

Sistemi operativi  
**Natella Roberto**  
a.a. 2023-2024

**Author**  
Alessio Romano

February 12, 2025

# Contents

<b>1 Sistemi Operativi</b>	<b>5</b>
1.1 Sistemi single-user . . . . .	5
1.2 Batch Systems . . . . .	5
1.3 Sistemi multiprogrammati . . . . .	5
1.4 Sistemi time-sharing . . . . .	5
1.5 Personal Computing . . . . .	6
1.6 Embedded Computing . . . . .	6
1.7 Mobile Computing . . . . .	6
1.8 Distributed, Cloud, Edge Computing . . . . .	6
<b>2 Cenni calcolatori</b>	<b>7</b>
2.1 Interfacciamento tra CPU e I/O . . . . .	7
2.2 Interrupt . . . . .	8
2.3 Protezione delle risorse hardware . . . . .	9
2.4 Protezione di memoria e delle memorie . . . . .	10
<b>3 Architettura SO</b>	<b>10</b>
3.1 Kernel . . . . .	10
3.2 Gestione della memoria . . . . .	11
3.3 Gestione periferiche . . . . .	11
3.4 File system . . . . .	11
<b>4 Invocazione sistema operativo</b>	<b>11</b>
4.1 System call . . . . .	11
4.2 Tipi di architetture SO . . . . .	13
4.2.1 Monolitiche e modulari . . . . .	13
4.2.2 Microkernel . . . . .	14
<b>5 Processi</b>	<b>14</b>
5.1 Stato di un processo . . . . .	14
5.1.1 Modello a 2/3 stati . . . . .	14
5.1.2 Context switch . . . . .	15
5.1.3 Modello a 5/6 stati . . . . .	16
5.1.4 Processi in Unix . . . . .	17
5.1.5 Code dei processi . . . . .	18
<b>6 Scheduling</b>	<b>18</b>
6.1 Scheduler a lungo termine . . . . .	18
6.2 Scheduler a medio termine . . . . .	19
6.3 Scheduler di breve termine . . . . .	19
6.4 Criteri, Starvation, Preemption . . . . .	19
6.4.1 Criteri di prestazione . . . . .	19
6.4.2 Starvation . . . . .	19
6.4.3 Preemption . . . . .	20
6.5 Algoritmi . . . . .	20
6.5.1 First-Come-First-Served FCFS . . . . .	20
6.5.2 Round-Robin . . . . .	21
6.5.3 Shortest Process Next . . . . .	21
6.5.4 Shortest Remaining Time . . . . .	22
6.5.5 Scheduler a Priorità . . . . .	22
6.5.6 Multilevel Feedback . . . . .	23
<b>7 SMP e Scheduling multiprocessore</b>	<b>23</b>
7.1 SO per SMP . . . . .	24
7.1.1 MASTER/SLAVE . . . . .	24
7.1.2 PEER . . . . .	24
7.1.3 Assegnazione dei processi . . . . .	25
<b>8 Scheduling in Linux</b>	<b>26</b>
8.1 Algoritmi di scheduling . . . . .	26

8.1.1 Scheduler O(1) . . . . .	26
8.1.2 CFS scheduler . . . . .	27
8.2 Cgroups . . . . .	28
<b>9 Threads</b>	<b>28</b>
9.1 Tipologie di Thread . . . . .	29
9.1.1 User-Level Thread . . . . .	29
9.1.2 Kernel-Level-Thread . . . . .	30
9.2 Implementazioni dei thread . . . . .	30
9.2.1 Thread in Go . . . . .	30
9.2.2 Thread in Linux . . . . .	30
9.2.3 Thread in Win2000 . . . . .	31
<b>10 Programmazione concorrente</b>	<b>31</b>
10.1 Multiprogrammazione . . . . .	31
10.2 Parallelismo . . . . .	31
10.2.1 Concorrenza-Parallelismo . . . . .	32
<b>11 Sincronizzazione Globale</b>	<b>32</b>
11.1 Sezione Critica . . . . .	32
11.2 Mutua Esclusione . . . . .	32
11.2.1 Lock . . . . .	33
11.2.2 Semafori e Mutex . . . . .	33
<b>12 Cooperazione</b>	<b>33</b>
12.1 Prod-Cons . . . . .	33
12.1.1 Prod-Cons single buffer . . . . .	34
12.1.2 Prod-Cons con coda . . . . .	34
12.1.3 Prod-Cons multipli con coda . . . . .	35
12.2 Lettori/Scrittori . . . . .	35
<b>13 Gestione memoria</b>	<b>35</b>
13.1 Rilocazione Dinamica . . . . .	36
13.2 MMU . . . . .	37
13.3 Gestione dello spazio virtuale . . . . .	37
13.3.1 Segmentazione . . . . .	38
13.4 Allocazione . . . . .	39
13.5 Paginazione . . . . .	40
13.5.1 Paginazione gerarchica . . . . .	42
13.5.2 Tabella delle pagine su Hash . . . . .	42
13.5.3 Tabella delle pagine invertita . . . . .	42
<b>14 Memoria Virtuale</b>	<b>43</b>
14.1 Page Fault . . . . .	43
14.1.1 Page fault con sostituzione . . . . .	44
<b>15 IPC-Shmem-Semafori</b>	<b>45</b>
15.1 IPC . . . . .	45
15.1.1 Primitiva get . . . . .	45
15.1.2 Primitiva ctl . . . . .	45
15.1.3 IPC Keys . . . . .	46
15.2 Memoria condivisa . . . . .	46
15.2.1 Creazione SHM . . . . .	46
15.3 Semafori . . . . .	47
<b>16 PThreads</b>	<b>48</b>
16.1 Gestione dei thread . . . . .	48
16.1.1 Create/Exit . . . . .	48
16.1.2 Passaggio di parametri . . . . .	49
16.1.3 Join . . . . .	49
16.1.4 Restituzione dei dati . . . . .	49
16.2 Mutex . . . . .	50

16.2.1 Creazione e distruzione . . . . .	50
--	----

# 1 Sistemi Operativi

Per **sistema operativo**, si intendono software che operano sull'hardware di un calcolatore

- **Semplificare** lo sviluppo
- **Gestire** efficacemente le risorse hardware
- **Proteggere** le informazioni e prevenire accessi senza autorizzazione

Vediamo i diversi tipi di sistemi operativi, e come si distinguono tra di essi.

## 1.1 Sistemi single-user

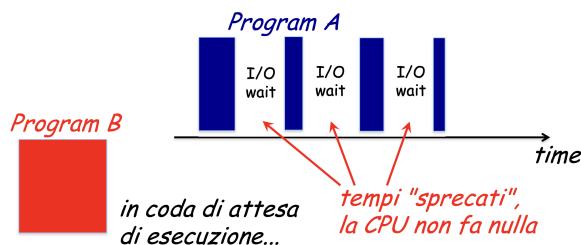
I sistemi single user, non sono propriamente definibili come sistemi operativi, in quanto caratterizzati da un'unico programma, con cui l'utente non è in grado di interagire durante l'esecuzione. Il problema principale di questo tipo di sistemi è l'effettivo utilizzo di risorse hardware

$$\% \text{ Utilizzo} = \frac{\text{Tempo di utilizzo dell'hardware}}{\text{Tempo totale}}$$

## 1.2 Batch Systems

I batch systems, o sistemi a lotti sono sistemi che si basano sulla compilazione di schede perforate che vengono successivamente lette dal calcolatore:

- L'utente scrive il suo **job** (programma) su delle schede perforate
- Più job sono raggruppati in pacchetti detti **batch**
- I job vengono caricati ed eseguiti in sequenza dal calcolatore
- L'utente attende spesso ore e giorni per avere i risultati (sistema non interattivo) Una variante dei classici Batch Systems sono i sistemi Batch monoprogrammati. In cui il sistema operativo si occupa attraverso un programma caricato in memoria di schedulare l'esecuzione dei programmi in sequenza. Si tratta a tutti gli effetti di una miglioria rispetto ai sistemi single-user in quanto la macchina è meglio utilizzata grazie all'assenza dell'interazione con l'utente, tuttavia i programmi sono eseguiti sequenzialmente, il che implica che l'utente spesso si trova ad attendere ore o giorni per ricevere i risultati e soprattutto il processore rimane inattivo durante le procedure di I/O.



Inoltre, un job alla volta veniva caricato in memoria, risultando in memoria non utilizzata

## 1.3 Sistemi multiprogrammati

Nei sistemi multiprogrammati, più job vengono caricati in memoria contemporaneamente, tuttavia la cpu esegue un job alla volta, e in caso un job sia interrotto da operazioni di I/O, la cpu viene riassegnata ad un altro job

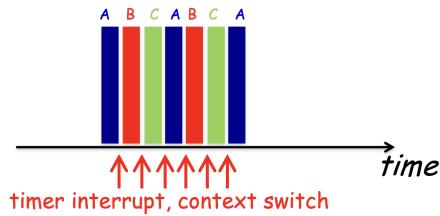
## 1.4 Sistemi time-sharing

Il sistema time-sharing, estende l'idea di multiprogrammazione, ma gli utenti non aspettano più il termine dell'esecuzione per avere risultati, ma interagiscono in tempo reale con i programmi. Il sistema operativo **unix**, rappresenta un esempio di sistema time-sharing. Nei sistemi time-sharing

- i job vengono interrotti dal sistema operativo (**preemption**) anche se non fanno I/O
- la cpu alterna frequentemente i processi (+100 v/s)

I job anche se condividono le stesse risorse hardware, "appaiono" agli utenti come se eseguissero in contemporanea su due processori diversi. Il metodo di realizzazione è il seguente:

- Un timer innesca periodicamente gli **interrupt**
- Allo scadere il sistema operativo, rimuove dalla cpu il programma corrente e ne pone un altro in esecuzione (**context switch**)



L'introduzione del timer di interrupt, introduce un ritardo di esecuzione definito come **overhead** di sistema. Il vantaggio di questo tipo di sistemi è che l'utente opera in maniera interattiva e su più programmi contemporaneamente, lo svantaggio è l'introduzione dell'overhead di sistema

	Batch Multiprogramming	Time Sharing
Obiettivo principale	Massimizzare l' <b>uso delle risorse</b>	Minimizzare i <b>tempi di risposta</b>
Modalità di utilizzo	Comandi di controllo forniti insieme al job al momento del <b>caricamento</b>	Comandi forniti tramite <b>interfaccia interattiva</b> (grafica oppure console testuale)

## 1.5 Personal Computing

Negli anni 80' nascono i primi personal computer, con sistemi operativi solitamente single-user e con un focus sull'interfacce sia grafiche che testuali come MS-DOS. Nel tempo questi si evolvono fino a diventare sistemi multiutente time-sharing come MacOS, Windows...

## 1.6 Embedded Computing

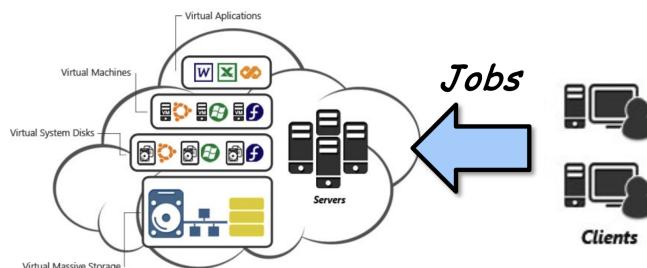
I sistemi embedded sono i sistemi integrati all'interno di oggetti o sistemi informatici che si occupano di svolgere lavori fisici, e sono estremamente legati al tempismo, ad esempio le macchine di una fabbrica ecc...

## 1.7 Mobile Computing

Sistemi operativi quali android, ios... progettati per funzionare su telefoni mobili, e con una forte enfasi sulla sicurezza, efficienza energetica e multimedia

## 1.8 Distributed, Cloud, Edge Computing

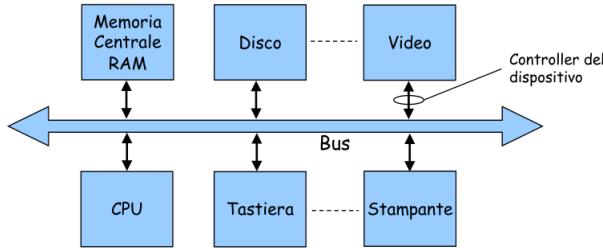
Il **Distributed computing**, è un ulteriore evoluzione nella virtualizzazione delle risorse, il sistema è formato da gruppi di macchine detti **cluster**, e un textbforchestratore ha il compito di distribuire automaticamente nel cluster le applicazioni degli utenti. L'**edge computing**, è una variante del cloud computing, in cui il calcolo avviene su nodi diversi contemporaneamente



Il **Cloud computing** si basa su dei provider che mettono a disposizione i loro data center su domanda. L'utente paga la potenza di calcolo necessaria (pay per use)

## 2 Cenni calcolatori

Il seguente schema riassuntivo, rappresenta l'architettura tipo di un calcolatore



Durante il processo di esecuzione di un programma, la cpu accede ai propri registri interni, alla memoria e alle periferiche di I/O. Lo scambio di dati e indirizzi avviene tramite bus di sistema solitamente a 8,16,32 o 64 bit. La comunicazione tra i dispositivi di I/O e la cpu avviene attraverso un **controller**

### 2.1 Interfacciamento tra CPU e I/O

Come accennato in precedenza, la comunicazione tra CPU e I/O passa attraverso due mezzi principali

- **Controller:** un sistema elettronico che gestisce il dialogo tra dispositivo e cpu tramite bus di sistema, e comunica con il dispositivo tramite collegamento fisico es: USB
- **Driver:** software di controllo del dispositivo, in esecuzione sulla CPU e parte integrante del sistema operativo

Approfondiamo il ruolo e il metodo operativo dei Controller. Questi ultimi hanno 3 tipi di registro

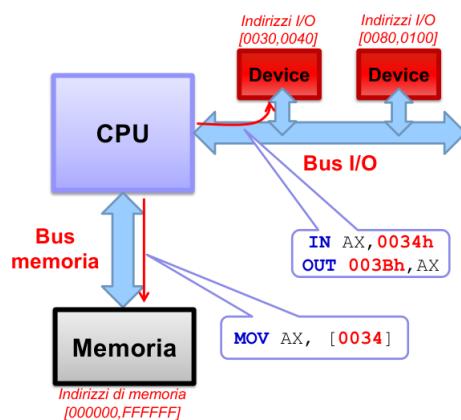
- **Stato:** il driver del dispositivo legge il registro di stato per determinare lo stato della periferica (es: Trasferimento in corso, Trasferimento completato)
- **Controllo:** il driver del dispositivo scrive sul registro di controllo per impartire comandi alla periferica (es: Inizia trasferimento ecc.)
- **Dato:** il driver del dispositivo scrive sul registro dato, per prelevare/inviare dati alla periferica

Ora vediamo in che modo il driver riconosce i cambiamenti di stato di un device di I/O. Ci sono due metodi principali

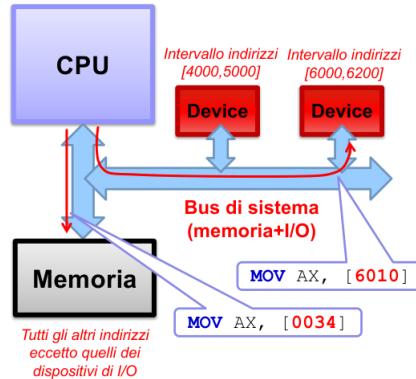
- **Polling:** Il driver accede ripetutamente in loop al registro di stato, fin quando non ci sono variazioni che indichino la disponibilità del dispositivo. Questa tecnica richiede di "bloccare" la cpu sull'attività di polling, impedendo di utilizzare la cpu per altre attività (**busy waiting**).
- **Interrupt-based:** Quando il dispositivo non è disponibile, la cpu viene utilizzata per altre attività, quando il dispositivo diventa disponibile, solleva un **interrupt**, passando il controllo dalla cpu alla ISR (**interrupt service routine**), una funzione all'interno del driver che si occupa di gestire l'interrupt

L'accesso ai registri del controller avviene in due possibili modi:

- **Port-mapped I/O:** il driver legge/scrive sui registri con istruzioni speciali della cpu (approccio dei sistemi più vecchi)



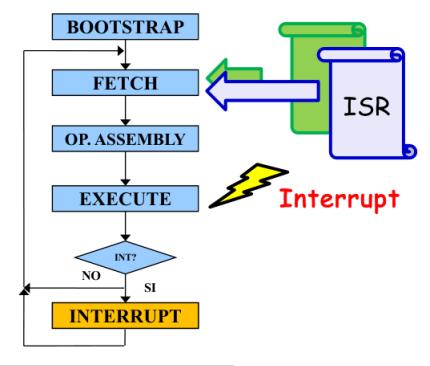
- **Memory-mapped I/O:** il driver legge/scrive sui registri utilizzando le stesse istruzioni e lo stesso spazio di indirizzamento per l'accesso in memoria (es. MOV), ogni device ha un suo intervallo di indirizzi



Una variante del memory-mapped I/O è il **DMA** (Direct Memory Access). Il DMA è un dispositivo che opera trasferimenti dati da e verso la memoria per conto della cpu, è vantaggioso per i trasferimenti di grossi blocchi di dati e per il minor numero di interrupt che la cpu deve gestire

## 2.2 Interrupt

Le interrupt permettono di alternare la cpu tra l'esecuzione dei programmi dell'utente e la gestione dell'I/O. Quando i dispositivi di I/O sollevano interrupt, la cpu si dedica (temporaneamente) alla loro gestione, ciò è alla base della multiprogrammazione e del time-sharing, in quanto aumentano l'utilizzo delle risorse e dunque l'efficienza. Il ciclo della cpu con gestione delle interrupt, differisce in uno step dal classico ciclo di fetch, decode, execute



Le interrupt possono essere sia **sincrone/asincrone** con il programma che **richieste/subite** dal programma

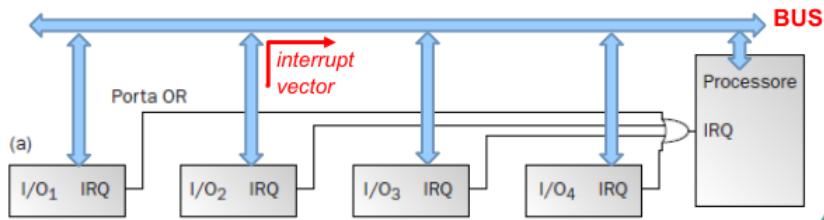
- **Interrupt asincrone:** sono semplici richieste di attenzione da parte dei dispositivi di I/O. Si possono verificare in qualsiasi momento durante l'esecuzione di un programma, e vengono gestite da una **interrupt service routine** fornita dal driver della periferica
- **Interrupt sincrone:** dette anche **traps/exceptions**, si verificano con specifici eventi del programma in esecuzione, sono sincrone rispetto a ciò che avviene nel programma (es: NULL pointer reference, divisione per zero, ecc). Spesso il sistema operativo risponde "uccidendo il programma"

Un caso particolare, sono le interrupt sincrone richieste da un programma. Queste interrompono volutamente l'esecuzione e vengono usate ad esempio nei tool di debug per impostare i breakpoint ecc... Di seguito una tabella riassuntiva dei tipi di interrupt.

Tipo	Sincrone / asincrone	User request / forzate
I/O device request	Asincrone	Forzate
Invoke operating system	Sincrone	User request
Instruction tracing	Sincrone	User request
Breakpoint	Sincrone	User request
Integer arithmetic overflow	Sincrone	Forzate
Floating-point arithmetic overflow or underflow	Sincrone	Forzate
Page fault	Sincrone	Forzate
Misaligned memory access	Sincrone	Forzate
Memory protection violations	Sincrone	Forzate
Using undefined instructions	Sincrone	Forzate
Hardware malfunctions (machine check exceptions)	Asincrone	Forzate
Power failure	Asincrone	Forzate

Inoltre le interrupt possono essere abilitate e disabilitate attraverso istruzioni macchina di STI/CLI, **set interrupt o clear interrupt**.

Più periferiche possono essere collegate alla stessa linea, **interrupt vector**, nel momento in cui viene lanciato un interrupt, la cpu identifica il dispositivo tramite un identificativo (**vector number**) fornito dal dispositivo, e successivamente ricerca nella **interrupt vector table** la routine di gestione dell'interrupt necessaria



Ricapitolando, nel dettaglio ciò che avviene per la gestione di un interrupt è il seguente

1. Un segnale di **interrupt request** (INT) viene inviato alla cpu
2. La cpu termina l'esecuzione dell'istruzione corrente
3. La cpu verifica la presenza del segnale INT ed invia un segnale di conferma (**ACK**) al dispositivo
4. La cpu salva sullo **stack di sistema** le informazioni necessarie a riprendere il programma interrotto (Program counter, Stack pointer, Program status)
5. La cpu seleziona la **ISR** corrispondente tramite il vettore di interrupt, e carica dal vettore il registro PC con l'indirizzo iniziale della ISR, e il registro PS
6. La ISR salva lo stato del processore su uno stack
7. La ISR gestisce l'interrupt
8. La ISR ripristina lo stato del processore
9. La cpu ritorna al controllo del processo interrotto

## 2.3 Protezione delle risorse hardware

I processi moderni presentano almeno due **stati di funzionamento**

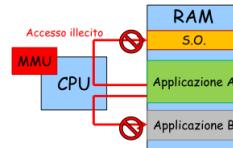
- User mode (non privilegiato)
- Kernel mode (supervisore/privilegiato)

Lo stato di funzionamento della cpu è tipicamente indicato nel Program Status (PS) da un bit S=1 (stato supervisore) o S=0 (stato utente). La cpu pone automaticamente S=1 all'avvio della ISR e il PS è ripristinato a 0 al termine della ISR (istruzione iret). Esempi di istruzioni privilegiate sono le istruzioni di STI e CLI, la modifica dei registri per la gestione delle interrupt, le istruzioni per gestione della memoria...

## 2.4 Protezione di memoria e delle memorie

Nei sistemi multiprogrammati e multiutente, più applicazioni condividono la memoria contemporaneamente. La coordinazione della memoria, viene gestita dalla cpu, e precisamente dalla **Memory Management Unit** (MMU), che protegge:

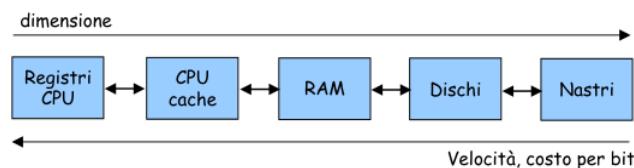
- Codice e dati del sistema operativo dalle applicazioni
- Codice e dati di una applicazione da altre applicazioni



Vediamo ora le tipologie di memoria esistenti:

- **Memoria Centrale**: spazio di memorizzazione volatile che può essere acceduto direttamente dalla CPU
- **Memoria Secondaria**: memoria non volatile con alta capacità di memorizzazione

Le memorie si organizzano in una gerarchia secondo 3 caratteristiche: velocità, costo, dimensione

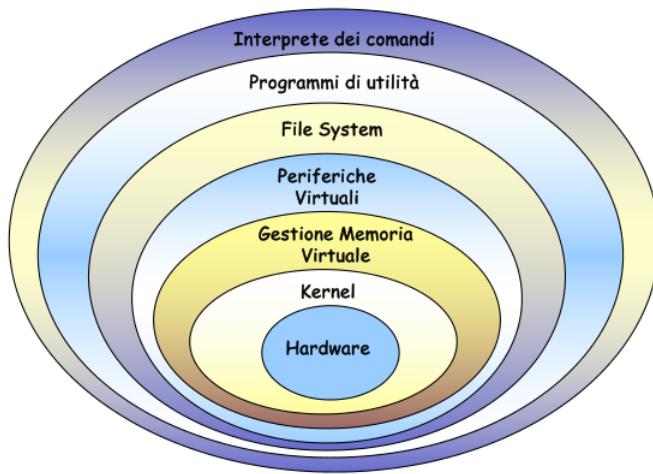


## 3 Architettura SO

Abbiamo già accennato alle funzionalità di un sistema operativo

- **Virtualizzazione delle risorse hardware**
- **Gestione e coordinamento**

I sistemi operativi sono costruiti in layer, ognuno con un preciso compito, di seguito una schematizzazione dei layer di un sistema operativo



### 3.1 Kernel

Il **kernel** è la parte del sistema operativo che risiede nella memoria principale, contenente le istruzioni fondamentali del SO. A livello kernel il sistema operativo si occupa di virtualizzare ed astrarre le risorse hardware:

- Possiede tante cpu quanti sono i processi (istanzia i processori virtuali)
- Non possiede meccanismi di interrupt
- Possiede istruzioni di sincronizzazione e scambio di messaggi tra processi che operano sui processori virtuali

### 3.2 Gestione della memoria

Il sistema operativo virtualizza anche la memoria nel layer di gestione della memoria. questo ha diversi benefit, quali

- garantisce la **protezione**
- consente di far riferimento a spazi di **indirizzi virtuali**
- consente in alcuni casi di ignorare se il programma e/o i dati siano fisicamente residenti in memoria centrale o su memoria di massa

### 3.3 Gestione periferiche

Al livello gestione periferiche, la virtualizzazione operata dal SO

- dispone di **periferiche dedicate** ai singoli processi
- maschera le caratteristiche fisiche delle periferiche
- gestisce parzialmente i malfunzionamenti delle periferiche

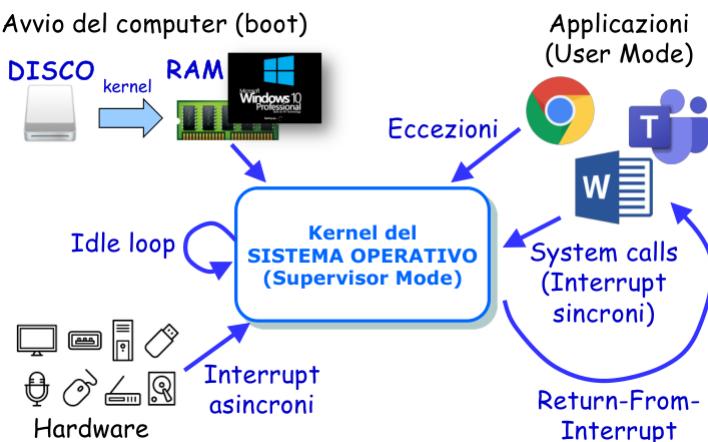
### 3.4 File system

Al livello del file system, la virtualizzazione operata dall'SO

- offre **strutture logiche** (es. file e directory) per memorizzare blocchi di dati
- controlla e gestisce gli accessi
- gestisce l'organizzazione fisica su memoria di massa

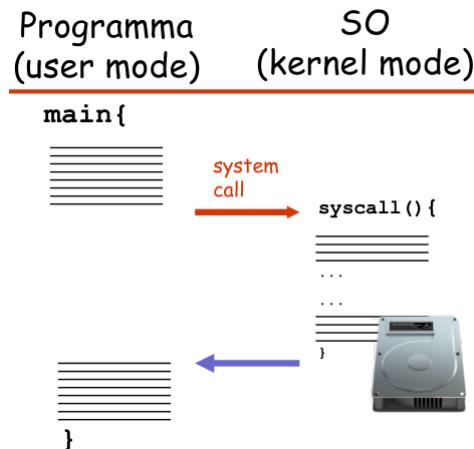
## 4 Invocazione sistema operativo

I sistemi operativi sono guidati dalle interruzioni (sincrone ed asincrone), gran parte del kernel viene eseguito come interrupt handler (ISR). Gli interrupt guidano l'avvicendamento dei processi. Ripercorrendo il processo di esecuzione del SO, si ha: caricamento in memoria centrale del kernel → Kernel (supervisor) → interrupt asincroni da parte dell'hardware → Interrupt sincroni e eccezioni da parte delle applicazioni.



### 4.1 System call

Per poter operare su una risorsa, i programmi devono fare una richiesta di servizio (**system call**) al sistema operativo (es: apertura di un file, comunicazione con un altro processo...). Una system call è una richiesta al SO di eseguire operazioni privilegiate per conto del programma chiamante

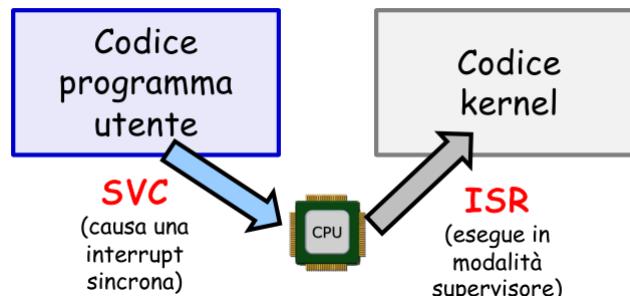


Una syscall corrisponde alla attivazione di una parte del kernel a favore del processo chiamante, e si dividono in diverse categorie:

- **Controllo dei processi:** Es. load, execute, allocate, mem, free mem
- **Manipolazione dei file:** Es. create, delete, open, close, read
- **Gestione dei dispositivi:** Es. request device, read, write
- **Informazioni di sistema:** Es. get time, set time
- **Comunicazione:** es. create connection, send, receive

Strutturare l'accesso alle risorse tramite syscall, ha come vantaggio che l'unico processo in esecuzione in modalità supervisore è il kernel. Ciò implica che le applicazioni potenzialmente contenenti bug o malevole, eseguono in modalità utente, ed è il kernel ad effettuare i controlli di sicurezza prima di concedere l'accesso alle risorse.

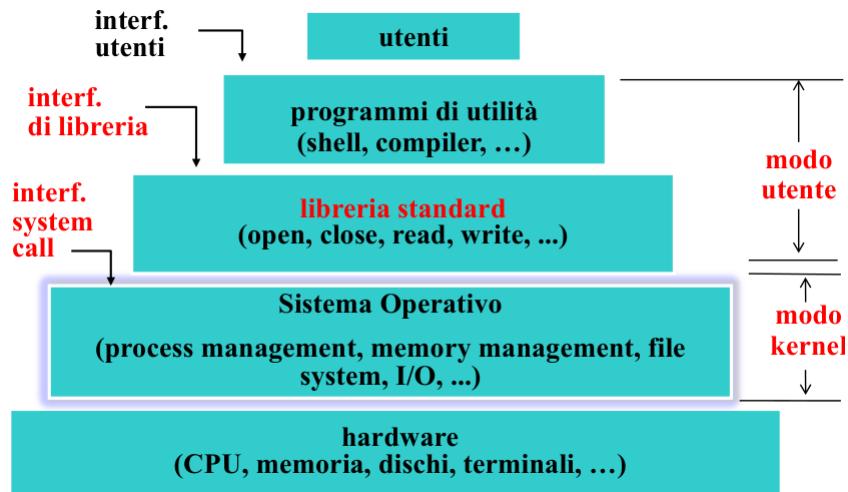
Il programma che invoca la syscall, non è autorizzato ad invocare istruzioni di salto (JSR), ma esegue la syscall tramite un'istruzione speciale della cpu spesso chiamata **supervisor call SVC**. Quest'ultima innesca una interrupt sincrona e l'esecuzione di una ISR che esegue le operazioni privilegiate



Il passaggio di parametri di una syscall avviene spesso tramite registri della cpu. In linux x86 si usa il registro **EAX**, in cui il programma ritorna il valore al termine della esecuzione.

Tipicamente, i linguaggi di programmazione forniscono **routine di interfaccia** che trasformano una tradizionale chiamata di procedura in una SVC. Questo facilita la chiamata e lo scambio dei parametri

- **libc** in c, **namespace std** e la libreria **Boost** in c++...



nel layer **libreria standard** risiedono le funzioni di libreria quali printf per stampare messaggi, e le funzioni di libreria interagiscono con il layer **sistema operativo** traducendo le funzioni in syscall. Ad esempio la funzione printf esegue una syscall di tipo write()

## 4.2 Tipi di architetture SO

### 4.2.1 Monolitiche e modulari

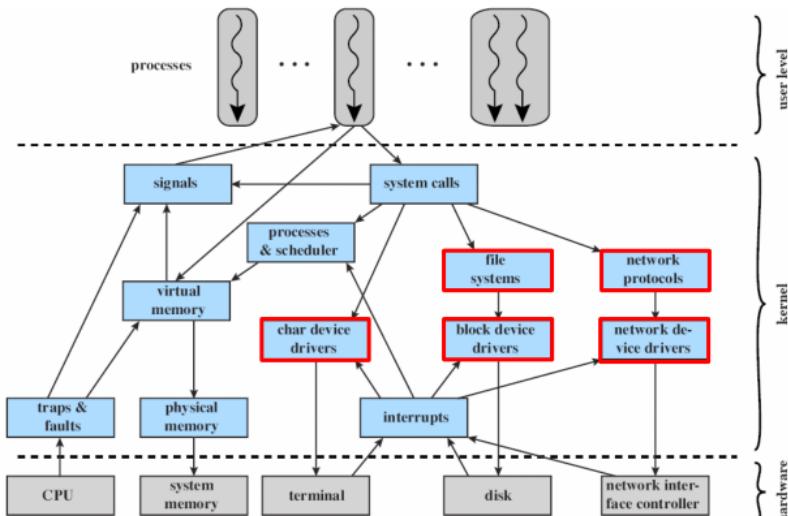
I primi sistemi operativi erano costituiti da un unico programma, senza particolari suddivisioni, che raccoglieva funzioni scritte in linguaggio macchina corrispondenti alle syscal. Qui nasce la distinzione tra due tipi di architetture del SO

- **Architetture monolitiche:** architettura del sistema operativo in cui l'intero sistema operativo funziona nello spazio del kernel
- **Architetture modulari:** architettura che divide il codice del SO in più moduli (Interfaccia e corpo)

I vantaggi di un'architettura modulare su una monolitica sono principalmente 2:

- La modifica di un modulo ha un impatto ridotto sul resto del SO
- Si possono caricare in memoria solo i driver necessari

Linux è un esempio di architettura modulare.



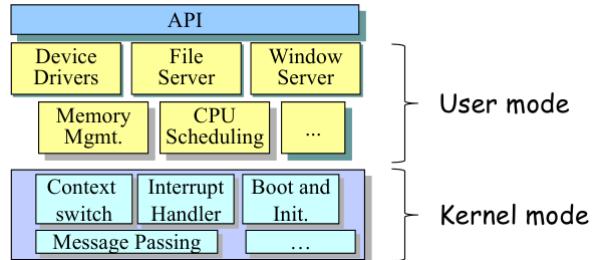
Un importante distinzione tra i moduli dell'SO riguarda i moduli **user-space** e i moduli **kernel-space**. I moduli user-space eseguono come fossero applicazioni (modo non-privilegiato), mentre i moduli kernel-space eseguono in maniera privilegiata (supervisor mode), e le chiamate da moduli user-space sono molto più veloci delle chiamate a sistema.

**N.B.** Un bug o una vulnerabilità nei moduli kernel può danneggiare l'intero sistema

La maggior parte del SO, è costituito da **device drivers** che sono tipicamente moduli kernel caricati in memoria su domanda e si occupano della gestione dei singoli dispositivi (ISR)

#### 4.2.2 Microkernel

Un ulteriore variante sull'architettura modulare, è l'architettura a microkernel. In quest'ultima si implementano solo i meccanismi essenziali nel kernel, e le politiche di gestione sono implementate all'esterno del kernel in processi di sistema



Un esempio di architettura a microkernel è MINIX 3 (risorse gestite da processi sistema detti server, il processo utente detto client invia ad un server una richiesta)

## 5 Processi

Iniziamo definendo cos'è un processo e cos'è un programma:

- **Programma:** è la codifica di un algoritmo in un linguaggio di programmazione, che ne rende possibile l'esecuzione da parte di un elaboratore
- **Processo:** è l'unità base di esecuzione del SO, identifica le attività dell'elaboratore relative ad una specifica esecuzione di un programma. Si tratta di un entità dinamica costituita da programma e contesto di esecuzione

### 5.1 Stato di un processo

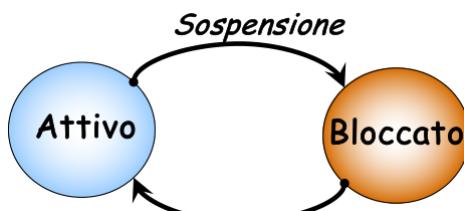
Lo **stato** di un processo rappresenta un'astrazione del suo contesto di esecuzione. I processi sono soggetti a transizioni di stato dovute a:

- l'attività corrente del processo
- eventi esterni asincroni con la sua esecuzione

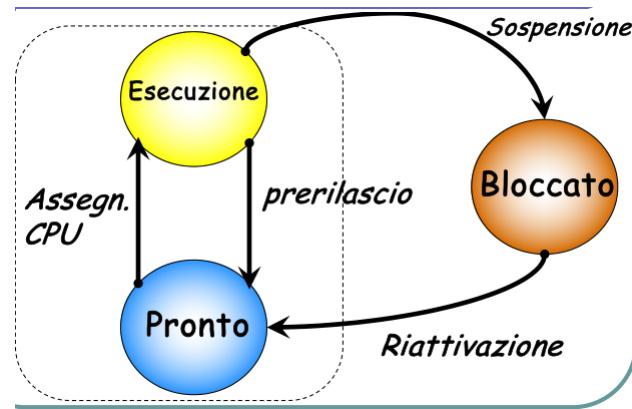
#### 5.1.1 Modello a 2/3 stati

In prima approssimazione, un processo può essere caratterizzato da due stadi:

- **Attivo:** il processo è in esecuzione sulla cpu
- **Bloccato:** il processo è in attesa di un evento (I/O, sincronizzazione...)



Il problema sorge nel caso in cui si ha 1 cpu e molti processi attivi, infatti si presuppone che vi siano tante cpu fisiche quanti processi. Per questo nasce il modello a 3 stati, in cui si ha una distinzione tra processo pronto ad eseguire, e processo in esecuzione

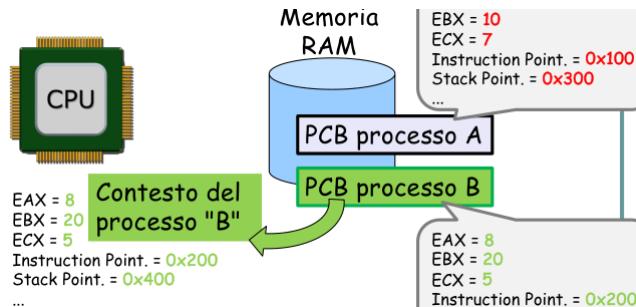


### 5.1.2 Context switch

Si definisce **context switch** l'insieme di operazioni eseguite dal SO per il prerilascio di un processo. Il contesto di un processo include le informazioni contenute nei registri del processore

- Program counter, Stack counter, Registri general-purpose, Registri di gestione della memoria...

I dati contenuti in questi registri, vengono salvati in una struttura dati del SO, definita **process control block** (PCB) che risiede nella memoria ram. Nel context switch, la cpu salva dunque in memoria il contesto di un processo, e preleva dalla memoria il contesto di un secondo processo pronto all'esecuzione



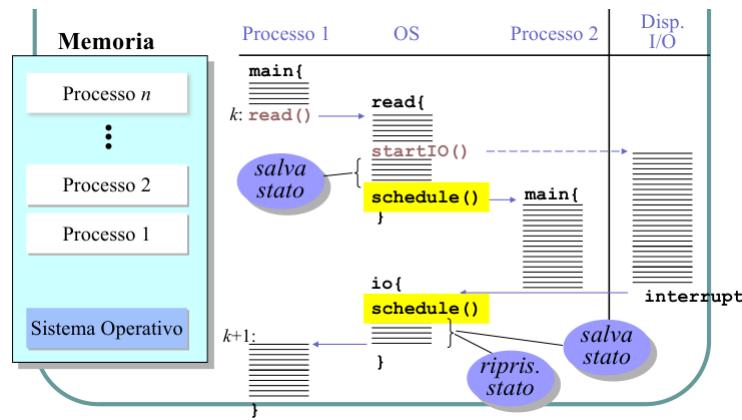
Il cambio di contesto avviene a seguito di diverse operazioni

- **Timeout**: termina il tempo assegnato al processo
- **Syscall**: il processo richiede un servizio al SO
- **Memory fault**: il processo accede ad un indirizzo di memoria non valido
- **Trap**: cpu exception (può causare la terminazione del processo)
- **I/O**

Ricapitolando ciò che avviene all'interno della cpu in un context switch:

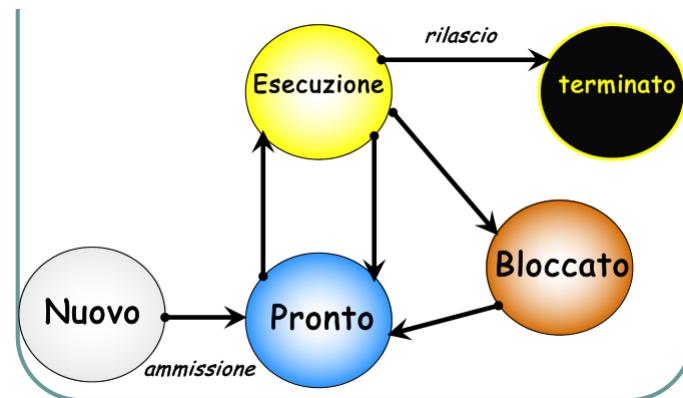
- **Salvataggio stato**: la cpu salva una copia del contesto del processo nel PCB
- **Scheduling cpu**: la cpu scegli il prossimo processo tra quelli pronti per l'esecuzione
- **Ripristino stato**: la cpu copia il contesto del processo scelto dal suo PCB ai registri della CPU

Ma come avviene la scelta del processo da eseguire in caso di più processi nello stato "pronto"? La scelta viene eseguita dallo **scheduler**, che fa generalmente parte del kernel del SO

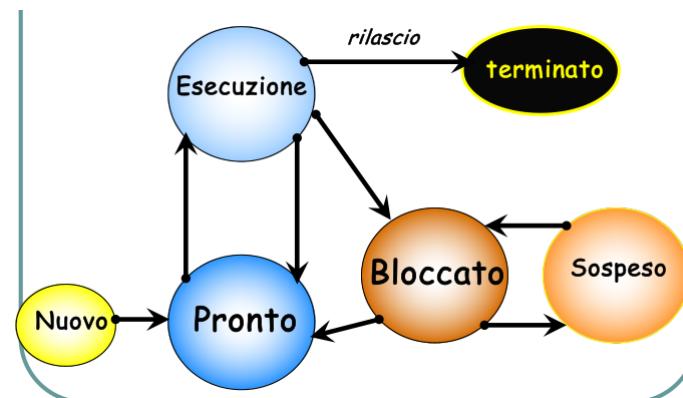


### 5.1.3 Modello a 5/6 stati

Vediamo ora una versione ancora più complessa della gestione dei processi, il modello a 5 stati. Questo modello introduce lo stato "nuovo" che corrisponde alla creazione di un nuovo processo, e lo stato "terminato" che corrisponde alla terminazione di un processo.

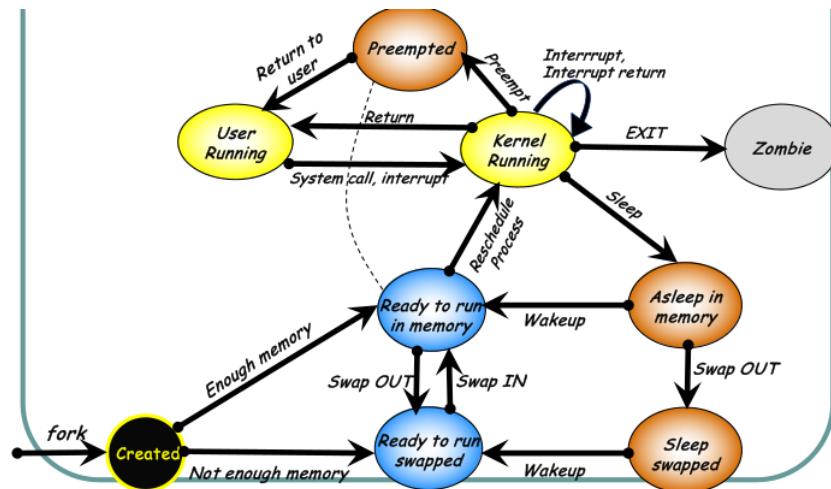


Un ulteriore versione per i SO che prevedono la possibilità di spostare temporaneamente un processo dalla RAM alla memoria secondaria (**swapping**) prevede un ulteriore stato: "sospeso"



### 5.1.4 Processi in Unix

Il sistema operativo Unix implementa lo stato dei processi come riassunto nello schema sottostante:

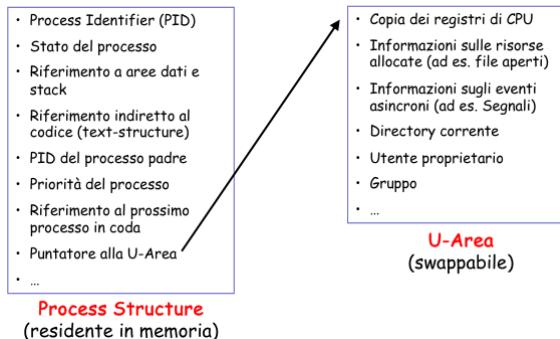


Si aggiunge un nuovo stato, lo stato "Zombie", in cui un processo ha terminato ma non può ancora essere terminato perché la sua immagine di memoria è ancora necessaria. Ex. (un processo (padre) avvia un altro processo (figlio), il figlio termina prima del padre e diventa "zombie" finché il padre non ne raccoglie lo stato di terminazione)

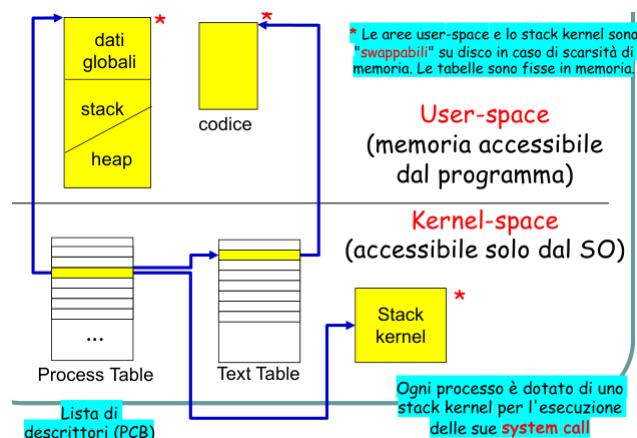
Le informazioni sui processi in linux, sono ottenibili da shell tramite due comandi principali:

- **top**: Una lista di tutti i processi in esecuzione sul calcolatore
- **ps aux**: stampa un'istantanea dei processi con l'identificatore (PID: process id), lo stato del processo, il terminale assegnato, il tempo di cpu usato, il nome del processo

Ad ogni processo come detto in precedenza viene assegnata una struttura dati PCB, tutti i PCB sono raggruppati nella **process table**. In unix il pcb è implementato nel seguente modo



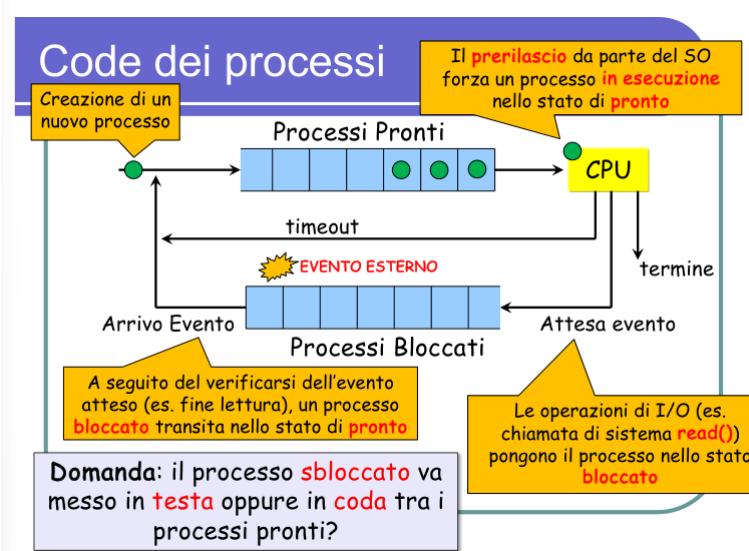
La process structure contiene informazioni che dovrebbero essere accessibili al kernel, la u-area contiene le informazioni che dovrebbero essere accessibili al processo solo quando in stato di esecuzione



### 5.1.5 Code dei processi

Il sistema operativo tien traccia dei processi utilizzando delle code:

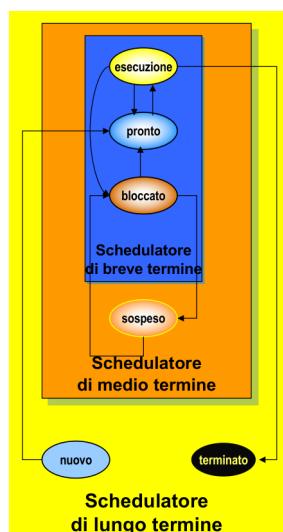
- Una o più code per i **processi pronti**
- Uno o più cide dei processi bloccati



## 6 Scheduling

Lo **scheduler** è la parte del SO preposta all'assegnazione delle risorse a favore dei processi. Un algoritmo di scheduling seleziona il processo assegnatario da una coda in base a vari criteri. Esistono diverse tipologie di scheduler:

- Scheduler a **breve termine** o scheduler della cpu
- Scheduler a **medio termine** o scheduler di swap (swapper)
- Scheduler a **lungo termine** o scheduler di job



### 6.1 Scheduler a lungo termine

Lo scheduler a lungo termine determina quali processi caricare nella coda dei processi pronti del sistema (nuovo → pronto). I possibili criteri di scelta per la posizione nella coda dei processi sono

- FIFO: first in first out
- Priorità: processi con alta priorità

- Tempo di esecuzione presunto
- Requisisti di I/O
- Tempo presunto di CPU

Per aumentare l'efficienza di utilizzo delle risorse, lo scheduler ammette nel sistema un numero comparabile di processi CPU-bound e I/O-bound. Per processi **CPU-bound** si intendono processi con poche chiamate a sistema, che tendono ad occupare la CPU per lunghi periodi se il SO non li interrompe e sono tipicamente applicazioni batch, calcolo numerico... I processi **I/O bound** sono quei processi che fanno frequenti chiamate di sistema, usano brevemente la CPU per poi rimettersi in attesa di I/O, tipicamente sono i programmi interattivi quali browser, editor di testo ecc

## 6.2 Scheduler a medio termine

Ha il compito di trasferire temporaneamente processi (o parte di essi) dalla memoria centrale alla memoria di massa (Bloccato → sospeso) per liberare spazio nella memoria centrale e rendere possibile il caricamento di altri processi. Ha come obiettivo quello di gestire efficientemente la memoria.

## 6.3 Scheduler di breve termine

Ha il compito di scegliere un processo pronto a cui assegnare la cpu (pronto → esecuzione). È lo scheduler attivato più frequentemente, quando il processo in esecuzione si interrompe

## 6.4 Criteri, Starvation, Preemption

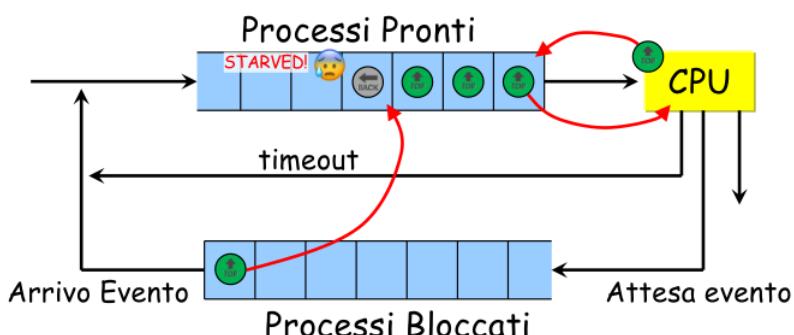
### 6.4.1 Criteri di prestazione

Iniziamo distinguendo i parametri con cui si valuta la prestazione di un algoritmo di scheduling:

- **Criteri user-oriented:** la percezione dell'utente e dei singoli processi
  - **Tempo di risposta:** Il tempo tra l'invio di una richiesta dell'utente fino all'inizio della sua elaborazione
  - **Tempo di turnaround:** Il tempo tra l'invio di un processo al sistema e la sua teminazione
  - **Deadlines:** Se l'utente indica una scadenza di completamento, si deve massimizzare la percentuale di scadenze rispettate
- **Criteri system-oriented:** Il punto di vista delle risorse e dell'amministratore di sistema
  - **Throughput:** produttività in termini di numero di processi completati per unità di tempo (es, 10 esecuzioni/ora)
  - **Utilizzo della cpu:** percentuale di tempo in cui la cpu risulta occupata
  - **Fairness:** a meno di indicazioni da parte dell'utente, tutti i processi dovranno essere trattati nello stesso modo e nessuno deve subire una attesa infinita (**starvation**)

### 6.4.2 Starvation

Un processo si definisce in **starvation** se può attendere per un tempo arbitrariamente lungo "attesa infinita". Questo non è da confondere con attesa "infinita", in quanto il processo ha la garanzia di essere eseguito, ma non nel prossimo futuro.



La starvation si verifica negli algoritmi che danno maggiore priorità a specifici processi, e lo scheduler tende a scegliere sempre i processi a priorità alta, lasciando gli altri in attesa.

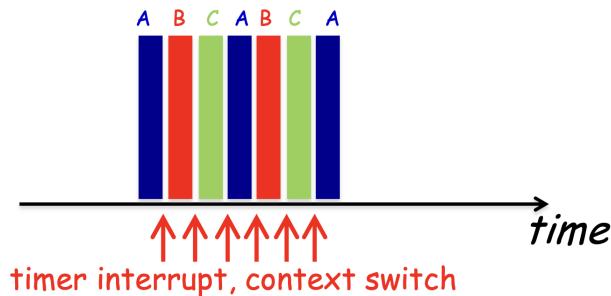
#### 6.4.3 Preemption

Come accennato in precedenza, la cpu ha un metodo per sospendere l'esecuzione di un processo, questo metodo si chiama preemption. Anche gli algoritmi di scheduling si suddividono in algoritmi preemptive e non preemptive

- **Algoritmi non-preemptive:** un processo in esecuzione rimane in tale stato finchè non si sospenderà volontariamente.



- **Algoritmi preemptive:** un processo in esecuzione può essere interrotto anche se non effettua syscall (viene forzato nello stato pronto)



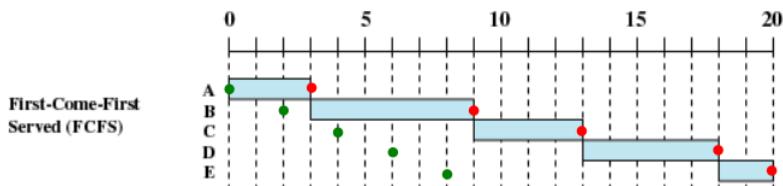
## 6.5 Algoritmi

Prima di vedere i principali algoritmi, ricapitoliamo due parametri importanti nell'analisi di questi ultimi

- **Ts:** tempo di servizio, ossia l'effettivo tempo di esecuzione sulla cpu
- **Tr:** tempo di turnaround, ossia il tempo trascorso dall'inserimento in coda alla fine dell'esecuzione
- **Ts/Tr:** indicatore di ritardo "relativo"

#### 6.5.1 First-Come-First-Served FCFS

Quando un processo termina viene eseguito il primo processo in ordine di arrivo (**non-preemptive**)



**Turnaround**  $TrA=3, TrB=7, TrC=9, TrD=12, TrE=12$

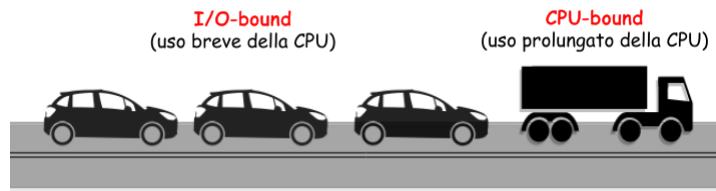
**Tr/Ts**  $TrA/TsA=1, TrB/TsB=1.17, TrC/TsC=2.25$

$TrD/TsD=2.40, TrE/TsE=6.00$

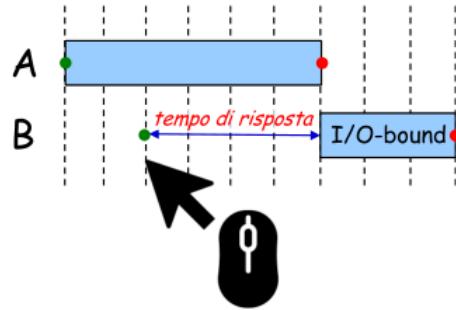
Si nota come il processo E, spende nel sistema 6 volte il tempo di esecuzione, questo è dovuto ad un tempo di turnaround molto elevato, che si traduce in un ritardo di esecuzione percepibile dall'utente.

Questo tipo di algoritmo non risente di problemi di **starvation**, ma può risentire dell'**effetto convoglio**

Per effetto convoglio si intende quell'effetto per cui un processo CPU-bound, occupa la CPU per un tempo eccessivo, a scapito dei processi I/O-bound

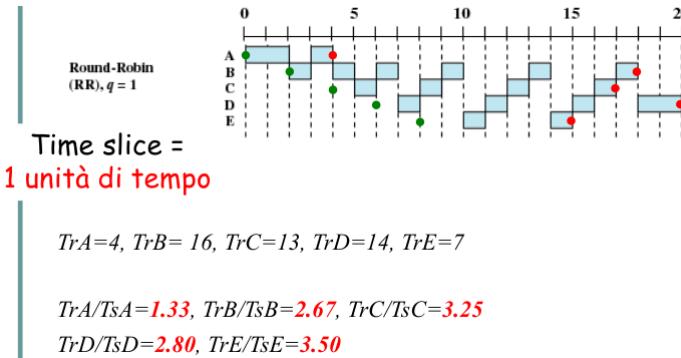


Questo genere di effetto causa una riduzione dell'utilizzo delle risorse (sia CPU che I/O), in quanto i processi I/O-bound rimangono fermi a lungo, lasciando i dispositivi inutilizzati e quando eseguono, la CPU è sotto utilizzata. Inoltre come già detto, l'utente percepisce latenza tra il comando di esecuzione e l'esecuzione effettiva in quanto il tempo di risposta è elevato



### 6.5.2 Round-Robin

Si tratta della versione **preemptive** del FCFS, mediante l'impiego di un timer. Viene assegnato un tempo massimo di esecuzione detto **time slice**, o **quanto di tempo** ad ogni processo



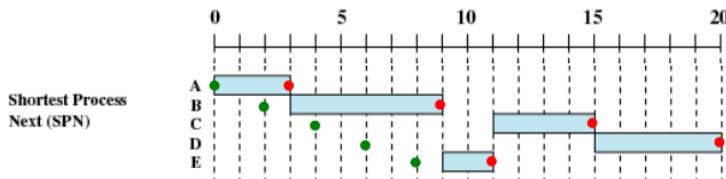
Nel caso del processo E, l'algoritmo round-robin, dimezza il ritardo effettivo.

Le prestazioni del sistema dipendono dalla scelta della durata del time slice, aumentando il time slice tende a trasformarsi in FCFS, ma diminuendo il time slice, aumenta la frequenza dei cambi di contesto fra processi, e dunque l'overhead dell'SO

Idealmente il time slice, dovrebbe essere appena superiore alla durata dei processi I/O-bound, i valori tipici sono da 10 a 100ms.

### 6.5.3 Shortest Process Next

L'algoritmo shortest process, esegue prima il processo con il minor tempo di esecuzione stimato, riduce dunque i tempi di risposta per i processi brevi, ma aumenta i tempi di risposta per i processi lunghi aggiungendo la possibilità di starvation. Ne esiste sia una versione non-preemptive **Shortest Process Next** che una versione preemptive **Shortest Remaining Time**



$TrA=3, TrB=7, TrC=11, TrD=14, TrE=3$

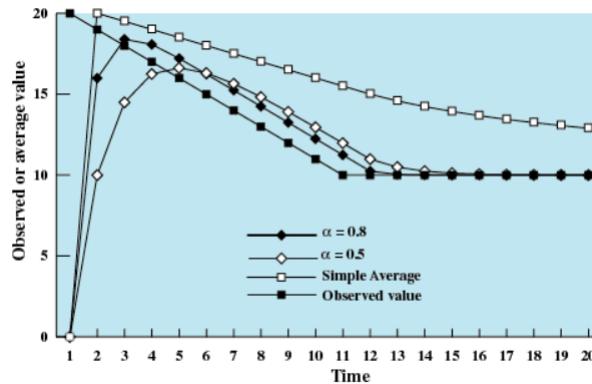
$TrA/TsA=1, TrB/TsB=1.17, TrC/TsC=2.75$

$TrD/TsD=2.80, TrE/TsE=1.50$

Come si nota dal processo D, c'è possibilità di starvation

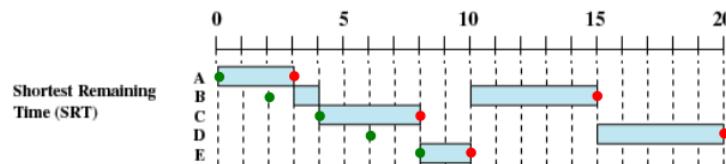
SPN, richiede di conoscere o almeno stimare il tempo di servizio di ciascun processo su indicazione del programmatore. La stima si basa sulla media dei periodi di esecuzione passati.

La media solitamente è aritmetica, ma nei sistemi più recenti si opta per una media esponenziale pesata, in modo da dare più peso ai tempi di esecuzione recenti in quanto più rappresentativi del comportamento futuro del processo. Di seguito una rappresentazione della stima del tempo di esecuzione e l'effettivo tempo utilizzando entrambe le medie



#### 6.5.4 Shortest Remaining Time

Si tratta della versione preemptiva dello SPN. La prelazione può avvenire subito quando entra un nuovo processo, implicando che i processi brevi non aspettano mai



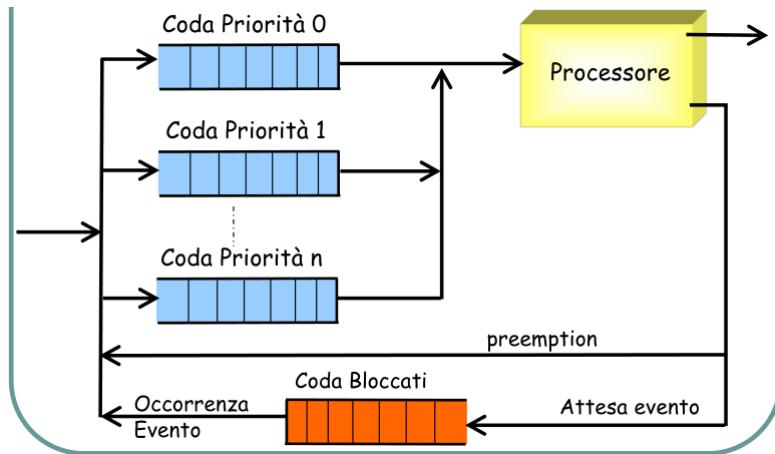
$TrA/TsA=1, TrB/TsB=2.17, TrC/TsC=1$

$TrD/TsD=2.80, TrE/TsE=1$

Rispetto al SPN, il processo C prelaziona il processo B. Gli algoritmi SPN e SRT sono di difficile utilizzo perché è difficile fare stime precise

#### 6.5.5 Scheduler a Priorità

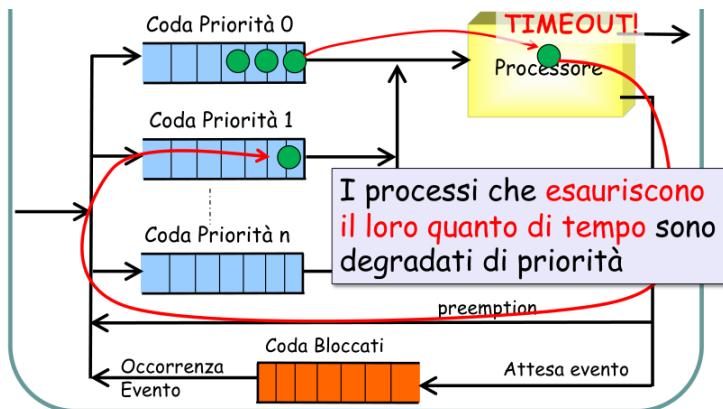
Lo scheduler a priorità (multilevel) semplifica, raggruppando i processi in livelli di priorità. I processi a stesso livello di priorità vengono gestiti dal round-robin.



I processi a bassa priorità eseguono quando si esauriscono quelli ad alta priorità, e la coda di esecuzione è gestita in round-robin.

#### 6.5.6 Multilevel Feedback

Si tratta di una variante dello scheduler a priorità in cui per non penalizzare i processi a bassa priorità, si stabilisce un sistema di feedback. Si assegna ai processi una priorità dinamica penalizzando i processi CPU-bound a favore di quelli I/O-bound...



Tipicamente ogni coda è gestita da round-robin, eccetto l'ultima che è caratterizzata da FCFS (es. i processi in background). Favorisce i processi più corti, e i processi a minor priorità sono compensati con un time slice più lungo

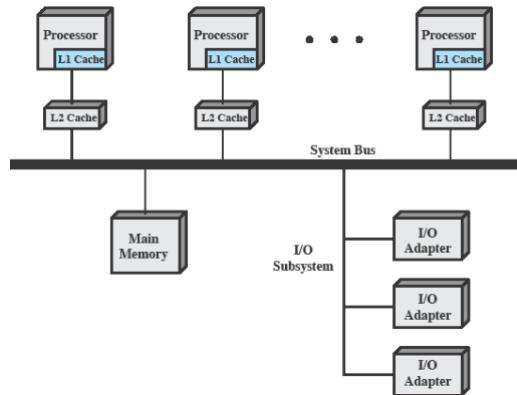
$$\text{Coda } i\text{-esima} = 2^i \text{ unità di tempo}$$

Dopo un tempo massimo, si fa ritornare un processo dalla coda inferiore alla coda ad alta priorità. Si noti che può causare starvation.

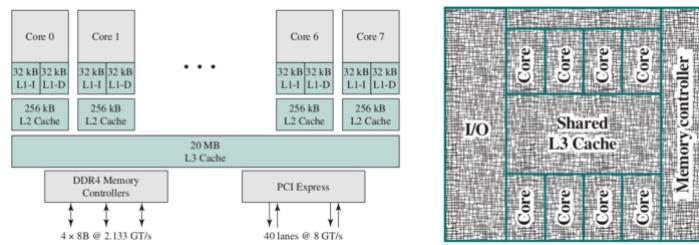
## 7 SMP e Scheduling multiprocessore

Iniziamo introducendo la definizione di alcuni concetti chiave

- **Symmetric Multi-Processing (SMP)**: più CPU identiche sono collegate alla stessa memoria condivisa, e hanno pieno accesso ai dispositivi di I/O
- **Multi-Core CPU**: più cpu (**core**) sono sullo stesso chip
- **Hyperthreading**: uno stesso core ha risorse multiple (ALU, registri, etc) e può eseguire più programmi contemporaneamente



Un esempio di cpu multicore è l'Intel Core i7-5960x. Con la seguente architettura e layout su chip



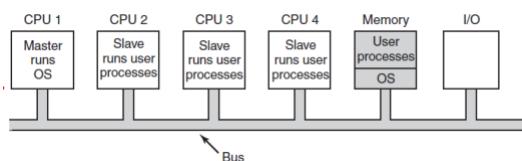
Oltre i core fisici implementati sul chip, il sistema operativo "vede" diversi **core virtuali**. Ognuno di questi esegue un programma come se fosse una cpu fisica, ma in realtà i core virtuali eseguono su uno stesso core fisico che ne fornisce le risorse (es: ALU)

## 7.1 SO per SMP

Ci sono alcune scelte di design legate ai sistemi SMP, quali la scelta di dove eseguire l'algoritmo di assegnazione, e l'implementazione dell'assegnazione dei processi ai processori

### 7.1.1 MASTER/SLAVE

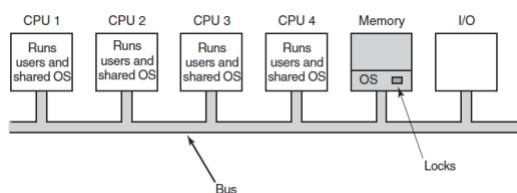
Nell'approccio **MASTER/SLAVE**, il kernel esegue su un unico processore detto **master**, responsabile dello scheduling, tutti gli altri processori **slave** possono eseguire solo processi utente e inoltrano le syscall fatte dai processi al master.



Quali sono i vantaggi e dove risiedono le problematiche di questo tipo di approccio? Se pur facilmente implementabile, questo approccio centralizza troppo il master che costituisce un single point of failure ed un bottleneck per le prestazioni

### 7.1.2 PEER

Nell'approccio **PEER**, il kernel può eseguire su tutti i processori, anche contemporaneamente, ed ogni processore gestisce autonomamente lo scheduler.



Si tratta di un approccio di più complessa implementazione, ma che tuttavia rimuove le negatività dell'approccio MASTER/SLAVE

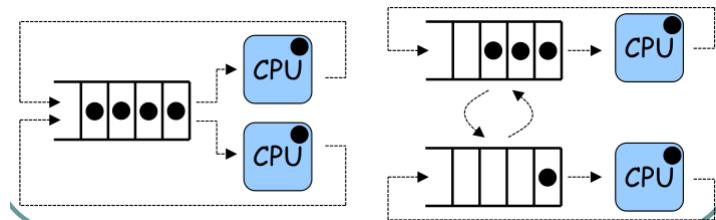
### 7.1.3 Assegnazione dei processi

L'assegnazione dei processi può seguire due diverse dinamiche:

- **Assegnazione statica:** ogni processo o thread viene assegnato permanentemente ad uno dei processori
  - Ogni processore ha una propria coda di processi
  - L'assegnazione viene fatta una volta e non può mutare
- **Assegnazione dinamica:** durante la sua vita un processo può eseguire su processori differenti

È chiaro come uno di questi sia nettamente l'approccio superiore in termini di prestazioni, infatti l'assegnazione statica, risente di forti problemi di underusage, basti pensare a un core sovraccaricato di lavoro e un core che ha terminato la propria coda. Passando all'assegnazione dinamica, ne esistono due versioni.

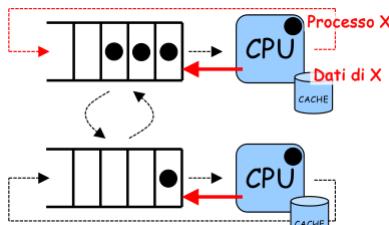
- **Load sharing:** Una sola coda dei processi pronti condivisa
- **Dynamic load balancing:** Più code di processi pronti, una per processore, in cui i processi possono essere spostati da una coda all'altra



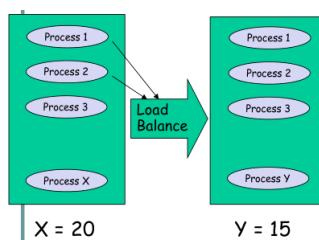
Idealemente, avere una coda unica garantisce il maggior utilizzo possibile della CPU, dato che in code separate c'è ancora un rischio minimo che una coda diventi vuota. Tuttavia avere una sola coda crea due problemi.

- La coda occupa una regione di memoria condivisa che deve essere protetta da accessi concorrenti (**mutua esclusione**)
- Non garantisce che un processo riprenda l'esecuzione sullo stesso processore (uso poco efficiente della cache dei processori)

Con code multiple, ogni CPU accede a una coda separata, evitando bottleneck, e i processi riutilizzano la stessa CPU a meno di load balancing. Abbiamo introdotto intuitivamente il concetto di **CPU affinity**, secondo il quale un processo tende ad eseguire più rapidamente se si riutilizza sempre lo stesso core.



Spostare un processo da un processore all'altro danneggia il principio di località, e inoltre il load balancing ha un impatto negativo sui processi migrati. Risulta necessario dunque migrare i processi con minore affinità. Vediamo l'approccio al load balancing in Linux.



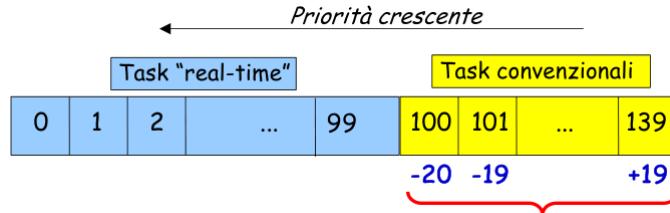
Linux adatta il dynamic load balancing:

- Il load balancing viene attivato periodicamente, o quando una coda è vuota
- Vengono estratti dalla lista i task che non stanno eseguendo e non sono cache-hot
- L'algoritmo termina quando la runqueue con il maggior numero di task non eccede del 25% le altre

## 8 Scheduling in Linux

Analizziamo il processo di scheduling in Linux. Uno dei punti centrali è il concetto di **task** (flusso di esecuzione), si tratta dell'unità fondamentale dello scheduler in Linux. Viene utilizzato per rappresentare sia processi che threads e ciascun task è identificato da un **process id** (PID). Ad ogni task viene attribuita una priorità che ne determina sia l'ordine di scheduling, che il quanto di tempo assegnato. Esistono due categorie di Task

- **Real time tasks:** richiedono una risposta garantita entro un dato intervallo di tempo
- **Task convenzionali:** la maggior parte dei programmi utente



La priorità in linux è rappresentata da un intero tra 0 e 39, e l'utente può sommare un valore di correzione tra (-20) e (+19)

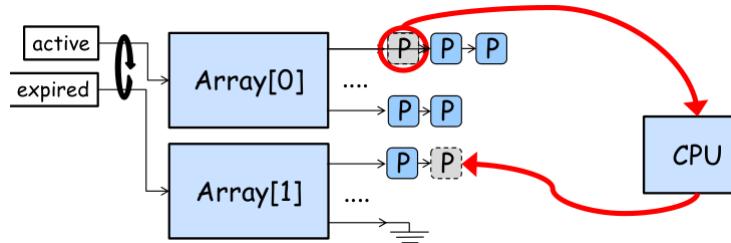
### 8.1 Algoritmi di scheduling

#### 8.1.1 Scheduler O(1)

Si tratta di un algoritmo basato sul modello classico del multilevel feedback, e ha come obiettivi quello di ottenere un overhead di sistema **costante**, un buon compromesso tra fairness e responsiveness e l'utilizzo efficace con le architetture SMP.

**N.B.** Il tempo impiegato per la scelta di un processo da eseguire è costante, fare una scelta tra 10 o 1000 task impiega sempre lo stesso tempo

In questa tipologia di algoritmo, viene assegnata una runqueue (struttura dati contenente le code di processi in attesa) a ciascun processore



Per ridurre il fenomeno della starvation, all'interno della runqueue, sono introdotti 2 array di code:

- **Active:** contiene i task che non hanno ancora consumato il quanto di tempo a loro assegnato
- **Expired:** contiene i task che hanno eseguito per un intero quanto di tempo

Quando tutti i task hanno esaurito il proprio time slice, si invertono i due array e si inizia un nuovo round.

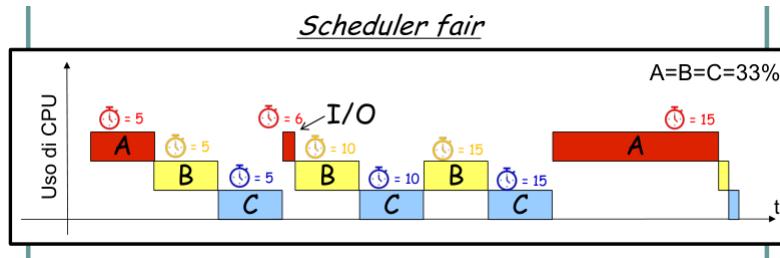
Una parentesi importante va aperta in merito all'assegnazione delle priorità in linux. Lo scheduler fa una distinzione tra due tipi di priorità

- **Priorità statica:** coincide con il nice value e determina la durata del time slice assegnato.
- **Priorità dinamica:** inizialmente pari al nice value, varia durante l'esecuzione e determina la coda in cui è inserito

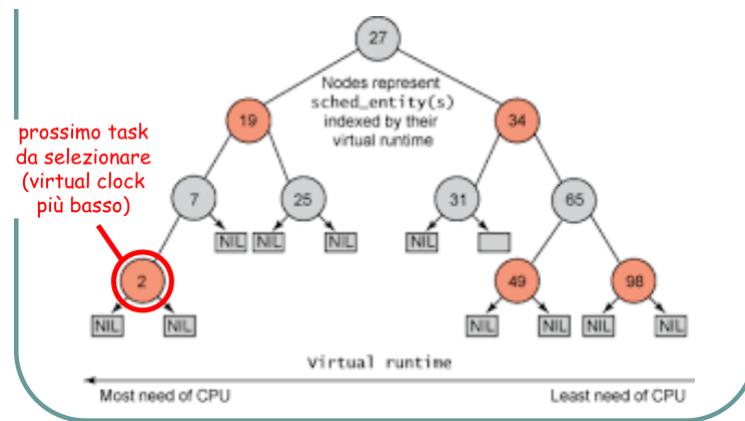
La priorità dinamica varia di un bonus di  $\pm 5$  dalla priorità statica. Se un task passa da blocked a running, il suo tempo di attesa viene aggiunto ad un contatore detto **tempo di sleep**, se passa da running a blocked il suo tempo di esecuzione viene sottratto al tempo di sleep. Un maggior tempo di sleep implica un maggior bonus e le task I/O bound sono identificate e premiate.

### 8.1.2 CFS scheduler

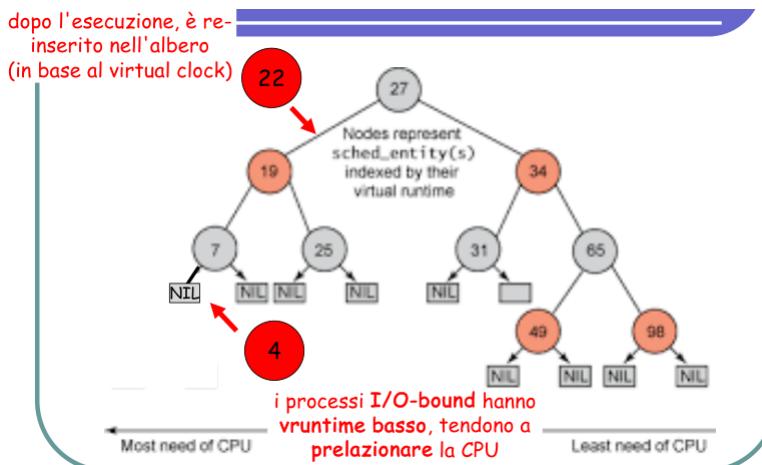
Il completely fair scheduler (CFS), rispetto all' $O(1)$ , garantisce la fairness, rende lo scheduling più controllabile, supporta il group scheduling ma ha maggiore onere computazionale  $O(n \log n)$ . Nel fair scheduling viene concesso a ciascun processo un time slice proporzionale alla sua priorità.



A ciascuna task viene assegnato un **vruntime**, pari al minimo dei vruntime degli altri processi in esecuzione al momento della creazione del nuovo processo. Durante l'esecuzione di un task, il suo vruntime è incrementato peridicamente. La selezione dei task avviene da una struttura dati chiamata **red-black-tree**.



Il cfs sceglie sempre il task con vruntime più basso, e l'algoritmo di ricerca del valore più piccolo è  $O(\log n)$ . Dopo l'esecuzione e l'obbligatorio incremento del vruntime, viene reinserito nell'albero con vruntime maggiorato e l'algoritmo si ripete.



Il time slice non è un valore fissato a priori ma viene determinato di volta in volta.

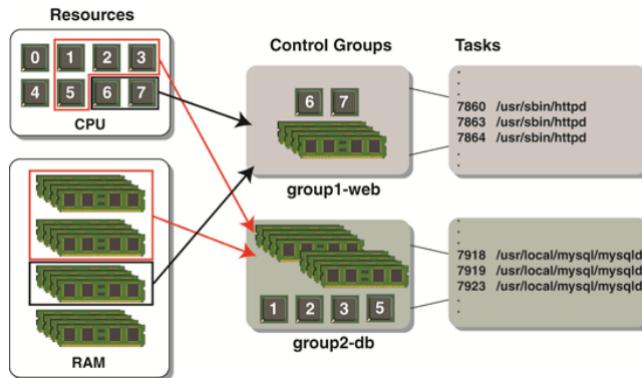
$$\text{timeslice} = \frac{\text{targeted latency}}{\text{numero di task}}$$

Dove la **targeted latency** è il tempo massimo entro il quale tutti i processi pronti devono avere l'opportunità di essere eseguiti almeno una volta. Nel caso in cui la task abbia priorità diversa, il time slice è calcolato moltiplicando la target latency per priorità task/totale priorità e dividendo tutto per il numero di task. Abbiamo dunque inserito un upper bound al time slice, il lower bound è invece dato dal parametro **minimum granularity**.

## 8.2 Cgroups

Il kernel Linux permette di raggruppare processi in un **control group** (cgroup), per diversi scopi

- Limitare l'uso delle risorse di un gruppo
- Priorizzare l'accesso di un gruppo alle risorse
- Tracciare l'uso delle risorse
- Controllare lo stato delle task con un solo comando



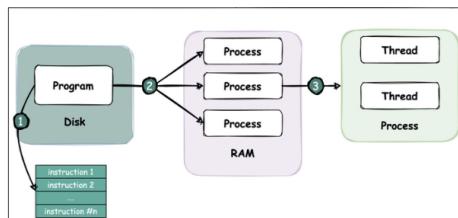
Il CFS può gestire i task in gruppi come un'unica entità schedulabile in maniera fair

## 9 Threads

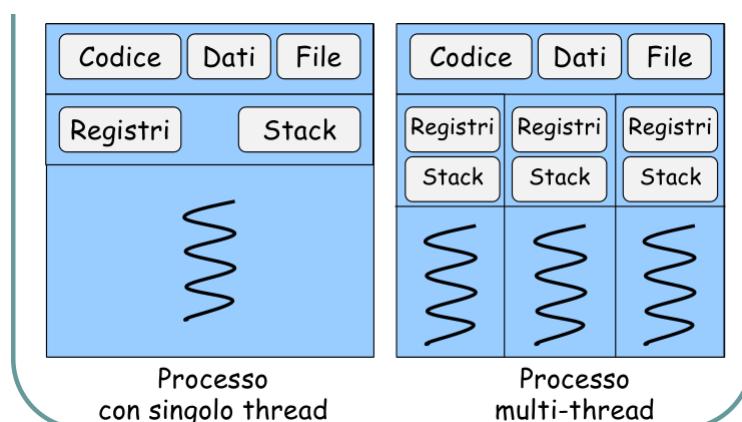
Iniziamo definendo un thread e come si distingue da un processo

- **Thread:** anche detto processo leggero, è un flusso di controllo sequenziale in un processo
- **Processo:** anche detto processo pesante, è l'insieme di spazio di indirizzamento e risorse che possono essere condivise da più threads

I thread dunque condividono le risorse del processo, e tutti i thread hanno visibilità di queste ultime (es. modifiche alla memoria etc.), ciò implica che è richiesta una sincronizzazione esplicita tra i thread (semafori, monitor)



I processi si distinguono in processi multi-thread e processi single-thread



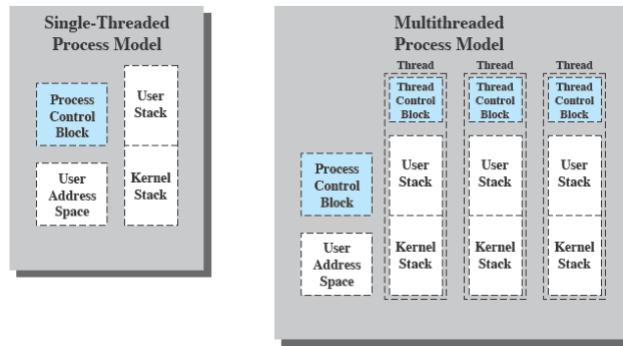
I thread possono aumentare la velocità di esecuzione suddividendo il lavoro in più parti, ed ogni thread può eseguire su cpu distinte. I thread inoltre conferiscono maggiore modularità ai programmi, ad esempio permettono di implementare funzioni asincrone al normale flusso di esecuzione, o separare le attività di background e foreground.

I programmi multi-thread migliorano rispetto ai programmi multiprocesso in quanto terminare o creare un thread è più rapido di terminare o creare un processo, la comunicazione tra thread è più rapida e il context switch ha un minor overhead rispetto a quello tra processi.

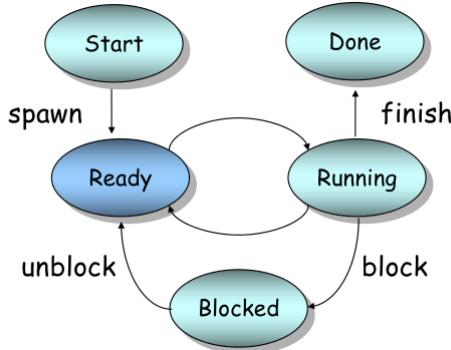
Esistono diversi modelli di programmazione multithread:

- **Manager/Workers:** un thread, il manager, riceve in input comandi e assegna i lavori ad altri thread, i workers.
- **Pipeline:** un task è suddiviso in una serie di operazioni più semplici che possono essere eseguite in serie e concorrentemente da altri thread
- **Peer:** simile al manager/workers ma una volta che il thread principale assegna il lavoro agli altri thread, partecipa attivamente al lavoro

La capacità di un SO di consentire l'esecuzione di più thread all'interno di un singolo processo è definita **multithreading**



I thread come i processi, possiedono anch'essi stati



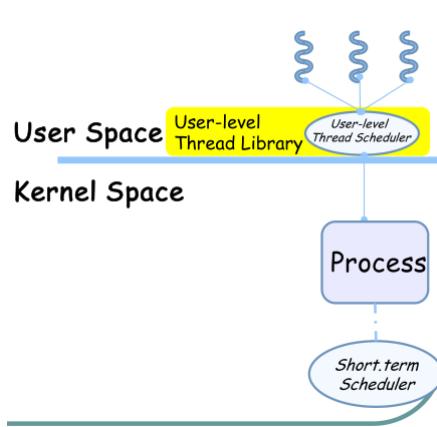
## 9.1 Tipologie di Thread

I thread si dividono in due tipologie principali:

- **User-Level Thread:** anche detti green threads e co-routines
- **Kernel-Level Thread:** anche detti kernel-supported threads e lightweight processes

### 9.1.1 User-Level Thread

La gestione dei thread è eseguita a livello applicativo (il programmatore deve programmarli nel suo software), e il kernel non ha visibilità dell'esistenza di questi thread. Il programma si avvale di una thread library per gestire il suo contesto di esecuzione, e simula il context switch del SO. Quest'approccio prende il nome di scheduling cooperativo.

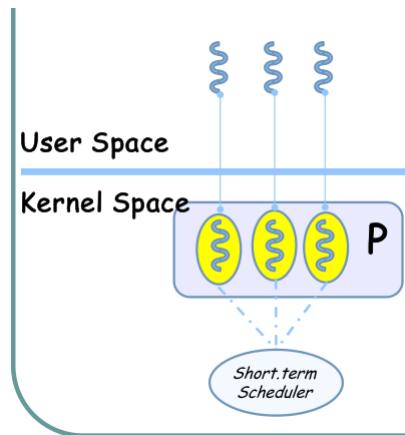


Agli occhi del sistema operativo, il processo esegue comunque su un unico thread, in quanto il multithreading avviene in User space. I vantaggi e gli svantaggi sono

- **Minor overhead** per il context switch in quanto non è richiesto il passaggio in kernel mode
- **Scheduler dei thread indipendente** dallo scheduler dei processi
- **Portabilità** delle applicazioni

### 9.1.2 Kernel-Level-Thread

Nel caso dei Kernel level thread, il kernel gestisce le informazioni di contesto sia per il processo che per i thread. Lo scheduling non viene effettuato sui processi ma sui thread. Questo tipo di approccio è quello adottato dai moderni SO



I vantaggi sono che il kernel è in grado di schedulare più thread dello stesso processo su più processori, e se un thread è "blocked", il kernel può schedulare un altro thread dello stesso processo. Tuttavia il trasferimento del controllo da un thread ad un altro, se pur nello stesso processo richiede un kernel switch a livello kernel.

C'è la possibilità di eseguire approcci combinati, avendo il multithreading implementato sia a livello utente che kernel

## 9.2 Implementazioni dei thread

### 9.2.1 Thread in Go

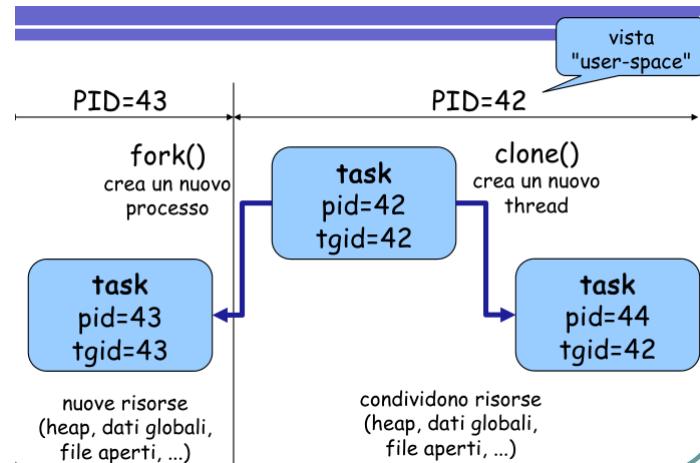
Nel linguaggio go, si ha un approccio misto

- user level thread gestiti dalla libreria del linguaggio
- kernel level thread forniti dall'SO

### 9.2.2 Thread in Linux

In linux, il thread è un task che condivide delle strutture con altri task (codice, heap, etc.). Ogni task ha il suo PID e un processo multithreaded è un gruppo di task identificato da un Thread Group ID (TID).

- La creazione di un task avviene attraverso la syscall **clone()**, che crea un nuovo task che condivide lo stesso spazio d'indirizzamento del task chiamante



### 9.2.3 Thread in Win2000

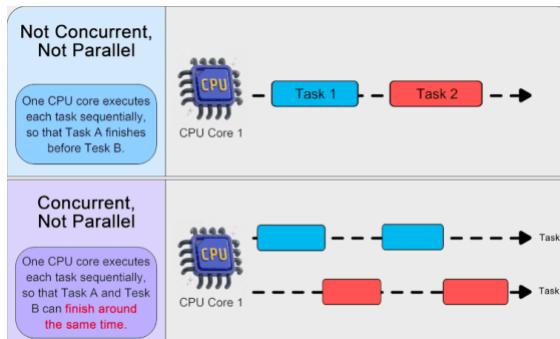
In windows 2000, i thread vengono mappati uno ad uno da ULT a KLT e ciascun thread contiene ID, Registri, User stack, Kernel Stack, e un area di memoria privata

## 10 Programmazione concorrente

Per programmazione concorrente, si intende l'insieme di tecniche, delle metodologie e degli strumenti per lo sviluppo di software come un insieme di attività svolte simultaneamente.

### 10.1 Multiprogrammazione

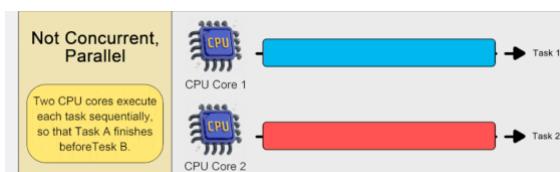
La **multiprogrammazione** è la forma più elementare di programmazione concorrente, e si basa sulla esecuzione intercalata di più processi o threads sulla stessa cpu. Il sistema diventa una macchina astratta che possiede più processori virtuali, uno per ogni processo



I pro sono i miglioramenti all'efficienza complessiva del sistema, ma non migliora la velocità di esecuzione del singolo programma.

### 10.2 Parallelismo

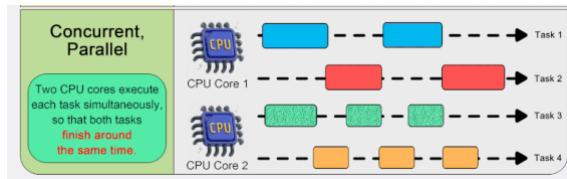
Il **parallelismo** utilizza più cpu fisiche su cui eseguire più processi contemporaneamente.



Anche il parallelismo, se usato da solo non migliora la velocità di esecuzione del singolo programma, ma del sistema.

### 10.2.1 Concorrenza-Parallelismo

L'unico modo per ottenere una velocizzazione di un dato programma, è attraverso l'utilizzo congiunto di parallelismo e multiprogrammazione.



## 11 Sincronizzazione Globale

Nella programmazione concorrente, esistono problemi derivati dall'accesso alle risorse. Senza alcun tipo di sincronizzazione tra processi o thread si arriva ad una condizione definita come **race condition**, dove il processo che accede più velocemente ha il controllo della risorsa. Per garantire l'uso corretto della risorsa essa deve essere acceduta al più da un processo alla volta.

### 11.1 Sezione Critica

Ipotizziamo due o più processi che vogliono utilizzare una risorsa ad uso esclusivo, definita come **risorsa critica**. La sezione di codice che utilizza la risorsa è detta **sezione critica**, e l'obiettivo è di garantirne l'accesso ad un unico processo alla volta. Ciò può avvenire tramite

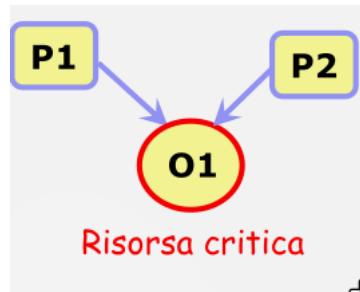
- **Mutua Esclusione:** l'ordine con cui devono avvenire due eventi non è fissato, è sufficiente che i processi non utilizzano in contemporanea la risorsa
- **Comunicazione:** si pone un ordinamento tra gli eventi

Il SO fornisce ai programmi meccanismi per richiedere accesso in mutua esclusione alle risorse

### 11.2 Mutua Esclusione

Supponiamo due processi P1 e P2 e una risorsa critica O1. Nel caso di mutua esclusione si ha

- Quando P1 entra nella sezione critica, acquisisce il possesso della risorsa
- Se P2 tenta di entrare prima che P1 sia uscito, P2 viene posto in attesa dal SO
- Quando P1 ha terminato la sua sezione critica, libera la risorsa e P2 potrà accederci



Il corretto funzionamento non dipende dal numero o dalla velocità di esecuzione dei processi (si evita la Race Condition), e bisogna evitare **deadlock** e **starvation** dei processi:

- **Deadlock:** indica la presenza di una condizione di **blocco permanente** di un gruppo di processi in competizione
- **Starvation:** attesa infinita da parte di un processo a bassa priorità.

Per realizzare la mutua esclusione, ci sono due tipi di soluzioni, soluzioni **hardware** e **software**:

- **Hardware (Disabilitazione degli interrupt):** In un sistema **monoprocesso**, per garantire la mutua esclusione è sufficiente disabilitare le interruzioni. Questo permette di evitare che un processo in una sezione critica venga prelazionato
  - **Vantaggi:** È conveniente per uso interno nel kernel quando è necessario aggiornare delle variabili condivise
  - **Svantaggi:** In un sistema multiprocesso non garantisce la mutua esclusione, e l'approccio viola i principi di protezione

- **Software:** esistono diversi metodi, quali variabili Lock, Mutex, Semafori, Monitor...

### 11.2.1 Lock

Si potrebbe pensare di introdurre una variabile **lock** con valore iniziale 0

- Se un processo accede alla risorsa critica, allora pone lock = 1, e gli altri processi non potranno accedere alla risorsa critica
- Se lock = 0 allora altri processi possono accedere e la porranno = 1

Il problema è che le operazioni di lettura e scrittura, del lock sono eseguite in momenti diversi, dunque se avviene un **context switch** prima di porre "lock=1", un altro processo può entrare nel frattempo nella sezione critica. Per risolvere questo problema molti processori forniscono l'istruzione macchina

TSL RX, LOCK

equivalente a

```
MOV RX, LOCK      // Lettura Lock
MOV LOCK, 1       // Scrittura Lock
```

Le operazioni di lettura e scrittura Lock sono indivisibili ed eseguono in un solo ciclo di CPU, ovviando al problema descritto in precedenza e permettendone l'utilizzo anche nei sistemi multiprocessore. Ci sono due problemi principali in questo approccio

- **Busy Wait:** la soluzione è caratterizzata da attesa attiva da parte del processo che vorrebbe eseguire ma trova la lock ad 1
- **Priority Inversion:** supponiamo di avere due processi che accedono alla stessa sezione critica H e L, con H ad alta priorità ed L a bassa priorità.
  - Si supponga inizialmente solo L in esecuzione, che entra nella sezione critica e imposta LOCK = 1
  - Il processo H diviene pronto e in quanto ha priorità maggiore, L viene prelazionato e la CPU assegnata ad H
  - H inizia il busy wait tramite TSL in quanto LOCK è stata lasciata ad 100
  - H rimane bloccato in attesa attiva per sempre in quanto L non avrà mai possibilità di eseguire con H in esecuzione

I problemi di attesa attiva e di priority inversion vengono risolti forzando il processo che trova la LOCK ad 1 a **sospendersi**, transita nello stato bloccato in attesa che la risorsa diventi disponibile. Ciò avviene tramite 2 syscall

- **suspend(P):** Il processo P che la chiama si auto-sospende in attesa di un segnale di risveglio
- **wake-up(P):** Un processo invia un segnale di risveglio al processo P sospeso

### 11.2.2 Semafori e Mutex

## 12 Cooperazione

Come accennato in precedenza, in alcuni casi non è sufficiente gestire casualmente l'ordine di accesso alla risorsa, ma possono esistere problematiche di **coordinazione** in cui si vuole gestire l'ordine di accesso.

### 12.1 Prod-Cons

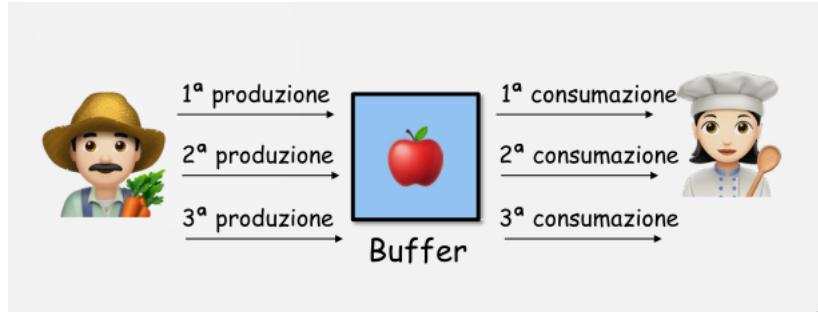
Il problema del **Produttore-Consumatore** è strutturato nel seguente modo. Esistono due categorie di processi

- **Produttori:** depositano un messaggio su di una risorsa condivisa
- **Consumatori:** prelevano il messaggio dalla risorsa condivisa

### 12.1.1 Prod-Cons single buffer

Supponiamo di voler realizzare un problema di tipo Produttore-Consumatore con **buffer unico**, i vincoli sono i seguenti:

- Il produttore non può produrre un messaggio se il consumatore non ha consumato il messaggio precedente
- Il consumatore non può prelevare un messaggio se il produttore non l'ha prima depositato



La realizzazione richiede l'utilizzo di 2 **semafori**:

- **AVAILABLE\_SPACE**: semaforo bloccato da un produttore prima di una produzione e sbloccato da un consumatore in seguito ad un consumo (valore iniziale 1)
- **AVAILABLE\_MSG**: semaforo sbloccato da un produttore in seguito ad una produzione e bloccato da un consumatore prima del consumo (valore iniziale 0)

### 12.1.2 Prod-Cons con coda

Si tratta di una variante del problema produttore consumatore gestita attraverso un **vettore di buffer**



- Il produttore si sospende se i buffer sono tutti pieni
- Il consumatore si sospende se i buffer sono tutti vuoti

Le operazioni sono svolte in ordine circolare. La coda è implementata mediante i seguenti campi

- **buffer[DIM]**: un array di elementi di tipo msg (tipo del messaggio depositato dai produttori) contenente i valori prodotti
- **testa**: un intero che indica la posizione del primo buffer libero in testa (`buffer[testa]`), ossia il primo buffer disponibile per la memorizzazione di un messaggio. L'accesso al msg prodotto più recentemente avviene tramite `buffer[testa - 1]`
- **coda**: indica la posizione dell'elemento prodotto meno recentemente, `buffer[coda]` permette di accedere alla prossima consumazione

Per la sincronizzazione si utilizzano due semafori

- **AVAILABLE\_SPACE**: indica la presenza di spazio disponibile in coda per la produzione di un messaggio ed ha come valore iniziale la dimensione della coda
- **NUM\_MSG**: indica il numero di messaggi presenti in coda, ha valore iniziale 0

Questo metodo garantisce che non vi siano accessi contemporanei alla stessa posizione della coda da parte di produttori e consumatori, tuttavia, va bene solo nel caso in cui ci sia 1 produttore e 1 consumatore. Se ci sono due buffer liberi e due produttori iniziano contemporaneamente a produrre si verifica una **race condition**.

### 12.1.3 Prod-Cons multipli con coda

Nell'ipotesi in cui vi siano più produttori e più consumatori che accedono allo stesso buffer, le operazioni di deposito e prelievo devono essere eseguite rispettivamente in mutua esclusione, ed essere dunque gestite come **sezioni critiche**. A tal fine si utilizzano due nuovi semafori:

- **MUTEX\_C**: per le operazioni di consumo (valore iniziale 1)
- **MUTEX\_P**: per le operazioni di produzione (valore iniziale 1)

Per una corretta sincronizzazione è necessario che gli indici testa e coda siano condivisi tra processi. Ciò avviene collocando sia il vettore buffer che gli indici coda/testa nella stessa memoria condivisa

```
typedef struct {
    int testa;
    int coda;
    int buffer [DIM];
} prod_cons;
```

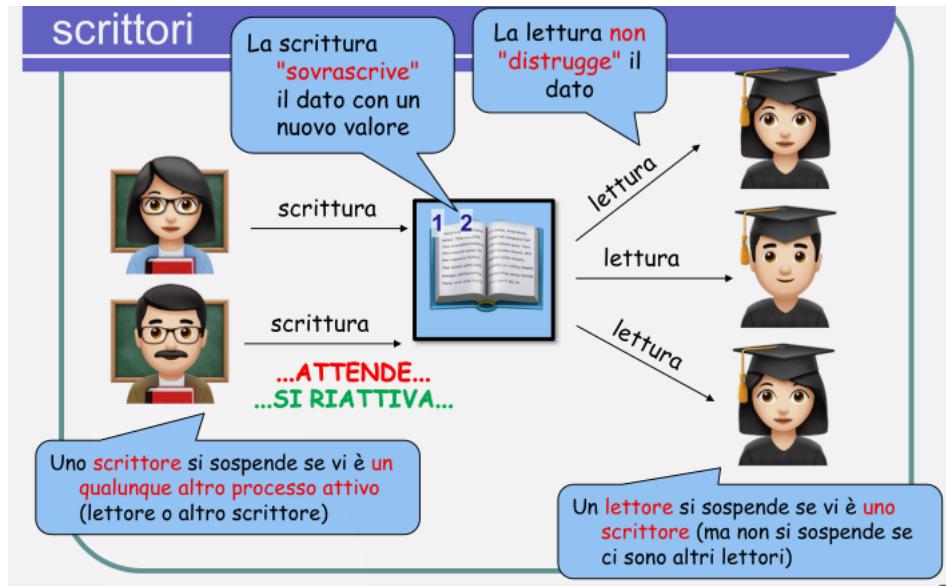
## 12.2 Lettori/Scrittori

Un altro problema di coordinazione è il problema dei Lettori/Scrittori. Esistono due attori principali

- **Lettori**: leggono un messaggio su di una risorsa condivisa
- **Scrittori**: scrivono il messaggio sulla risorsa condivisa

I processi lettori possono accedere contemporaneamente alla risorsa, mentre i processi scrittori hanno accesso esclusivo alla risorsa. I lettori e scrittori si escludono mutuamente dall'uso della risorsa. Questo problema può portare ad una condizione di **starvation**:

- Un processo lettore attende solo se la risorsa è occupata da un processo scrittore, e un processo scrittore può accedere alla risorsa solo se questa è libera
- I processi scrittori sono dunque soggetti a possibile attesa indefinita (starvation)

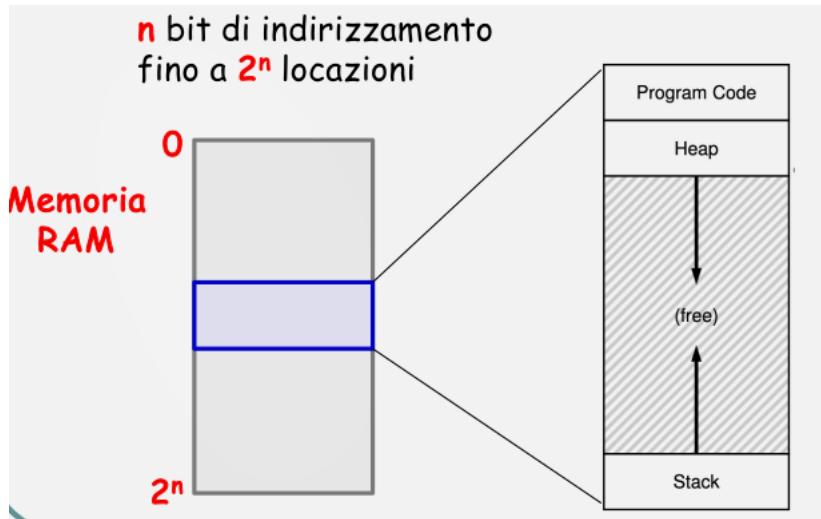


La presenza di un lettore permette solo ad altri lettori di entrare, ma non permette l'ingresso agli scrittori, funziona se vi è almeno un lettore attivo, lo scrittore subisce starvation

## 13 Gestione memoria

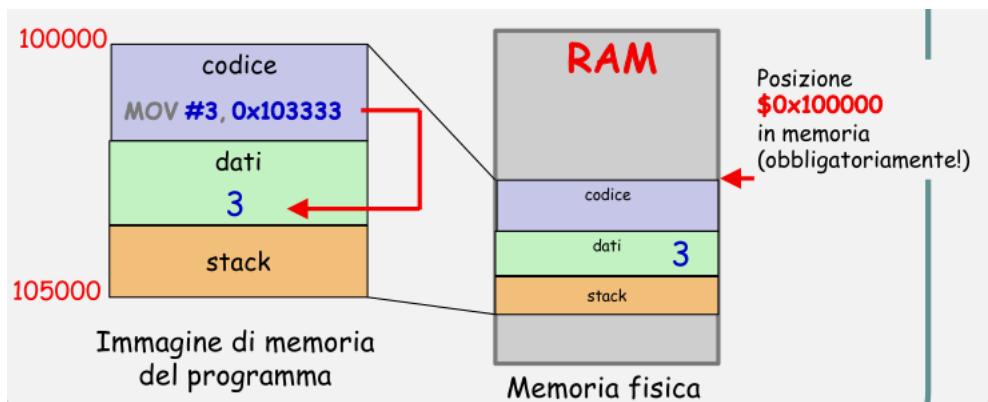
La **memoria principale** costituisce, insieme alla CPU, una delle risorse per realizzare la astrazione di processo.

- Il processo dispone di una area di memoria ad esso riservata



La posizione (indirizzi) di codice e dati nella memoria del processo è un'astrazione. Il sistema operativo gestisce la **memoria fisica** e mette a disposizione dei processi **indirizzi virtuali** in un'operazione definita **virtualizzazione della memoria**. Gli indirizzi virtuali possono essere assegnati in 2 modi:

- **Rilocazione statica:** stabilisce gli indirizzi di codice e dati al momento della **compilazione**. Quest'ultima non può essere modificata e l'indirizzo di memoria virtuale corrisponde con quello di memoria fisica

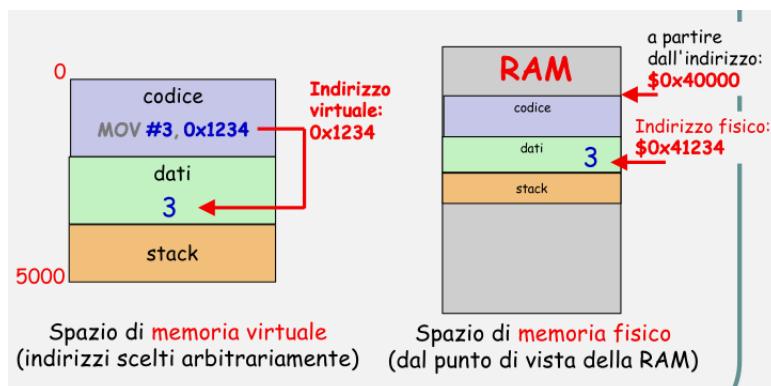


- **Rilocazione dinamica:** la traduzione tra indirizzi virtuali e fisici avviene durante l'esecuzione da parte della **MMU**

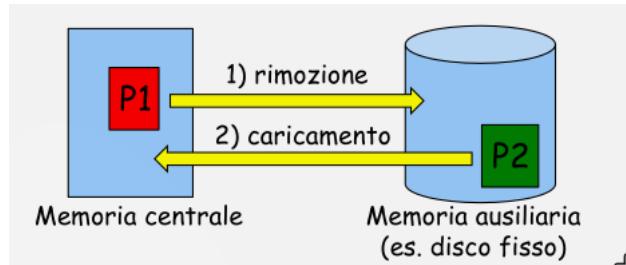
### 13.1 Rilocazione Dinamica

Per comprendere il processo di rilocazione dinamica, è necessario introdurre una distinzione tra tipi di indirizzi

- **Indirizzo Fisico:** indirizzo visto dall'unità di memoria, **posizione effettiva** del dato/istruzione in memoria fisica
- **Indirizzo Virtuale:** indirizzo acceduto dal programma durante l'esecuzione, un astrazione dell'indirizzo fisico



La rilocazione dinamica, oltre a fornire una preziosa astrazione per la gestione della memoria di ogni singolo processo, consente di effettuare lo **swapping**, ossia un'operazione che permette di sospendere e trasferire dalla memoria centrale alla memoria secondaria un processo per liberare la memoria centrale.



## 13.2 MMU

Come detto in precedenza, la gestione degli indirizzi fisici e virtuali, è affidata alla **Memory Management Unit (MMU)**. Quest'ultima è un componente hardware della CPU che può funzionare in diversi modi

- **Caso Basilare**: approccio primordiale con fallo di sicurezza
- **Base - Bound**: approccio moderno

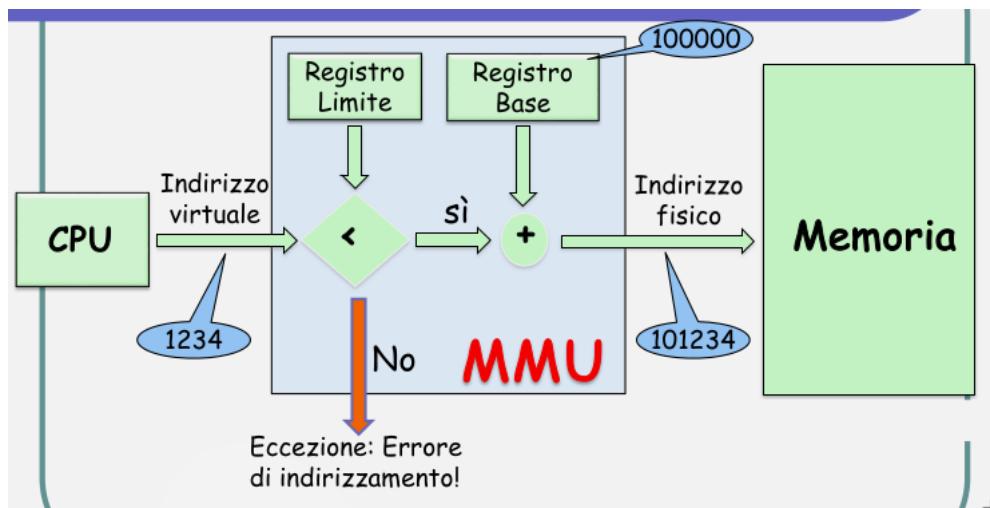
l'approccio Base-Bound si basa su due registri, rispettivamente:

- **Base**: contiene l'indirizzo di memoria fisico da cui parte l'immagine di memoria del processo o il segmento
- **Bound**: contiene l'indirizzo di memoria fisico in cui termina l'immagine di memoria del processo o il segmento

L'indirizzo fisico, è dato da

$$\text{indirizzo fisico} = \text{indirizzo virtuale} + \text{registro base}$$

Il registro bound, permette di controllare che il processo non stia accedendo a zone di memoria non riservate a quest'ultimo



## 13.3 Gestione dello spazio virtuale

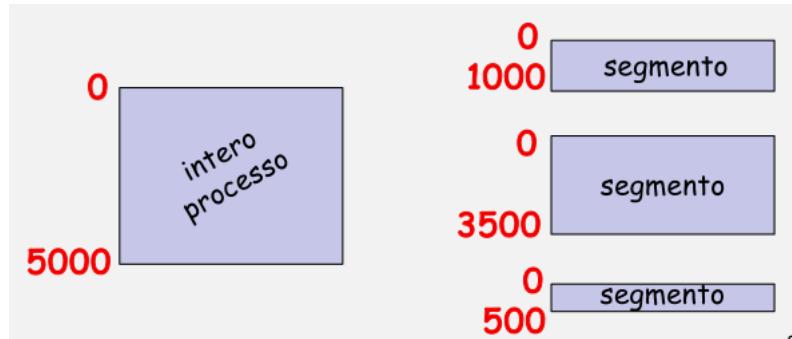
Un indirizzo virtuale è una coppia composta da **identificativo** del segmento e **scostamento** all'intero del segmento

```
push $0x1234
pop %es
mov %es: 0x5678 , %eax
```

l'indirizzo 0x1234 è l'identificativo, e 0x5678 è l'offset. Vi sono due possibili approcci per gestire lo spazio virtuale degli indirizzi:

- **Spazio unico**: si alloca uno spazio corrispondente all'intero processo in un unico "blocco" di memoria

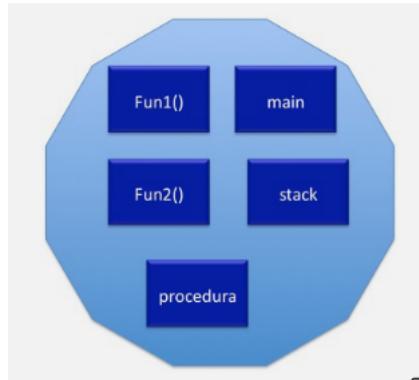
- Segmenti:** si suddivide il processo in diverse "porzioni" e si gestiscono separatamente (segmento codice, segmento heap, segmento stack)



Possono esistere problemi di frammentazione (spazio non utilizzato) in entrambi i casi. Nel caso in cui sia interna all'intero processo di definisce **frammentazione interna**, altrimenti **frammentazione esterna**

### 13.3.1 Segmentazione

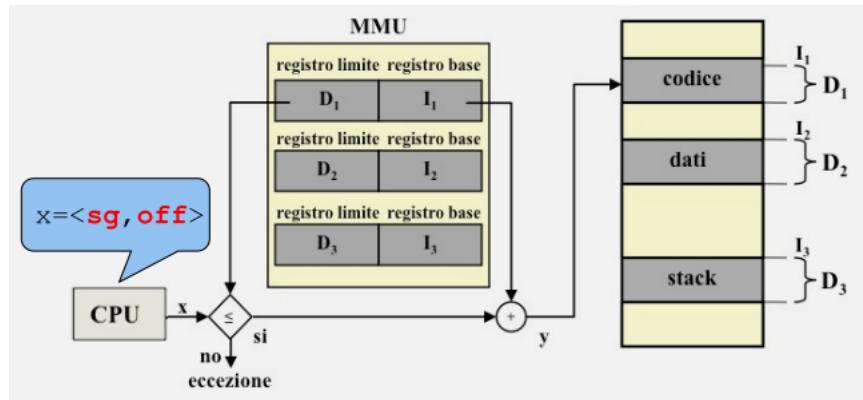
A tempo di compilazione si configura lo spazio virtuale segmentato, e viene creato un segmento diverso per ciascun modulo del programma.



Quest'approccio comporta alcuni vantaggi

- Protezione** dei segmenti
- Condivisione** dei segmenti
- Allocazione indipendente** dei segmenti in memoria fisica

Anche la segmentazione si basa su MMU, tuttavia il numero di registri base/bound sono limitati, dunque in caso di pochi segmenti, è possibile avere nella MMU più coppie di registri base/bound

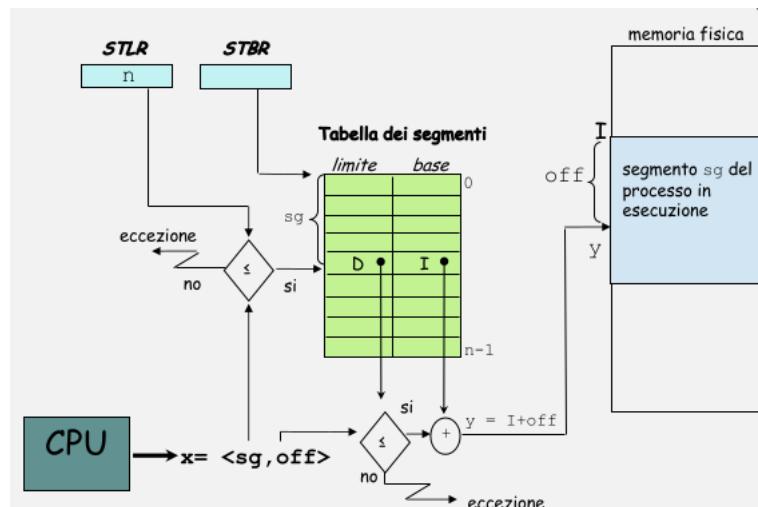


ma in caso si voglia ottenere un numero arbitrario di segmenti è necessaria una struttura dati in memoria RAM per contenere le coppie base-bound detta **segment table**. La MMU gestisce la segment table con due appositi registri

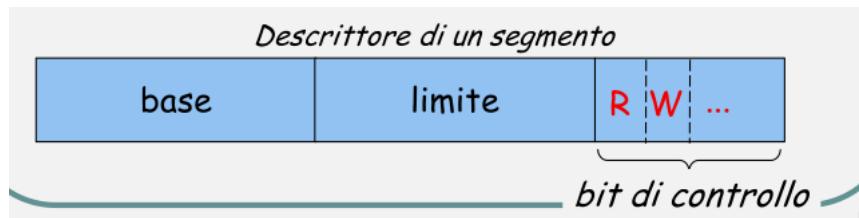
- STBR** (segment table base register): indirizzo in memoria in cui si trova la tabella dei segmenti

- **STLR** (segment table limit register): dimensione della tabella dei segmenti

La traduzione degli indirizzi segmentati diventa dunque



Ogni processo ha una segment table differente, e i registri STBR/STLR sono configurati ad ogni **context switch** dei processi. Il sistema operativo carica i valori di STBR/STLR dal PCB. La segment table permette anche di specificare diversi permessi di accesso al segmento, attraverso dei bit di controllo



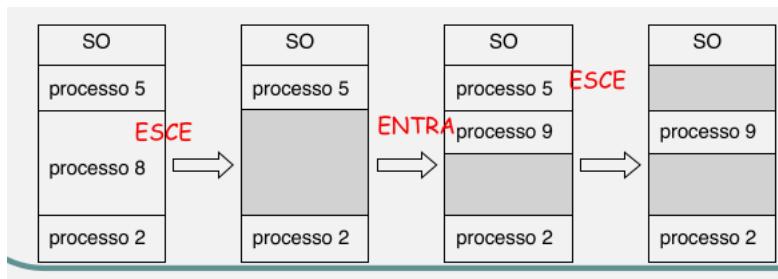
È possibile **condividere** segmenti tra più processi, allocando in memoria fisica solo una copia del segmento e facendo riferimento a quest'ultima nella segment table di due processi distinti

## 13.4 Allocazione

Uno spazio/segmento di memoria virtuale può essere allocato in memoria fisica in due possibili modi

- **Allocazione contigua:** lo spazio/segmento è copiato per intero in un intervallo di memoria fisica agli indirizzi  $[I, I+D]$
- **Allocazione non contigua:** tramite paginazione

Nel caso di allocazione contigua, il SO colloca il blocco di memoria virtuale, in intervalli non sovrapposti della memoria fisica, quando un processo termina la memoria fisica occupata si libera creando un buco (**hole**). Quando si carica un processo, occorre cercare un hole sufficientemente grande da contenerlo



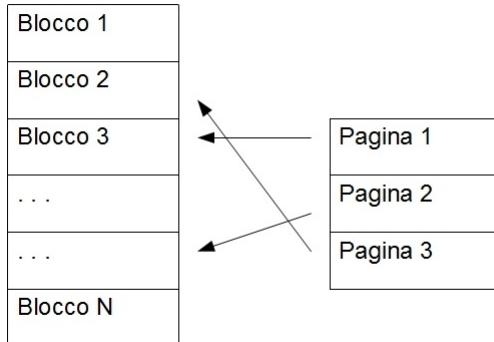
Se ci sono più buchi liberi, ci sono vari criteri per scegliere dove collocare il segmento

- **First-fit:** si assegna il primo hole sufficientemente grande
- **Best-fit:** si assegna il più piccolo hole tra i sufficientemente grandi
- **Worst-fit:** si assegna l'hole più grande

In genere l'allocazione contigua soffre di problemi di frammentazione esterna

### 13.5 Paginazione

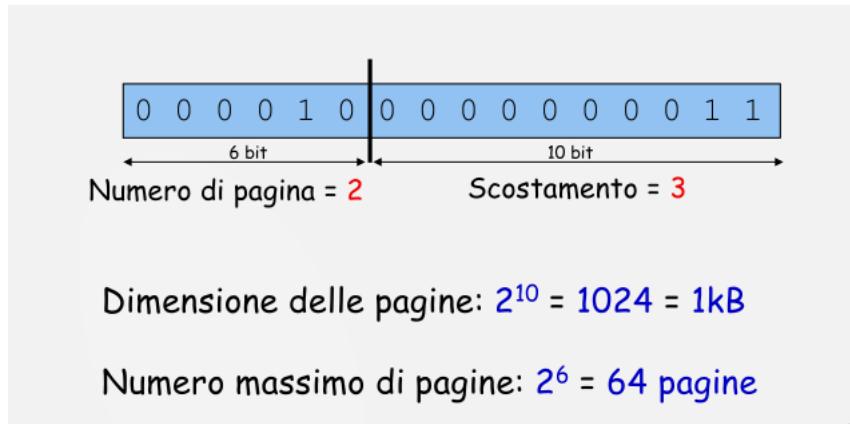
La **paginazione** è una tecnica di allocazione non contigua, in cui gli spazi/segmenti sono divisi in **blocchi di dimensione fissa**. Ciò permette di evitare la frammentazione esterna ma introduce la frammentazione interna



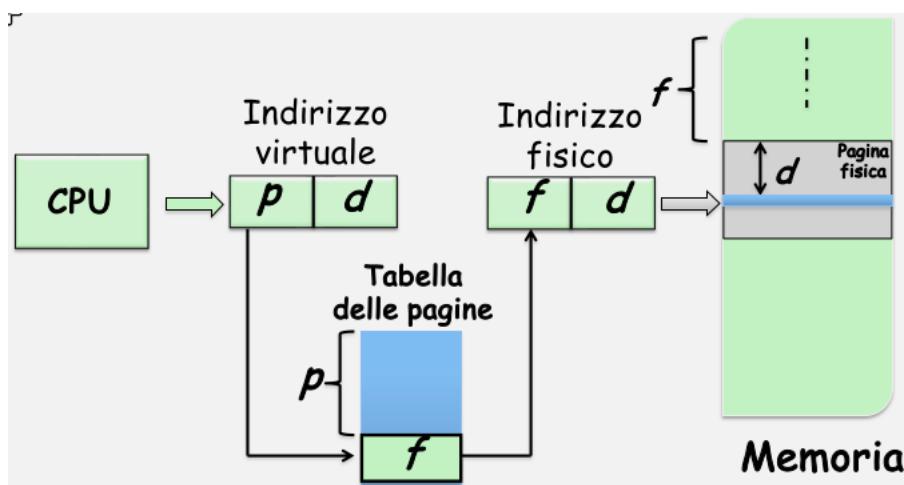
- La memoria virtuale è divisa in blocchi di dimensione fissa detti pagine virtuali
- La memoria fisica è divisa in blocchi di dimensione fissa detti pagine fisiche o **frame**
- Ogni pagina virtuale è abbinata ad una pagina fisica tramite **tabella delle pagine**

Si ha un problema di **frammentazione interna** quando il processo non utilizza a pieno le pagine assegnate, tuttavia si tratta di un fenomeno trascurabile per pagine piccole. Tipicamente la dimensione di pagina è una potenza di 2 compresa tra (512byte e 16MB). Nella paginazione un indirizzo contiene la coppia

- **numero di pagina (p)**: indice della pagina nella memoria fisica
- **scostamento di pagina (d)**: indica la posizione dell'indirizzo all'interno della pagina

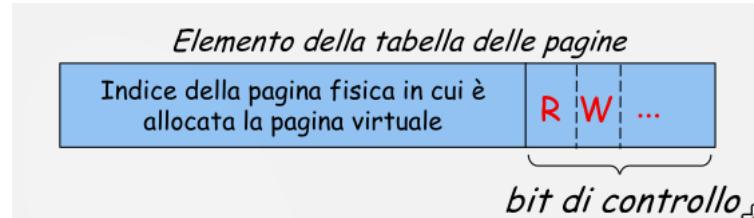


A differenza della segmentazione, non sono due valori separati, ma sono contenuti entrambi in un unico valore. L'architettura di paginazione è la seguente



La tabella delle pagine ha una riga per ogni **pagina virtuale** del processo contenente

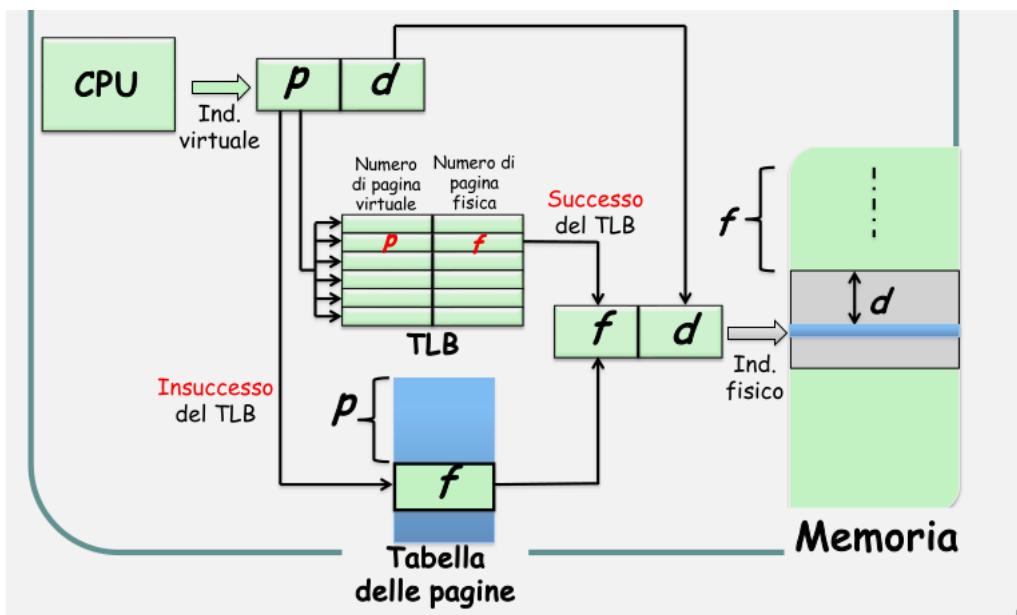
- l'indice della **pagina fisica**
- **bit di gestione** (permessi di accesso ecc...)



La tabella delle pagine è in **memoria principale** e la MMU usa 2 registri per riconoscerla

- **PTBR (page-table base register)**: indirizzo base della tabella delle pagine
- **PTLR (page-table length register)**: dimensione della tabella delle pagine

Per accedere alla memoria occorrono dunque due accessi, uno per **leggere la tabella delle pagine**, e uno per **accedere al dato/istruzione** vero e proprio. Per migliorare l'efficienza si usa una cache associativa detta **TLB** (Translation Look-Aside Buffer)



Il TLB esegue una **selezione associativa** basata sul contenuto, mentre la selezione della riga dalla tabella delle pagine in memoria è sempre **lineare** (basata sulla posizione). Inoltre, il sistema operativo può marcare le **pagine virtuali in uso**, usando un **bit di validità** nella page table. Il bit viene attivato nel momento in cui la pagina è allocata dal processo (es. tramite malloc())



Ci sono diversi problemi da risolvere riguardanti la tabella delle pagine:

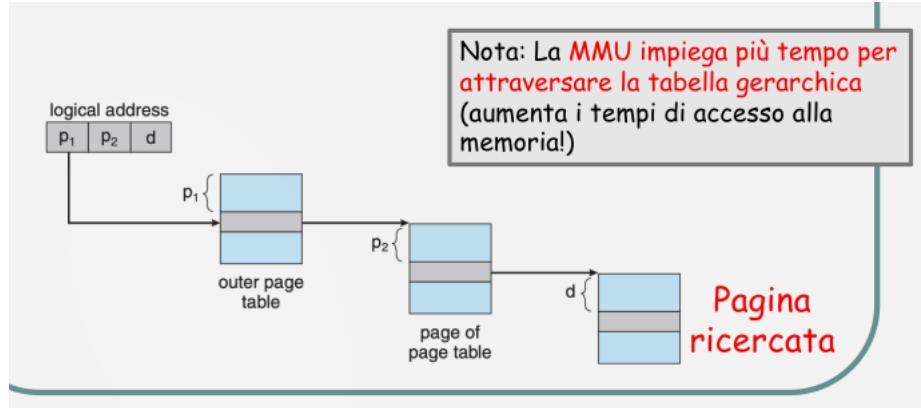
- pagine con dimensioni troppo grosse
- pagine troppo numerose
- pagine "sparse" (poche pagine valide)

Questi problemi vengono risolti tramite **paginazione gerarchica**, **tabella delle pagine basata su hash**, **tabella delle pagine invertita**

### 13.5.1 Paginazione gerarchica

È una tecnica che permette di suddividere la tabella delle pagine in parti più piccole, secondo una **organizzazione gerarchica**.

- La MMU divide l'indirizzo di pagina in più parti ( $p_1, p_2$ )
- Nella tabella di primo livello, trova l'indirizzo della tabella di secondo livello ecc...



### 13.5.2 Tabella delle pagine su Hash

Le righe della tabella delle pagine sono organizzate utilizzando una **linked list**

- Si memorizzano esclusivamente le righe per le pagine valide
- Ulteriore risparmio di memoria ma rallenta la ricerca

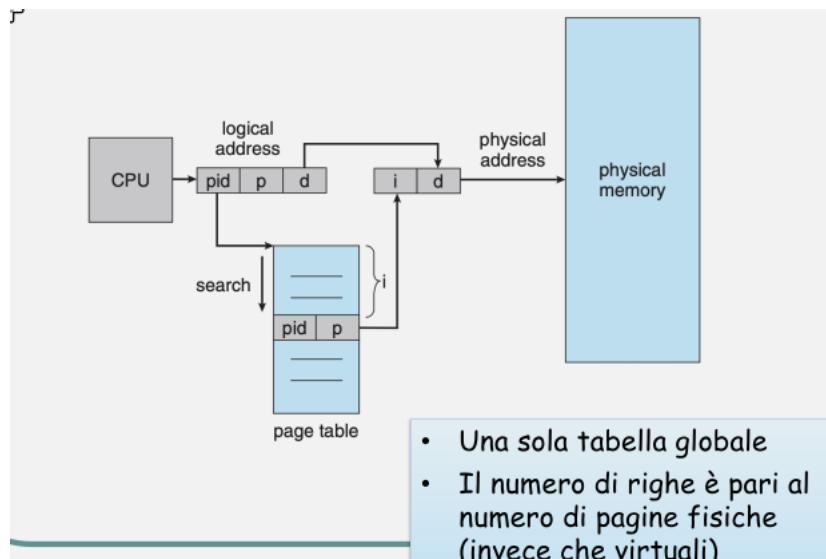
Per ottimizzare i tempi di ricerca si dividono le righe su tante liste concatenate di piccole dimensioni

- Una funzione di **Hash** è applicata al numero della pagina virtuale
- Elementi con lo stesso valore della funzione di hash sono collocati nella stessa lista concatenata

### 13.5.3 Tabella delle pagine invertita

Negli schemi precedenti, si ha una tabella distinta per ogni processo, nella tabella delle pagine invertita invece

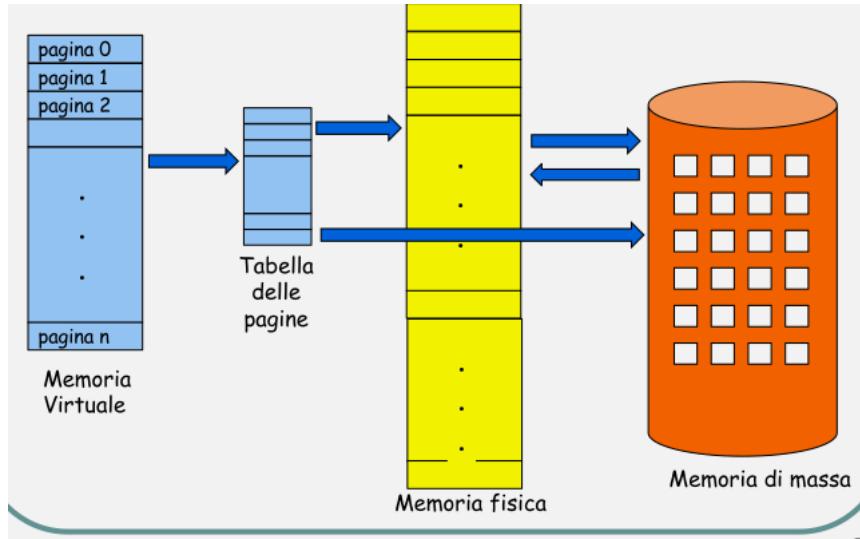
- Si ha una sola tabella comune a tutti i processi
- La tabella ha un elemento per ogni pagina fisica
- Ogni elemento contiene l'indirizzo virtuale della pagina memorizzata in quella locazione fisica



## 14 Memoria Virtuale

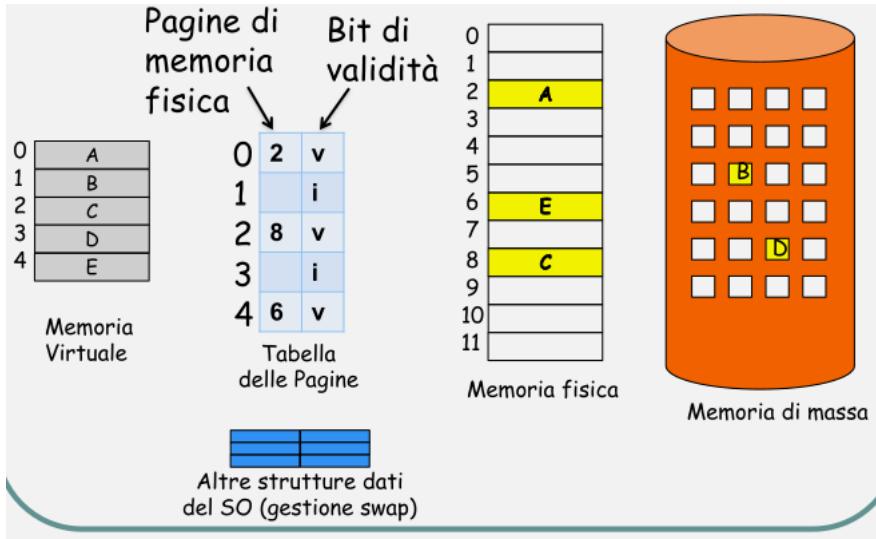
La **memoria virtuale** separa la memoria fisica dalla memoria vista da un processo.

- Può essere più grande della memoria fisica a disposizione attraverso swapping
- Si realizza con la paginazione o segmentazione "su richiesta"



La **paginazione su richiesta** implica che solo le pagine effettivamente utilizzate sono caricate in memoria principale (pagine "residenti"). Ciò permette un minore consumo di memoria fisica e un generale risparmio di risorse del calcolatore, si realizza attraverso il **bit di validità**:

- **Bit di validità ad 1:** se la pagina virtuale è stata allocata al processo (es. malloc) e la pagina fisica corrispondente è risedente in memoria
- **Bit di validità a 0:** se anche una sola condizione è falsa



Nel caso in cui un indirizzo virtuale fa riferimento ad una pagina non ancora caricata in memoria

- La MMU nota che il bit di validità non è attivo
- La MMU genera un'eccezione di pagina mancante **page fault**

### 14.1 Page Fault

In seguito ad un interrupt di tipo **page fault**, una ISR del SO gestisce l'eccezione nel seguente modo

1. Individua una pagina fisica di memoria libera
2. Trasferisce la pagina desiderata nella memoria libera

3. Aggiorna le tabelle, bit di validità = 1
4. All'uscita dalla ISR, la CPU riavvia l'istruzione che ha causato l'eccezione

La ISR verifica anche che l'indirizzo virtuale sia stato allocato dal processo, in caso negativo il processo viene terminato con **segmentation fault**.

#### 14.1.1 Page fault con sostituzione

Una variante della page fault classica, è la **page fault con sostituzione**, in cui la ISR

1. Il SO individua la posizione sul disco della pagina richiesta
2. il SO cerca una pagina fisica di memoria libera:
  - Se esiste, la si usa
  - Altrimenti occorre togliere dalla memoria principale un'altra paginda detta "**vittima**" (scelta con un algoritmo di sostituzione)
  - La pagina vittima viene scritta sul disco (swap-out)
3. Il SO scrive la pagina richiesta sulla pagina libera
4. Il SO modifica le tabelle delle pagine (sia richiedente che processo vittima)
5. Si riprende l'esecuzione del processo richiedente

La scelta della pagina vittima è fatta da un **algoritmo di sostituzione delle pagine**, il cui obiettivo è minimizzare la frequenza dei page fault. I quattro principali sono

- Algoritmo "ottimo" (puramente teorico)
- FIFO
- LRU
- Second-Chance

## 15 IPC-Shmem-Semafori

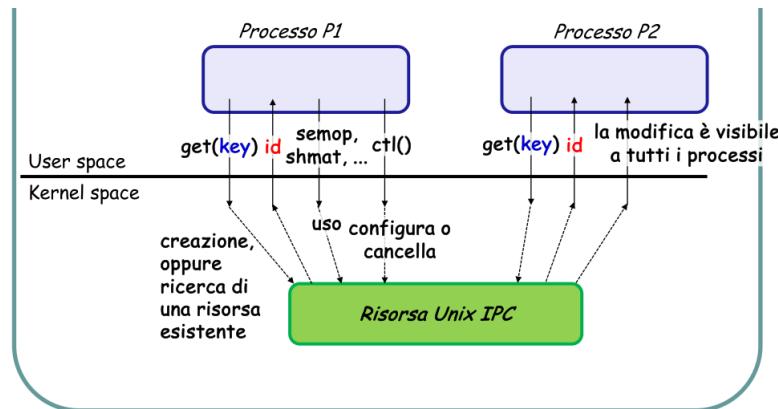
### 15.1 IPC

Linux/Unix permette la comunicazione tra processi mediante primitive e strutture dati fornite dal kernel, dette **IPC** (interprocess communication).

- **Memoria condivisa:** SHM - shared memory segments
- **Semafori:** SEM - semaphore arrays
- **Code di messaggi:** MSG - queues

Ogni risorsa IPC è gestita da due primitive: get/ctl

- **get:** utilizza una "chiave" (IPC key), ed opportuni parametri per restituire al processo un descrittore della risorsa.
- **ctl:** permette dato un descrittore, di verificare lo stato di una risorsa, cambiare lo stato di una risorsa, rimuovere una risorsa



Le risorse IPC non sono volatili, se un processo esce, non vengono eliminate automaticamente ma è necessaria una chiamata alla primitiva ctl.

#### 15.1.1 Primitiva get

```
int ... get(key_t key, ... , int flag);
```

per **key** si intende la chiave dell'oggetto, questa può essere un valore intero arbitrario, un valore generato dalla funzione ftok(), o la costante IPC\_PRIVATE. Flag indica la modalità di acquisizione della risorsa, si possono passare una o più costanti tra cui

- **IPC\_CREAT:** crea una nuova risorsa se non ne esiste già una con la chiave indicata. Se già esiste viene riusata
- **IPC\_EXCL:** si usa in combinazione con IPC\_CREAT, e permette di gestire i due casi di risorsa già esistente e risorsa appena creata (utile per gestire il valore iniziale della risorsa)
- **Permessi di accesso:** ad esempio 0644

#### 15.1.2 Primitiva ctl

```
int ... ctl(int desc, ... , int cmd, ... );
```

- **desc:** indica il descrittore della risorsa
- **cmd:** indica il comando da eseguire tra
  - **IPC\_RMID:** rimozione della risorsa
  - **IPC\_STAT:** richiede informazioni sulla risorsa
  - **IPC\_SET:** richiede la modifica di attributi della risorsa (es. parametri di accesso)

### 15.1.3 IPC Keys

Ogni risorsa IPC è identificata da una IPC key univoca. Ci sono diversi modi per scegliere una chiave, volendo può essere anche hardcoded con un valore a scelta del programmatore, ma il metodo più pulito è l'ftok().

```
key_t ftok( char * path , char id )
```

Genera una chiave automaticamente e prende in ingresso il percorso di un file/cartella appartenente al programma, e carattere scelto a piacere dal programmatore. Un altro m

```
key_t mykey = ftok( "tmp" , "a" )
```

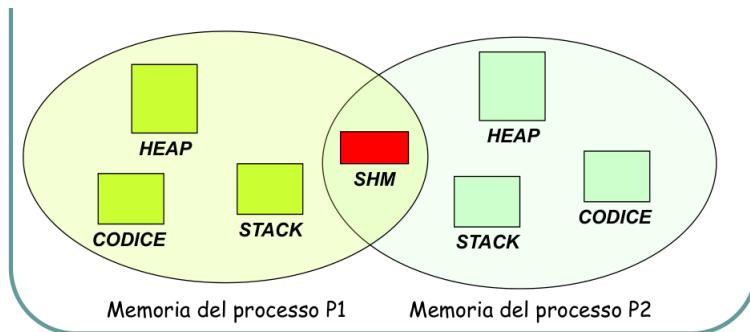
Un altro metodo per generare una chiave corrisponde al IPC\_PRIVATE. Questo equivale a 0 ed è un valore costante per creare una risorsa senza chiave, semplificando la gestione ma impedendo l'accesso ad altri processi. L'unico modo per condividere questo tipo di risorsa è tramite fork()

```
key_t mykey = IPC_PRIVATE
```

I comandi ipcs e ipcrm in shell sono utili per visualizzare le risorse IPC allocate nel sistema o rimuovere risorse non rimosse dai processi (es. terminazione anomala).

## 15.2 Memoria condivisa

Una memoria condivisa (**SHM**) è una porzione di memoria accessibile da più processi



Due comandi utili da shell sono

- **ipcs**: visualizza le risorse IPC allocate nel sistema
- **ipcrm**: rimuove una risorsa IPC

### 15.2.1 Creazione SHM

```
int shmget(key_t key , int size , int flag )
```

L'unico parametro differente da una get è il parametro size, che stabilisce la dimensione in byte della memoria condivisa. la funzione shmget restituisce un identificatore numerico (descrittore) in caso di successo, o -1 in caso di fallimento

```
key_t chiave = 40;
int ds_shm;

ds_shm = shmget(chiave , 1024 , 0);
if(ds_shm < 0) {
    // la risorsa non esiste! esci dal programma
    perror("errore_shmget!");
    exit(1);
}
```

Se la shm non è stata già creata da un altro processo, la shmget() produce un errore. Successivamente alla creazione o all'accesso ad una shm, bisogna effettuare un'operazione di **attach** che collega il segmento di memoria condivisa allo spazio di indirizzamento del chiamante attraverso la syscall shmat().

```
void* shmat(int shmid , const void *shmaddr , int flag );
```

dove shmid, rappresenta l'id della shm, e la funzione restituisce un puntatore all'area di memoria collegata o -1 in caso di fallimento. I parametri opzionali possono essere

- **shmaddr**: indirizzo a quale collegare il segmento di memoria condivisa, se NULL viene scelto in automatico dall'SO
- **flag**: 0 per lettura/scrittura, IPC\_RONLY per sola lettura

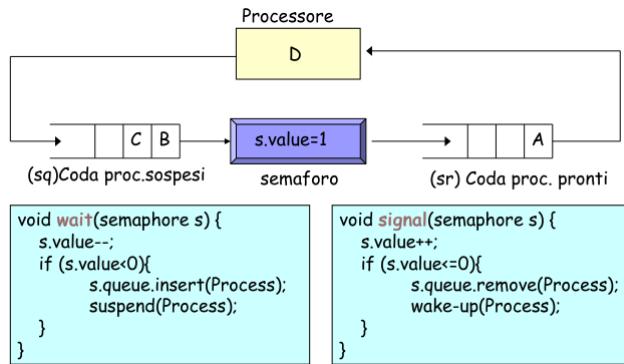
Le operazioni di controllo `shmctl()`, seguono lo stesso metodo delle `ctl`, quindi

```
int shmctl(int ds_shm, int cmd, struct shmid_ds * buff)
```

Dove `ds_shm` è il descrittore della shm, `cmd` permette di specificare uno dei comandi (IPC\_STAT, IPC\_SET, IPC\_RMID, SHM\_LOCK), `buff` è un puntatore alla struttura di tipo `shmid_ds`. La funzione restituisce -1 in caso di errore

### 15.3 Semafori

L'idea dietro un semaforo è quella di gestire l'accesso a risorse condivise in zone critiche di un programma



Le syscall per operare sui semafori sono

```
int semget(key_t key, int nsems, int semflg);
```

la `semget` definisce un array di più semafori (`nsems` è il numero di semafori), e ogni semaforo dall'array è rappresentato internamente nel kernel da una struttura dati. L'altra syscall rilevante è

```
int semctl(int semid, int semnum, int cmd, ...);
```

Dove `semid` è l'id dell'array di semafori, `semnum` il numero di semafori su cui operare, `cmd` il tipo di operazione. Vediamo come avviene l'inizializzazione di un semaforo

```

key_t = IPC_PRIVATE;

// richiedo array di due semafori
int sem = semget(key, 2, IPC_CREAT | 0664);

// inizializzo i semafori
semctl(sem, 0, SETVAL, 1) // mutex o semaforo binario
semctl(sem, 1, SETVAL, 5) // semaforo n-ario
    
```

Un ulteriore syscall per quanto riguarda i semafori, è la `semop()`.

```
int semop(int semid, struct sembuf *sops, unsigned nsops)
```

Ci permette di effettuare un'operazione o un gruppo di operazioni su un vettore di semafori. Prende in ingresso un `semid`, un vettore di operazioni `*sops` di dimensioni `nsops`, e restituisce un intero. Le implementazioni che ci interessano principalmente sono 2, la `wait` e la `signal`

```

void Wait_Sem(int id_sem, int numsem){
    struct sembuf sem_buf;

    sembuf.sem_num = numsem;
    sembuf.sem_op = -1;
    
```

```

sembuf.sem_flg = 0;

semop(id_sem, &sem_buf, 1);
}

```

La wait decrementa il semaforo di 1 per indicare che un processo sta già operando sulla risorsa condivisa

```

void Signal_Sem (int id_sem, int numsem){
    struct sembuf sem_buf;

    sem_buf.sem_num=numsem;
    sem_buf.sem_flg=0;
    sem_buf.sem_op=1;

    semop(id_sem,&sem_buf,1);
}

```

La signal incrementa il semaforo di 1 per indicare che un processo ha finito di operare sulla risorsa condivisa.  
Possiamo dunque definire 3 operazioni principali su un semaforo in base al valore di sem\_op.

- $\text{sem\_op} < 0 \rightarrow \text{wait}$

Se  $\text{sem\_op}$  ha valore negativo, l'operazione si articola in 2 modi.

- $\text{semval} \geq |\text{sem\_op}|$ , l'operazione procede senza sospendere il processo e il valore del semaforo sarà  $\text{semval} - = |\text{sem\_op}|$
- $\text{semval} \leq |\text{sem\_op}|$ , il processo si sospende finché  $\text{semval} \geq |\text{sem\_op}|$  e poi si procede come nel caso precedente

- $\text{sem\_op} > 0 \rightarrow \text{signal}$

Aumenta sempre il semaforo di  $\text{semval} += |\text{sem\_op}|$ , dunque non causa blocco del processo

- $\text{sem\_op} < 0 \rightarrow \text{wait for zero}$  Se  $\text{sem\_op}$  ha valore nullo, si ha l'operazione di wait for zero
  - se il valore  $\text{semval} = 0$ , l'operazione procede immediatamente e il processo non viene bloccato
  - se il valore  $\text{semval}$  è diverso da zero, il processo si sospende finché  $\text{semval} = 0$

## 16 PThreads

La libreria PThreads ci permette di operare con i thread in c. Vediamo come è strutturata:

- **Gestione dei thread:** creazione, distruzione e join di thread
- **Gestione dei mutex:** creazione, distruzione, lock e unlock dei mutex per sezioni critiche
- **Condition variables:** creazione, distruzione, wait e signal su variabili condition

### 16.1 Gestione dei thread

#### 16.1.1 Create/Exit

La funzione che ci permette di creare thread è la `pthread_create()`.

```
pthread_create(id, attr, start_routine, arg);
```

dove

- id: di tipo `pthread_t` è un identificatore del thread creato
- attr: imposta gli attributi del thread
- start\_routine: è un puntatore alla funzione che verrà eseguita dal thread
- arg: è un puntatore passato come parametro di ingresso alla startign routine

Per terminare un thread si utilizza la funzione `pthread_exit()`

```
pthread_exit(status);
```

Si utilizza per terminare esplicitamente un thread, e status rappresenta lo stato di uscita del thread.

### 16.1.2 Passaggio di parametri

La `pthread_create()` può passare un singolo argomento di tipo `void *`, per passare più di un argomento al thread occorre definire una struct.

```
struct dati {
    int dato1;
    char dato2;
}
```

Nel thread padre, si alloca sull'heap (`malloc`) una istanza della struct e si passa il puntatore al thread figlio.

```
struct dati *d=
    (struct dati *) malloc(sizeof(struct dati));
d -> dato1 = 10
d -> dato2 = 'x';

pthread_create(&id, NULL, start_func, (void *) d)
```

Per usare correttamente la struttura dati nel thread figlio, occorre fare il casting inverso del puntatore `void *`

```
void * start_func(void * arg){
    struct dati *dati = (struct dati *) arg;
    printf("Dato1: %d\n", dati->dato1)
}
```

### 16.1.3 Join

L'operazione `join` permette di sincronizzare un thread padre con uno o più thread figli.

La chiamata `pthread_join(threadId, status)`, blocca il chiamante (padre) finché il `threadId` specificato non termina.

```
pthread_join(id, NULL)
```

Con il parametro null, il padre rinuncia a ricevere lo stato di uscita dal figlio, mentre per riceverlo si utilizza

```
pthread_join(id, (void **) &status)
```

### 16.1.4 Restituzione dei dati

Il thread figlio può passare dei dati in uscita al thread padre tramite una struct sull'area heap

```
struct exit_data{
    int res
};

void * figlio(void *){
    struct exit_data *status=
        malloc(sizeof(struct exit_data))
    status->res = 10

    pthread_exit(status);
}

int main(){
    // thread padre
    struct dati_uscita *status;
    pthread_join(..., &status);

    int result = status->res
}
```

## 16.2 Mutex

### 16.2.1 Creazione e distruzione

Per mutex si intendono semafori binari che possono assumere lo stato di Lock o Unlock.

- **pthread\_mutex\_t**: rappresenta il tipo mutex
- **pthread\_mutex\_init(mutex, attr)**: inizializza un nuovo mutex come sbloccato
- **pthread\_mutex\_destroy(mutex)**: disattiva un mutex

I mutex vengono gestiti attraverso delle operazioni di base

- **pthread\_mutex\_lock(mutex)**: Un thread invoca la lock su un mutex per acquisire l'accesso in mutua esclusione alla sezione critica relativa al mutex. Se già un altro thread ha acquisito il mutex, il chiamante si blocca in attesa di un unlock.
- **pthread\_mutex\_unlock(mutex)**: Un thread invoca la unlock su un mutex per rilasciare la sezione critica, e consentire l'accesso ad un altro thread
- **pthread\_mutex\_trylock(mutex)**: Come la lock, ma se il mutex è già acquisito ritorna immediatamente con codice di errore EBUSY