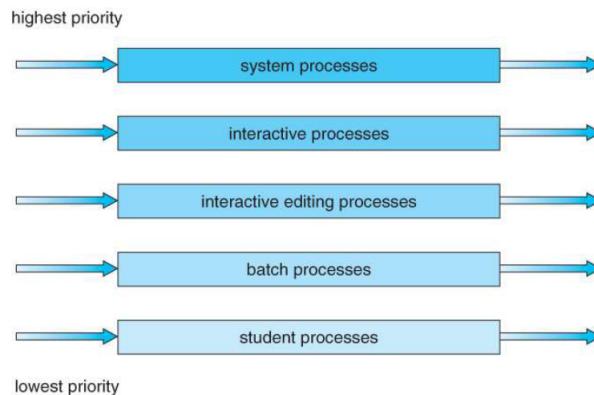
Lo spostamento di un job può rivelarsi necessario quando:

- Un job passa dall'utilizzare molto la CPU all'utilizzare molto l'I/O e viceversa
- Un job è in stato di starving, venendo spostato per breve tempo in una queue a priorità più alta

Per gestire gli spostamenti, l'algoritmo utilizza le seguenti regole:

- Inizialmente ad ogni job viene assegnata la **priorità più alta**
- Se il time slice di un job **scade**, allora quest'ultimo viene spostato nella queue con un livello di priorità **inferiore** rispetto a quella precedente
- Se il time slice di un job **non scade**, allora quest'ultimo viene spostato nella queue con un livello di priorità **superiore** rispetto a quella precedente
- La priorità di processi strettamente legati alla CPU viene diminuita rapidamente
- La priorità di processi strettamente legati all'I/O rimarrà alta

### Esempio di suddivisione tramite MLQ e MLFQ:



### Esempi:

- Consideriamo la seguente coda di processi gestita da un algoritmo MLFQ con 3 queue e strict priority.

New [ A | B | C ]

Order	Job	CPU burst (time units)
1	A	30
2	B	20
3	C	10

Queue	Time Slice (time units)	Jobs
1	1	
2	2	
3	4	

- Per indicare l'avanzamento di ogni processo all'avanzare del tempo, utilizziamo la notazione  $JOB_{total\_time\_elapsed}^{execution\_time}$

New [ A | B | C ]

Order	Job	CPU burst (time units)
1	A	30
2	B	20
3	C	10

Queue	Time Slice (time units)	Jobs
1	1	$A^1_1, B^1_2, C^1_3$
2	2	
3	4	

- Siccome il time slice per tutti e tre i processi è scaduto, ognuno di essi viene spostato nella queue a priorità inferiore

New	A	B	C	
Order	Job	CPU burst (time units)		
1	A	30		
2	B	20		
3	C	10		

Queue	Time Slice (time units)	Jobs
1	1	A <sup>1</sup> <sub>1</sub> , B <sup>1</sup> <sub>2</sub> , C <sup>1</sup> <sub>3</sub>
2	2	A <sup>3</sup> <sub>5</sub> , B <sup>3</sup> <sub>7</sub> , C <sup>3</sup> <sub>9</sub>
3	4	

- Analogamente, a prima, anche in questo caso il time slice per tutti e tre scade, dunque vengono spostati alla queue inferiore

New	A	B	C	
Order	Job	CPU burst (time units)		
1	A	30		
2	B	20		
3	C	10		

Queue	Time Slice (time units)	Jobs
1	1	A <sup>1</sup> <sub>1</sub> , B <sup>1</sup> <sub>2</sub> , C <sup>1</sup> <sub>3</sub>
2	2	A <sup>3</sup> <sub>5</sub> , B <sup>3</sup> <sub>7</sub> , C <sup>3</sup> <sub>9</sub>
3	4	A <sup>7</sup> <sub>13</sub> , B <sup>7</sup> <sub>17</sub> , C <sup>7</sup> <sub>21</sub>

- Una volta raggiunta la queue a priorità più bassa, il time slice di tutti i processi scadrà sempre fino al loro completamento

New	A	B	C	
Order	Job	CPU burst (time units)		
1	A	30		
2	B	20		
3	C	10		

Queue	Time Slice (time units)	Jobs
1	1	A <sup>1</sup> <sub>1</sub> , B <sup>1</sup> <sub>2</sub> , C <sup>1</sup> <sub>3</sub>
2	2	A <sup>3</sup> <sub>5</sub> , B <sup>3</sup> <sub>7</sub> , C <sup>3</sup> <sub>9</sub>
3	4	A <sup>7</sup> <sub>13</sub> , B <sup>7</sup> <sub>17</sub> , C <sup>7</sup> <sub>21</sub> A <sup>11</sup> <sub>25</sub> , B <sup>11</sup> <sub>29</sub> , C <sup>10</sup> <sub>32</sub>

## 2.6.6 Lottery scheduling

### Method 8. Lottery scheduling

L'algoritmo **lottery scheduling** è un algoritmo di scheduling basato sulla **casualità**:

1. Ad ogni job vengono assegnati un determinato numero di biglietti. I job attivi da poco riceveranno più biglietti, mentre quelli attivi da molto ne riceveranno di meno. Per evitare la starvation, ad ogni job viene assegnato almeno un biglietto
2. Ogni volta che un time slice scade viene determinato un biglietto vincitore. Il processo che detiene tale biglietto prenderà il controllo della CPU.
3. Successivamente, il procedimento viene ripetuto

Per via della **legge dei grandi numeri**, all'aumentare del numero di estrazioni effettuate il tempo di utilizzo della CPU di ogni processo tenderà a raggiungere la media.

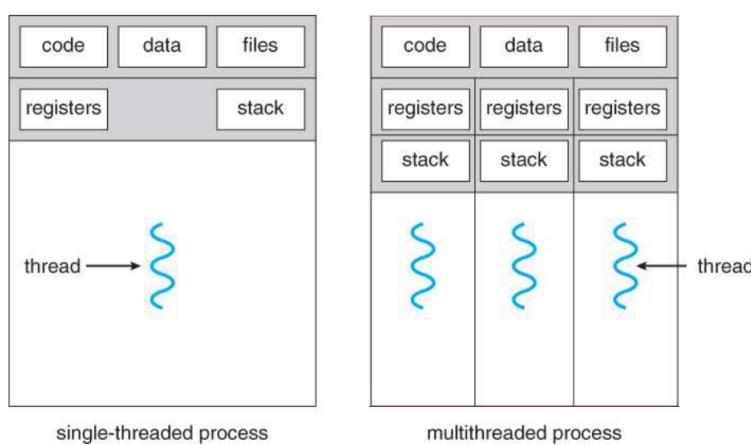
## 2.7 Thread e Multi-threading

### Definition 19. Thread

Ogni processo può essere costituito da **uno o più thread**, ognuno composto da un proprio program counter, un proprio stack, un proprio set di registri e un proprio ID.

Ogni processo definisce le **risorse "globali"** (ossia lo spazio d'indirizzamento, le istruzioni da eseguire, i dati, le risorse, ...,) mentre ogni suo thread definisce **un singolo stream di esecuzione** all'interno del processo stesso.

Poiché lo spazio d'indirizzamento del processo è **condiviso tra tutti i suoi thread**, nessuna syscall è richiesta per far cooperare i thread tra di loro, risultando in una comunicazione più semplice rispetto allo scambio di messaggi e alla memoria condivisa.



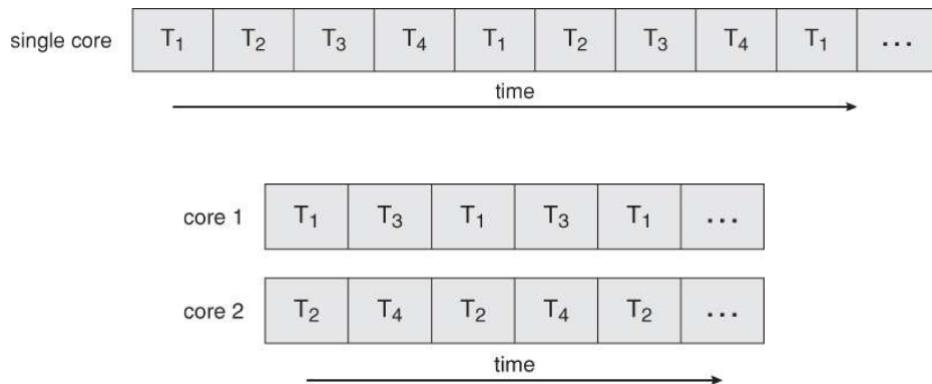
L'utilizzo dei thread risulta essere particolarmente utile nel caso in cui un processo deve svolgere **più attività indipendenti l'una dall'altra**. In particolare, nel caso in cui un'attività richieda che il processo si **blocchi**, utilizzando più thread viene concesso alle altre attività di continuare ad essere svolte senza dover aspettare l'attività bloccata.

Teoricamente, ogni attività svolta da un software potrebbe essere svolta da un processo single-thread comunicante con gli altri. Tuttavia, l'utilizzo dei thread per la suddivisione delle attività svolte da un processo risulta essere la scelta migliore poiché:

- La comunicazione tra thread risulta essere estremamente **più veloce** di quella tra processi
- Il context switch tra processi risulta essere estremamente **più veloce** rispetto a quello tra processi

In definitiva, possiamo riassumere i **vantaggi dei processi multi-thread** in:

- **Reattività**, poiché un processo potrebbe fornire una risposta in modo rapido mentre tutti gli altri thread sono bloccati o rallentati a causa dell'intenso uso della CPU
- **Condivisione delle risorse "globali"**
- **Economicità**, poiché creare e gestire thread è più veloce rispetto alla creazione e gestione dei processi
- **Scalabilità**, poiché un processo single-thread può essere eseguito su un singolo core, mentre ogni thread di un processo multi-thread può essere suddiviso tra tutti i core disponibili della CPU (solo per architetture multi-core)

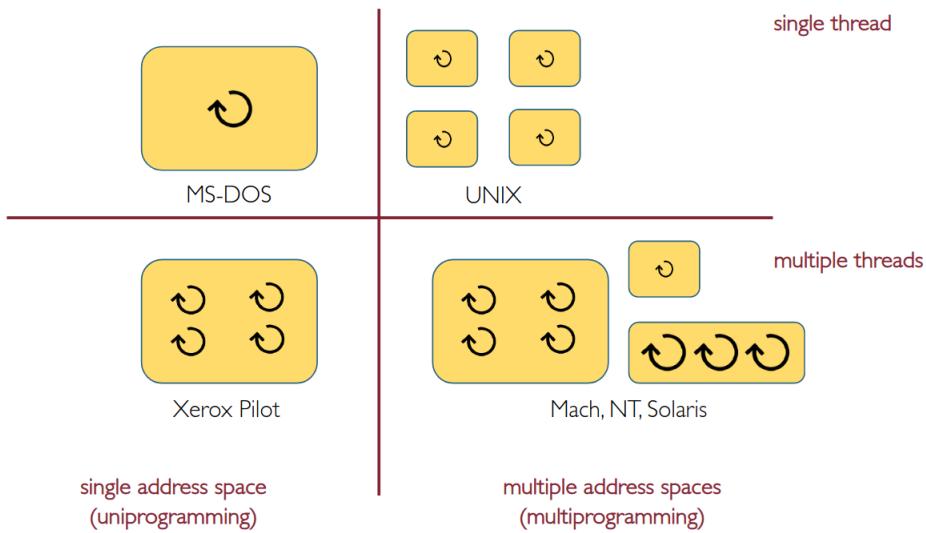


L'utilizzo di un'architettura basata su processori multi-core permette un **vero parallelismo tra processi**. Spesso, le CPU moderne sono anche dotate di **hyperthreading**, dove ogni singolo core fisico appare come due core logici all'OS, permettendo lo **scheduling concorrente** di due processi per ogni core.

In generale, esistono due modi per parallelizzare il carico di lavoro:

- **Data parallelism**, dove i dati sono suddivisi tra più core (thread), svolgendo la stessa attività su ogni blocco di dati
- **Task parallelism**, dove le varie attività vengono svolte da core differenti in simultaneo

Classificazione degli OS:



### Method 9

Il supporto per il **multi-threading** può essere fornito tramite a livello di kernel space tramite i **kernel thread** oppure a livello di user space tramite gli **user thread**.

### Definition 20. Kernel thread

Un **kernel thread** è l'unità di esecuzione più piccola schedulabile dall'OS, il quale è responsabile per il supporto e la gestione di tutti i kernel thread attivi, fornendo delle syscall per poterli creare e gestire dall'user space. Ogni kernel thread è dotato di un **Thread Control Block (TCB)**

#### Pro:

- Il kernel è a conoscenza di tutti i kernel thread avviati
- Lo scheduler potrebbe decidere di cedere più tempo di esecuzione della CPU ad un processo costituito da un numero elevato di thread
- Utile per applicazioni con blocchi frequenti
- Passare da un thread all'altro è più veloce rispetto al passare da un processo all'altro

#### Contro:

- Rende il kernel estremamente più complesso
- Lento ed inefficiente, poiché è necessario effettuare chiamate al kernel
- Nonostante il context switch sia più leggero, la sua gestione tramite il kernel non è consigliata

**Definition 21. User thread**

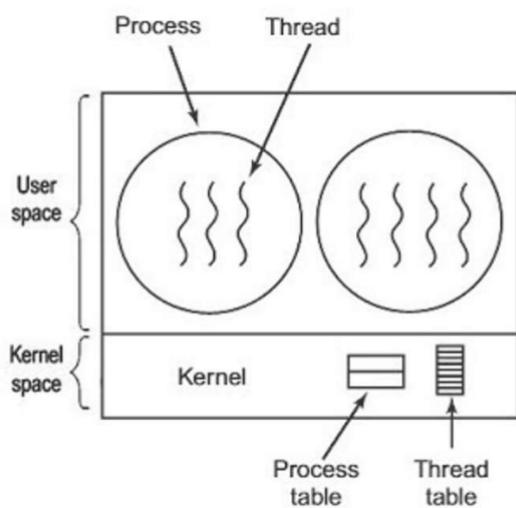
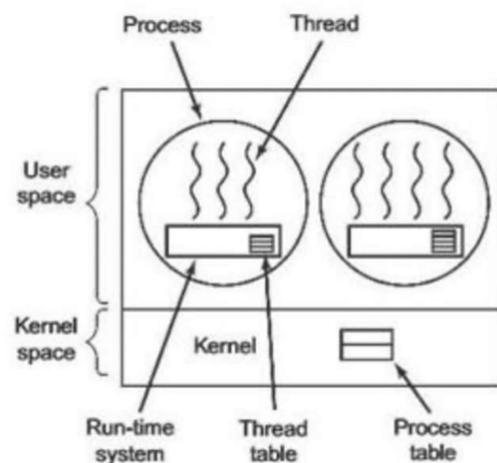
Al contrario dei kernel thread, gli **user thread** sono gestiti interamente dal sistema di run-time tramite **library per la gestione dei thread**, i quali vengono gestiti come se fossero dei processi single-threaded. Idealmente, le attività svolte da tali thread dovrebbero essere veloci come chiamate a funzioni.

**Pro:**

- Molto veloci e leggeri
- Le politiche di scheduling sono più flessibili
- Può essere implementato da OS che non supportano il threading in modo nativo
- Nessuna syscall o context switch necessario

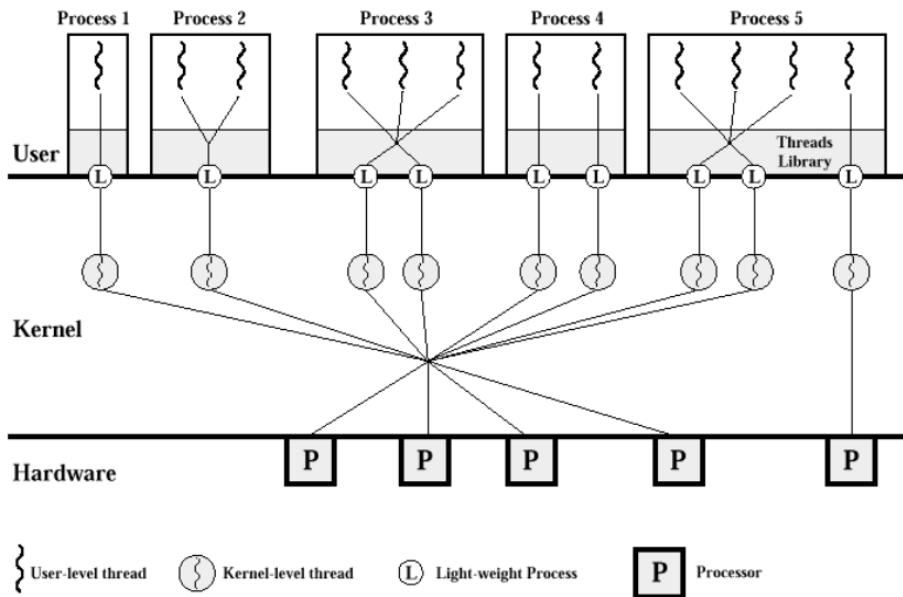
**Contro:**

- Il kernel non è a conoscenza degli user thread attivi
- Non vi è una vera concorrenza per i processi multi-threaded
- Le decisioni prese dallo scheduler spesso sono inefficienti
- Mancanza di coordinazione tra kernel e thread (ad esempio, un processo con 100 thread potrebbe competere per un time slice con un processo con un solo thread)
- Richiede necessariamente che tutte le syscall dell'OS siano non bloccanti, poiché altrimenti tutti i thread in un processo dovrebbero attendere

**Kernel Thread****User Thread**

### Definition 22. Lightweight Process

Un **lightweight process (LWP)** è un **processore virtuale** attivo nella user space contenente un **singolo kernel thread**. **Multipli user thread** gestiti tramite una libreria per i thread vengono posti su uno o più LWP, i quali assumono un ruolo di "tramite" tra le due tipologie di thread.



#### 2.7.1 Modelli di multi-threading

Per gestire il **multithreading**, vengono utilizzati più modelli:

- **Modello Many-to-One**
  - Più user thread vengono mappati ad un singolo kernel thread
  - Ogni processo può eseguire **un solo user thread per volta** poiché vi è un singolo kernel thread associato, il quale può operare su un singolo core, dunque i processi con più user-thread non possono essere suddivisi tra più core
  - Se viene effettuata una syscall bloccante, l'**intero processo viene bloccato** anche nel caso in cui gli altri user-thread potrebbero continuare l'esecuzione.
  - **Puramente basato sul livello user**
- **Modello One-to-One**
  - Ogni singolo user thread viene mappato ad un singolo kernel thread
  - Ogni processo può eseguire **più user thread per volta** poiché vi sono più kernel thread associati, i quali possono essere suddivisi tra più core

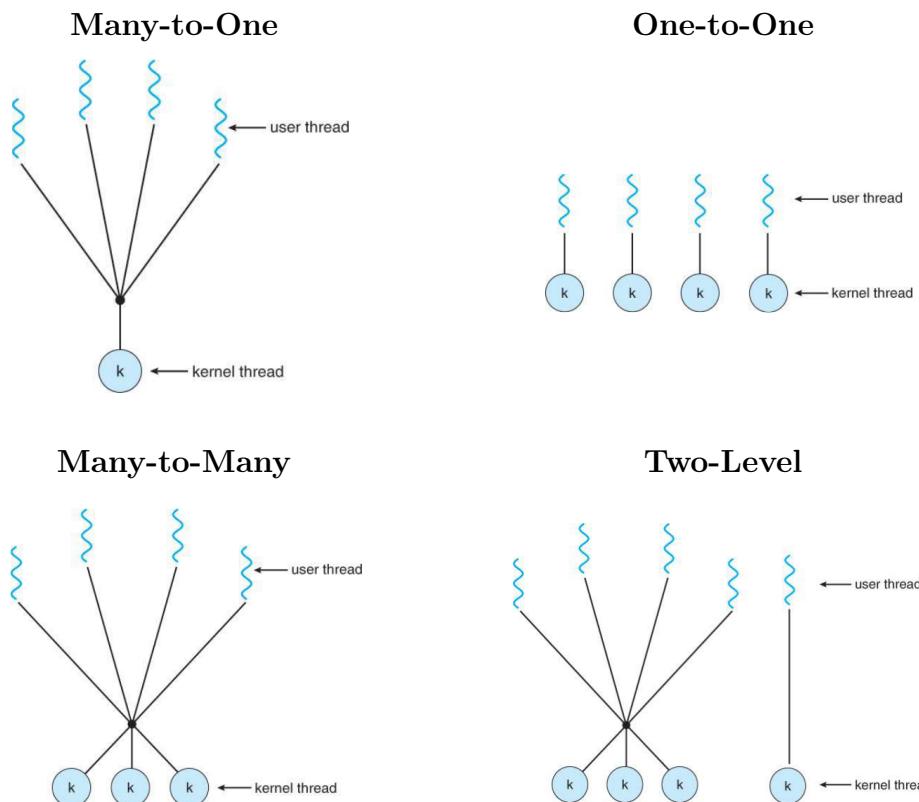
- Se viene effettuata una syscall bloccante, l'**intero processo non viene bloccato**
- È richiesto un kernel molto più complesso, portando il sistema ad essere rallentato
- La maggior parte delle implementazioni fissano un **limite al numero di thread** che possono essere creati
- **Puramente basato sul livello kernel**

- **Modello Many-to-Many**

- Più user thread vengono mappati a più kernel thread
- Il numero di kernel thread può essere minore o uguale al numero di user thread
- I processi possono essere **suddivisi tra più core**
- Le syscall bloccanti **non bloccano l'intero processo**
- Il numero di thread creabili **non è limitato**

- **Modello Two-Level**

- Variante del modello Many-to-Many, dove alcuni thread vengono gestiti tramite il modello **One-to-One**
- Le politiche di scheduling risultano **più flessibili**



### Observation 7

Se un thread effettua una syscall `fork()` o `exec()`, l'**intero processo** potrebbe essere copiato oppure potrebbe essere generato un **nuovo processo single-thread** contenente una copia del singolo thread.

Tale problematica viene gestita in uno dei seguenti modi:

1. È strettamente dipendente dalla progettazione dell'OS
2. Se il nuovo processo viene eseguito subito, allora non c'è necessità di copiare anche gli altri thread. Altrimenti, l'intero processo dovrebbe essere copiato.
3. Molte versioni di UNIX forniscono più versioni delle syscall `fork()` ed `exec()` per gestire casi specifici

### Observation 8

Nel caso in cui un processo multi-thread riceva un segnale, è importante gestire quale thread sia il destinatario di tale segnale.

Tale problematica viene gestita in uno dei seguenti modi:

1. Il segnale viene inviato al thread che lo necessita
2. Il segnale viene inviato ad ogni thread del processo
3. Il segnale viene inviato ad alcuni thread specifici del processo
4. Viene scelto un thread specifico che vada a gestire tutti i segnali ricevuti

### Definition 23. Scheduling dei thread

Per essere schedulato ed eseguito dalla CPU, ogni thread deve contendere con gli altri. I thread contendenti vengono gestiti dal **contention scope**, il quale si suddivide in due tipologie:

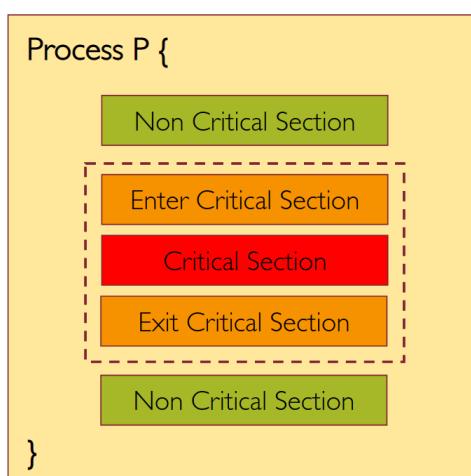
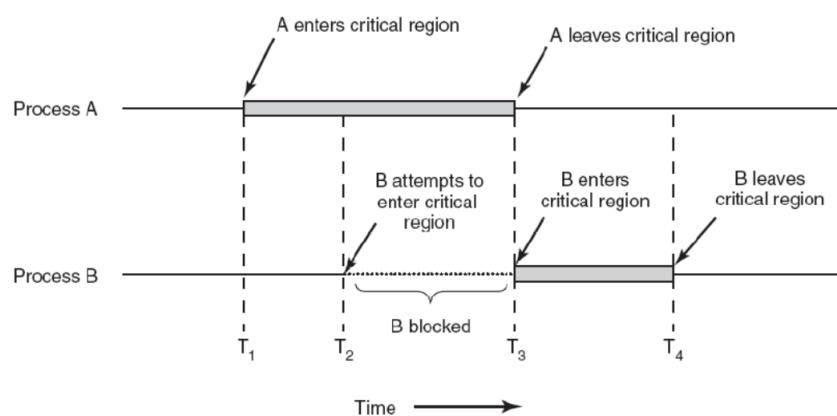
- **Process Contention Scope (PCS)**, dove la competizione avviene tra thread appartenenti allo stesso processo. Affinché sia implementabile, è necessario che il sistema utilizzi un modello Many-to-Many o Many-to-One
- **System Contention Scope (SCS**, dove la competizione avviene tra ogni thread di ogni processo. Viene implementato in sistemi basati sul modello One-to-One

Tramite il lightweight process (LWP), il kernel è in grado di comunicare con le librerie per i thread nel momento in cui un determinato evento si verifica attraverso delle **upcall**, le quali vengono gestite da un **upcall handler** presente all'interno della libreria stessa. Ogni upcall fornisce un nuovo LWP tramite cui l'upcall handler verrà eseguito.

# Capitolo 3

## Sincronizzazione tra Processi/Thread

Abbiamo già accennato come i processi o i thread possano cooperare tra di loro per svolgere un'attività comune. Tuttavia, affinché la cooperazione sia possibile è richiesta una **sincronizzazione** tra di essi per via della presenza di **settori critici**. Per evitare che tali settori generino problemi, vengono utilizzate delle primitive che si assicurino che **un solo thread** possa lavorare sul settore stesso.



### Proposition 1. Proprietà della sincronizzazione

Ogni soluzione al problema del settore critico tramite la sincronizzazione deve soddisfare tre proprietà:

- **Mutua esclusione**, ossia solo un processo/thread per volta può utilizzare il settore critico
- **Liveness**, ossia la possibilità per ogni processo di accedere al settore critico nel caso in cui nessun altro processo vi sia già
- **Attesa limitata**, ossia la possibilità per ogni processo richiedente l'accesso al settore critico di potervi accedere eventualmente, limitando il numero di processi che vi accederanno prima di esso

Per poter sincronizzare i processi/thread tra di loro, i linguaggi di programmazione forniscono delle primitive atomiche basate su una delle seguenti **tre soluzioni**:

- La sincronizzazione avviene tramite un **lock**, dove prima di accedere ad un settore critico un processo acquisisce tale lock, per poi rilasciarlo una volta uscito dal settore critico
- La sincronizzazione avviene tramite un **semaphore**, una generalizzazione del lock
- La sincronizzazione avviene tramite un **monitor**, il quale connette dei dati condivisi alle primitive di sincronizzazione

#### 3.0.1 Lock

##### Definition 24. Lock

Un **lock** fornisce la mutua esclusione tra processi utilizzando due primitive:

- **Lock.acquire()**, dove il processo rimane in attesa che il lock sia libero, per poi acquisirlo
- **Lock.release()**, dove il processo rilascia il lock precedentemente acquisito, inviando un segnale a tutti i processi che attualmente stanno eseguendo la primitiva `acquire()`

Per poter utilizzare correttamente un lock, è necessario seguire tre regole:

1. È sempre necessario acquisire il lock **prima** di accedere a dati condivisi
2. È sempre necessario rilasciare il lock **dopo** aver finito di utilizzare i dati condivisi
3. Il lock deve essere **inizialmente libero**
4. **Solo un processo per volta** può acquisire il lock

---

Poiché lo scheduler della CPU prende il controllo solo a seguito di **eventi interni**, ossia quando il thread in esecuzione lascia il controllo della CPU (ad esempio a seguito di una syscall), oppure a seguito di **eventi esterni** (ad esempio a seguito dell'esaurimento time slice).

Per evitare problemi all'interno dei settori critici, quindi, è necessario **impedire che possa avvenire un context switch**, dunque prevenendo che la CPU possa prendere il controllo dello scheduler quando viene eseguita la primitiva `acquire()`.

Per implementare le primitive ad alto livello relative alla sincronizzazione è necessario del **supporto hardware al basso livello** ossia la **disabilitazione degli interrupt** e le **istruzioni atomiche**.

### Method 10. Disabilitazione degli interrupt

La **prima soluzione a supporto hardware** per poter implementare la sincronizzazione, corrisponde alla **disabilitazione degli interrupt** mentre un thread **acquisisce o rilascia un lock**

```
Class Lock {
    public void acquire(Thread t);
    public void release();
    private int value; // 0=FREE, 1=BUSY
    private Queue q;

    Lock() {
        // lock is initially FREE
        this.value = 0;
        this.q = null;
    }
}
```

```
public void acquire(Thread t) {
    disable_interrupts();
    if(this.value) { // lock is held by someone
        q.push(t); // add t to waiting queue
        t.sleep(); // put t to sleep
    }
    else {
        this.value = 1;
    }
    enable_interrupts();
}
```

```
public void release() {
    disable_interrupts();
    if(!q.is_empty()) {
        t = q.pop(); // extract a waiting thread from q
        push_onto_ready_queue(t); // put t on ready queue
    }
    else {
        this.value = 0;
    }
    enable_interrupts();
}
```

## Method 11. Istruzioni atomiche

La seconda soluzione a supporto hardware per poter implementare la sincronizzazione, corrisponde all'uso di un'**istruzione atomica** in grado di leggere un valore dalla memoria e modificarlo in un singolo istante.

Nella maggior parte delle architetture, viene utilizzata l'istruzione **test&set**:

- Nel caso in cui il lock sia libero (dunque il suo valore associato è 0), l'istruzione `test&set(value)` leggerà 0 per poi settare tale valore ad 1, restituendo il valore 0 letto (**acquisizione del lock**)
- Nel caso in cui il lock sia occupato (dunque il suo valore associato è 1), l'istruzione `test&set(value)` leggerà 1 per poi settare tale valore ad 1, restituendo il valore 1 letto (**attesa del lock**)

```
Class Lock {  
    public void acquire();  
    public void release();  
    private int value;  
  
    Lock() {  
        // lock is initially free  
        this.value = 0;  
    }  
}
```

```
public void acquire() {  
    while(test&set(this.value) == 1) {  
        // while busy do nothing  
    }  
}
```

```
public void release() {  
    this.value = 0;  
}
```

## Observation 9

L'implementazione del lock tramite **disabilitazione degli interrupt** presenta alcune problematiche:

- La necessità dell'invocazione del kernel rende il tutto più complesso
- Inutilizzabile in sistemi multi-core

## Observation 10

Analogamente, anche l'implementazione del lock tramite **istruzioni atomiche** presenta alcune problematiche:

- Ogni thread che non ha acquisito il lock è in attesa che esso si liberi, dunque la CPU non sta realmente svolgendo delle attività
- Una volta che il lock è libero, tutti i thread in attesa si contenderanno il lock, dunque non è possibile determinare quale thread otterrà il controllo

---

Nonostante il tempo in attesa della CPU sia un problema irrisolvibile nel caso delle istruzioni atomiche, esso può essere **minimizzato**. Inoltre, l'aggiunta di una **queue** permette di gestire in modo deterministico quale thread vada ad ottenere il lock a seguito del suo rilascio:

```
Class Lock {  
    public void acquire(Thread t);  
    public void release();  
    private int value;  
    private int guard;  
    private Queue q;  
  
    Lock() {  
        // lock is initially free  
        this.value = 0;  
    }  
}
```

```
public void acquire(Thread t) {  
    while(test&set(this.guard) == 1) {  
        // while busy do nothing  
    }  
    if(this.value) {  
        q.push(t);  
        t.sleep_and_reset_guard_to_0();  
    }  
    else {  
        this.value = 1;  
        this.guard = 0;  
    }  
}
```

```
public void release() {  
    while(test&set(this.guard) == 1) {  
        // while busy do nothing  
    }  
    if(!q.is_empty()) {  
        t = q.pop();  
        push_onto_ready_queue(t);  
    }  
    else {  
        this.value = 0;  
    }  
    this.guard = 0;  
}
```

### 3.0.2 Semafori

#### Definition 25. Semaforo

Un **semaforo** è una generalizzazione del lock dotata di una propria **queue** ed una **variabile** sulla quale vengono svolte due primitive:

- **wait()** (o **P()**), la quale decrementa la variabile e inserisce il processo nella queue, in attesa che il semaforo sia aperto. Se quando la primitiva viene invocata il semaforo è aperto, allora il processo continua l'esecuzione.
- **signal()** (o **V()**), la quale incrementa la variabile e permette ad un altro thread di accedere al settore critico. Se un thread è in attesa nella coda, allora esso viene sbloccato. Altrimenti, se non ci sono thread nella coda, il segnale viene ricordato per il thread successivo.

I semafori si differenziano in due tipi:

- **Semafori binari**, detti **Mutex**, utilizzati per gestire una singola risorsa condivisa, dove il valore iniziale della variabile associata corrisponde a 1. Di conseguenza, il suo funzionamento coincide con quello di un lock.
- **Semafori di conteggio**, utilizzato per gestire risorse multiple condivise, dove il valore iniziale della variabile associata corrisponde al numero di risorse stesso. Un processo può accedere ad una risorsa finché almeno una di esse è disponibile

```
// Semaphore S

S.wait(); // wait until S is available

<critical section>

S.signal(); notify other processes that S is open
```

```
Class Semaphore {
    public void wait(Thread t);
    public void signal();
    private int value;
    private int guard;
    private Queue q;

    Semaphore(int val) {
        // initialize semaphore
        // with val and empty queue
        this.value = val;
        this.q = null;
    }
}
```

```

public void wait(Thread t) {
    while(test&set(this.guard) == 1) {
        // while busy do nothing
    }
    this.value -= 1;
    if(this.value < 0) {
        q.push(t);
        t.sleep_and_reset_guard_to_0();
    }
    else {
        this.guard = 0;
    }
}

```

```

public void signal() {
    while(test&set(this.guard) == 1) {
        // while busy do nothing
    }
    this.value += 1;
    if(!q.isEmpty()) {
        t = q.pop();
        push_onto_ready_queue(t);
    }
    this.guard = 0;
}

```

### Esempi:

1. Considerando i seguenti due processi A e B, un possibile sviluppo dell'esecuzione è il seguente:



	s (value)	Queue	A	B
	2	Ø	ready to exec	ready to exec
A: S.wait()	1	Ø	ready to exec	ready to exec
B: S.wait()	0	Ø	ready to exec	ready to exec
A: S.wait()	-1	A	blocked	ready to exec
B: S.signal()	0	Ø	ready to exec	ready to exec
A: S.signal()	1	Ø	ready to exec	ready to exec
A: S.signal()	2	Ø	ready to exec	ready to exec

- 
2. Consideriamo i due seguenti processi Producer e Consumer:

**Producer Process:**

```
while (true)
{
    /* produce an item in nextProduced */
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

**Consumer Process:**

```
while (true)
{
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in nextConsumed */
}
```

- I due processi condividono un **buffer comune**. La variabile `counter` tiene traccia del numero di elementi attualmente nel buffer.
- In tale caso, potrebbe crearsi una **race condition**: in assenza di sincronizzazione, i due processi potrebbero lavorare contemporaneamente sulla variabile
- Ad esempio, ipotizziamo che inizialmente si abbia `counter = 5`. In tal caso, una possibile esecuzione del programma potrebbe essere la seguente:

**Producer:**

```
register1 = counter
register1 = register1 + 1
counter = register1
```

**Consumer:**

```
register2 = counter
register2 = register2 - 1
counter = register2
```

**Interleaving:**

$T_0$ :	producer	execute	$register_1 = \text{counter}$	{ $register_1 = 5$ }
$T_1$ :	producer	execute	$register_1 = register_1 + 1$	{ $register_1 = 6$ }
$T_2$ :	consumer	execute	$register_2 = \text{counter}$	{ $register_2 = 5$ }
$T_3$ :	consumer	execute	$register_2 = register_2 - 1$	{ $register_2 = 4$ }
$T_4$ :	producer	execute	$\text{counter} = register_1$	{ $\text{counter} = 6$ }
$T_5$ :	consumer	execute	$\text{counter} = register_2$	{ $\text{counter} = 4$ }