

# Sistemi Operativi

Giacomo Simonetto e Diego Chiesurin

Secondo Semestre 2024-25

## **Sommario**

Appunti del corso di Sistemi Operativi della facoltà di Ingegneria Informatica dell'Università di Padova.

# Indice

<b>1 Introduzione e struttura dei sistemi operativi</b>	<b>4</b>
1.1 Introduzione e compiti di un sistema operativo . . . . .	4
1.2 Caricamento del sistema operativo . . . . .	4
1.3 Servizi offerti dal sistema operativo . . . . .	5
1.4 Progettazione e sviluppo dei SO . . . . .	6
1.5 Gestione delle periferiche e delle risorse . . . . .	9
1.6 Strutture dati del kernel . . . . .	10
1.7 Protezione e sicurezza . . . . .	10
1.8 Virtualizzazione ed emulazione . . . . .	11
1.9 Sistemi embedded . . . . .	11
1.10 Sistemi operativi open source . . . . .	12
<b>2 Ripasso Hardware</b>	<b>13</b>
2.1 CPU . . . . .	13
2.2 Memoria . . . . .	14
2.3 Concetti di sistemi e reti . . . . .	15
2.4 Sistemi multiprocessore, multiscaleare e parallelismo . . . . .	16
2.5 Coprocessori: GPU, TPU, FPGA . . . . .	18
<b>3 Concorrenza</b>	<b>19</b>
3.1 Processi e Thread . . . . .	19
3.2 Concorrenza vs Parallelismo . . . . .	20
3.3 Diagrammi temporali . . . . .	21
3.4 Risorse . . . . .	22
3.5 Deadlock . . . . .	23
3.6 Prevenzione del deadlock . . . . .	24
3.7 Race condition e sezioni critiche . . . . .	25
3.8 Lock Mutex . . . . .	25
3.9 Semafori . . . . .	26
3.10 Monitor . . . . .	27
3.11 Parallelismo e concorrenza in Linux . . . . .	29
3.12 Memory Barrier . . . . .	30
<b>4 Scheduling CPU</b>	<b>31</b>
4.1 Introduzione . . . . .	31
4.2 Comunicazione tra processi . . . . .	32
4.3 Algoritmi di scheduling . . . . .	33
4.4 Code multiple . . . . .	34
4.5 Scheduling multiprocessore . . . . .	34
4.6 Nei sistemi operativi attuali . . . . .	35
<b>5 Gestione memoria principale</b>	<b>36</b>
5.1 Concetti base . . . . .	36
5.2 Gestione della memoria e degli indirizzi dei programmi . . . . .	36
5.3 Paginazione . . . . .	37
5.4 Segmentazione . . . . .	39
5.5 Paginazione e segmentazione nei processori Intel e ARM . . . . .	39
5.6 Memoria virtuale e swap . . . . .	40
5.7 Gestione dei frame tra i processi attivi e Working Set . . . . .	42
5.8 Allocazione della memoria del kernel . . . . .	43
5.9 Mappatura della memoria e gestione dell'I/O . . . . .	43
5.10 Gestione della memoria in Linux . . . . .	44
5.11 Gestione della memoria in Windows . . . . .	44
<b>6 Gestione file</b>	<b>45</b>

<b>7 Affidabilità</b>	<b>46</b>
7.1 Nozioni introduttive . . . . .	46
7.2 Formule matematiche . . . . .	47

# 1 Introduzione e struttura dei sistemi operativi

## 1.1 Introduzione e compiti di un sistema operativo

Il sistema operativo è un software che agisce da intermediario tra l'utente e l'hardware del computer, deve fornire servizi e caratteristiche presenti a tutti i suoi utenti attraverso l'interfacciamento di hardware e applicativi e deve gestire le varie risorse hardware a disposizione.

### Funzioni del sistema operativo / problemi che deve risolvere

- fornire all'utente applicativi (programmi e comandi) per far eseguire operazioni all'elaboratore
- coordinare l'esecuzione dei programmi dell'utente e dei processi del sistema (es. multithreading)
- gestire le risorse (memoria, I/O, ...) efficientemente e prevenire errori dal loro uso improprio

### Architettura di un sistema di calcolo

Il sistema di calcolo è l'insieme di hardware, sistema operativo, programmi di sistema e applicativi con la possibilità di accedere alle risorse hardware.

### Struttura a cipolla

La struttura del sistema di calcolo è descritta con il modello a cipolla, composto da una serie di gusci concentrici, che racchiudono l'hardware, posto al centro, e i diversi strati di software che lo gestiscono/utizzano. I gusci rappresentano programmi, che operano a livelli diversi di interazione uomo-macchina, ogni strato sfrutta gli strati sottostanti e fornisce funzioni per quelli superiori. L'utente interagisce solo con lo strato più esterno detto shell, mentre lo strato più interno a diretto contatto con l'hardware è detto kernel.

I SO dei dispositivi mobili comprendono anche un middleware, ovvero uno strumento che si posiziona tra hardware e software per semplificare la programmazione ed aumentare la portabilità siccome gli hardware sono estremamente diversificati.

## 1.2 Caricamento del sistema operativo

### Caricamento con BIOS

Il bootstrap consiste in tutte le operazioni e programmi che vengono eseguiti dall'accensione allo spegnimento di un computer. In fase di accensione il SO viene caricato attraverso una procedura che utilizza il BIOS (ormai non più usata). Il BIOS (Basic Input Output System) o firmware è salvato nella ROM ed è costituito da routine software che forniscono funzioni base eseguibili sull'hardware. La prima procedura prevede che

1. il BIOS esegue il caricamento del bootloader dalla partizione MBR (Master Boot Record) del disco
2. il bootloader inizializza (e controlla) tutte le componenti del sistema e procede al caricamento del kernel del SO e ne lancia l'esecuzione
3. il kernel avvia i programmi di sistema, che garantiscono i servizi fondamentali per l'esecuzione dei programmi utente

### Caricamento con UEFI

Nei computer moderni è ancora presente il BIOS, ma la procedura UEFI (Unified Extensible Firmware Interface) ne controlla l'esecuzione attraverso una interfaccia grafica, e ne estende le funzionalità con la possibilità di indirizzare più memoria ed eseguire compiti più complessi come strumenti per la diagnostica e il ripristino dei dati, servizi di crittografia e funzionalità per la gestione dei consumi. La procedura UEFI prevede che:

1. l'UEFI carica il bootloader dalla cartella EFI nella partizione del disco dedicata al sistema operativo (non più da una partizione apposita)
2. il bootloader carica il kernel che a sua volta caricherà il resto del SO

## 1.3 Servizi offerti dal sistema operativo

### Servizi user oriented

- **interfaccia utente:** strumento con cui l'utente interagisce con il computer, a riga di comando (Command Line Interface) e/o grafica (Graphical User Interface)
- **esecuzione di programmi:** caricare ed eseguire comandi correttamente, su richiesta dell'utente
- **operazioni di I/O:** interazione con le periferiche connesse
- **gestione del filesystem:** creare, leggere, scrivere, cancellare files e muoversi tra le directory
- **comunicazioni:** gestione dello scambio di informazioni tra processi in esecuzione
- **rilevamento di errori:** individuare e agire opportunamente se si verificano errori

### Servizi per il funzionamento corretto ed efficiente del sistema

- **gestione delle risorse:** assegnazione delle risorse e monitoraggio del loro utilizzo da parte del sistema e degli utenti
- **protezione:** garantire un accesso controllato alle risorse
- **sicurezza:** fornire mezzi di autenticazione e proteggere le risorse da accessi illegali

### Interfaccia CLI

Interfaccia utente a riga di comando che permette all'utente di impartire comandi al sistema. Il sistema può offrire più shell, ovvero ambienti CLI diversi (bash, zsh) e incorpora un interprete dei comandi che esegue i comandi richiesti dall'utente. I comandi possono far parte built-in dell'interprete, oppure possono essere customizzati dall'utente.

### Interfaccia GUI

L'interfaccia grafica permette una più intuitiva iterazione con l'utente tramite mouse, tastiera e monitor. I files, le cartelle e i programmi sono rappresentati da icone con cui è possibile interagire con il mouse. Le prime interfacce grafiche sono state sviluppate dalla Xerox PARC di Palo Alto e sono state adottate subito dalla Apple.

### Chiamate a sistema o system call

Le chiamate a sistema sono gli strumenti che permettono di accedere ai servizi del sistema operativo, sono realizzate in linguaggi ad alto livello (C, C++) e sono richiamate attraverso le API (Application Programming Interface). Esempi di API sono: Win64 API per Windows, POSIX API per Unix, Linux e macOS e Java API per Java Virtual Machine.

Quando viene effettuata una chiamata al sistema, avviene il cambio di mode bit (da user mode a kernel mode) e quando la routine del sistema termina, restituisce il risultato al programma chiamante ritorna al bit mode iniziale (da kernel mode a user mode).

Il passaggio dei parametri alle chiamate a sistema può avvenire in tre modi: salvandoli nei registri, salvandoli in un'area di memoria e passare l'indirizzo di memoria in un registro (preferita da Linux e Solaris) oppure tramite lo stack

Esempi di alcune chiamate a sistema in Linux:

- `fork, exit, wait/waitpid, exec execve, signal, kill` per controllo dei processi
- `open, read, write, close` per gestione dei files
- `ioctl, read, write` per gestione dei dispositivi
- `getpid, ps, alarm, sleep` per recupero informazioni
- `pipe, shmget, mmap` per comunicazione tra processi
- `alloc, free` per assegnazione e rilascio della memoria

## Programmi di sistema e servizi

I programmi di sistema sono una interfaccia conveniente e semplificativa delle system call. Vengono invocati da terminale e sono lo strumento con cui l'utente interagisce con il sistema. Si dividono in due categorie:

- **servizi in background** (servizi, sottosistemi o daemon) che vengono avviati in fase di boot e permettono il corretto funzionamento del sistema, vengono eseguiti in modalità utente
- **programmi applicativi** non fanno parte del sistema, ma sono scaricati dall'utente

## Compilazione, ABI, linker e loader

Il risultato della **compilazione** è specifico di ogni sistema, ovvero non è universale per tutti i SO, su Unix si utilizza il formato ELF (Executable and Linkable Format), in Windows si usa il formato PE (Portable Executable) e in macOS si usa Mach-O.

I dettagli di come un file compilato agisce sul sistema operativo (chiamate a sistema, passaggio dei parametri, formato degli eseguibili, gestione della memoria, ...) sono contenute nelle **ABI** (Application Binary Interface).

Il **linker** è lo strumento che unisce i diversi file oggetto rilocabili ottenuti dalla compilazione di un determinato programma e dalle relative librerie statiche. Nei moderni sistemi si predilige il linking dinamico delle librerie dinamiche (DLL in Windows) che vengono caricate e condivise dai vari programmi in base alle necessità.

Il **loader** è lo strumento che carica il file eseguibile e le necessarie dipendenze dalla memoria secondaria alla memoria primaria.

## 1.4 Progettazione e sviluppo dei SO

### Linguaggi con cui è scritto un sistema operativo

Il sistemi vengono sviluppati con un mix di linguaggi per favorirne la portabilità, la leggibilità e la facilità nella manutenzione: Si usa assembly per i driver dei dispositivi, il C per il kernel e altri linguaggi di alto livello come C, C++, shell script, Python per programmi di sistema e applicativi.

### Sistemi monolitici

Nei sistemi monolitici si ha che tutte le funzioni di gestione delle risorse sono realizzate nel kernel e i diversi moduli sono molto legati tra di loro. Lo svantaggio è che, siccome tutti lavorano nella stessa area di memoria, un errore di un modulo può comportare il blocco dell'intero sistema. Il vantaggio è che un sistema monolitico è molto efficiente.

Esempi di sistemi monolitici:

#### - MS-DOS:

era nato con l'idea di offrire il maggior numero di funzionalità nel minor spazio possibile. Non è suddiviso in moduli e non c'è una netta separazione tra interfacce e funzionalità. Gli applicativi accedono direttamente alle routine di sistema del BIOS, senza nessuna protezione, in quanto l'Intel 8088 (hardware su cui girava MS-DOS) non aveva protezione hardware tramite i bit mode.

#### - UNIX:

è un sistema scarsamente stratificato (date le limitate funzionalità hardware di quando è stato progettato) ed è diviso in due parti: i programmi di sistema e il kernel (che incorpora tutto ciò che si trova tra le system call e l'hardware).

### Approccio stratificato

L'approccio stratificato consiste nell'organizzare i vari moduli del sistema su più livelli in cui ogni livello può interagire con quelli sottostanti e fornisce funzionalità e servizi per quelli più esterni. Il livello più interno 0 è l'hardware, quello più esterno è l'interfaccia utente. Il vantaggio è nella semplicità di realizzazione e manutenzione, gli svantaggi sono la difficile caratterizzazione degli strati (uno strato può usare solo funzionalità di quello immediatamente inferiore) e la scarsa efficienza in quanto è necessario attraversare più strati per ogni system call.

## **Microkernel**

I sistemi microkernel sono composti da un kernel minimalista che si occupa solo di gestire processi, memoria e comunicazione, mentre lascia il resto (applicativi, filesystem e device driver) al lato utente. In questo modo l'aggiunta di nuove funzionalità (dal lato utente) non modifica il kernel, è più facile da mantenere portare su nuove architetture, è più sicuro e affidabile in quanto poco codice viene eseguito in modalità kernel. Lo svantaggio è il possibile rallentamento causato dall'overhead tra modalità utente e modalità kernel. Esempi di microkernel sono Windows NT, Mach (incluso in Darwin), GNU Hurd.

## **Kernel modulari**

I kernel modulari o Loadable Kernel Modules (LKMs) si organizzano in moduli che implementano componenti base con relative interfacce, possono comunicare tra di loro e possono essere caricati in memoria solo quando richiesto. Rispetto all'architettura a strati, quella a moduli è più flessibile perché non si ha l'obbligo di attraversare i diversi strati, sono facili da mantenere e da evolvere. Esempi sono Linux (periferiche e filesystem) e Solaris.

## **Sistemi ibridi**

I sistemi ibridi combinano diversi approcci allo scopo di migliorare performance, manutenibilità e usabilità. Alcuni esempi sono:

- **Linux e Solaris** sono costituiti da kernel monolitici, ma prevedono la possibilità di nuovi moduli da caricare in runtime
- **Windows** è di base monolitico, ma conserva alcuni comportamenti di sistemi microkernel come il supporto a sottosistemi separati (personalità) eseguibili in modalità utente

## **macOS**

```
|  
+- Darwin: core open source di macOS  
|   +- XNU: kernel ibrido "X is Not Unix"  
|   |   +- Mach: microkernel per gestione di memoria, scheduling thread, RPC e IPC  
|   |   +- BSD: sottosistema Unix-derived per gestione di filesystem, rete e API POSIX  
|   |   +- IOKit: moduli dinamici per la gestione dei driver (estensioni kernel)  
|   +- librerie C di basso livello: libc, dyld, ecc.  
|   +- utility di sistema base: shell, tool UNIX, ecc.  
+- framework grafico Cocoa: GUI, AppKit, UIKit  
+- applicazioni di sistema: Finder, Safari, ecc.  
+- servizi Apple proprietari: Spotlight, iCloud, ecc.
```

- Da Mac OS X (2001), il SO è costruito secondo la struttura sopra, con un kernel ibrido dotato di microkernel, sottosistema Unix e moduli dinamici per gestione dei driver.
- Le chiamate di procedura remota (RPC, Remote Procedure Calls) sono un meccanismo di comunicazione usato in informatica per permettere a un programma di eseguire una funzione o procedura che risiede su un altro computer o processo, come se fosse una chiamata locale.
- La comunicazione tra processi (IPC, Inter-Process Communication) è un insieme di meccanismi che permettono di gestire lo scambio di dati e informazioni tra due processi concorrenti (pipe, named pipe, message queue, shared memory, semafori, socket e signal).
- Le API POSIX (Portable Operating System Interface) sono un insieme di specifiche standard per le interfacce di programmazione dei sistemi operativi, progettate per garantire la portabilità delle applicazioni tra diversi sistemi Unix-like e altri sistemi compatibili. Grazie a POSIX, gli sviluppatori possono creare software che funziona su Linux, macOS, BSD e altri sistemi conformi, senza dover riscrivere codice specifico per ogni piattaforma.
- Il framework grafico Cocoa è l'insieme degli strumenti per creare interfacce grafiche nativamente in macOS. L'interfaccia utente predefinita di sistema si chiama Aqua che viene mantenuta sempre aggiornata e determina il look e il feel del sistema.

## iOS

iOS si basa sullo stesso core di macOS chiamato Darwin integrato con:

- il framework grafico Cocoa Touch apposito per i dispositivi touch
- i media service per le applicazioni multimediali (grafica, video, audio)
- i core services per supporto al cloud computing e ai database

## Android

Android è un sistema sviluppato dalla Open Handset Alliance guidata da Google, insieme a Asus, Htc, Intel, Motorola, Qualcomm, T-Mobile, Samsung e Nvidia. È un sistema a struttura stratificata basato su un kernel Linux modificato. I vari programmi e servizi vengono eseguiti sull'ART (Android Run Time), ovvero su macchina virtuale. Le app sono sviluppate in Java e tradotte in eseguibili per la ART virtual machine. È disponibile anche l'interfaccia JNI (Java Native Interface) per bypassare la ART e avere accesso diretto all'hardware. Nel sistema è incluso uno strato di astrazione dell'hardware HAL (Hardware Abstraction Layer).

## Windows Subsystem for Linux - WSL

Grazie all'architettura ibrida di Windows è possibile eseguire sottosistemi in modalità utente in grado di emulare altri SO. Nel caso del WSL, viene avviata una istanza di Linux in grado di eseguire le applicazioni native di Linux grazie alla presenza dei servizi kernel LXCore e LXSS che traducono le system call Linux in system call Windows (es. fork in CreateProcess) e in mancanza di una corrispondenza esatta, fornisce una funzionalità equivalente.

## Debugging

Il debugging è l'attività di individuazione e risoluzione dei bachi (bug) e di performance tuning. Il SO genera diversi file con i dati runtime dei processi attivi e dell'intero sistema:

- **file di log:** informazioni sugli errori durante l'esecuzione di un processo
- **core dump:** immagine della memoria di un processo al momento della sua terminazione anomala
- **crash dump:** immagine completa della memoria nell'istante in cui si verifica un crash, ovvero un guasto nel kernel, spesso si ricorrono a tecniche particolari (salvare l'immagine in un'area di memoria dedicata) per evitare di compromettere il filesystem in caso di kernel in stato inconsistente

## Performance tuning

I problemi che coinvolgono le prestazioni del sistema sono considerati bachi e il performance tuning si occupa di individuarli e risolverli per ottimizzare le prestazioni ed eliminare i colli di bottiglia. Ciò viene fatto tramite strumenti per monitorare gli eventi in rilievo, le risorse in uso (es. comando top di Unix) e le system call più frequenti.

## 1.5 Gestione delle periferiche e delle risorse

### Interfacciamento tramite driver, controller e interrupt

L'interfacciamento tra hardware e programmi applicativi avviene secondo il seguente schema:

applicativi  $\leftrightarrow$  kernel  $\leftrightarrow$  kernel I/O subsystem  $\leftrightarrow$  drivers  $\leftrightarrow$  controllers  $\leftrightarrow$  risorsa/dispositivo

- **controller**: componente hardware specifico per ogni risorsa che gestisce un buffer per l'input e l'output dei dati dalla risorsa all'elaboratore
- **driver**: componente software specifico di ogni risorsa che permette al kernel di interfacciarsi con un determinato controller e fornisce le indicazioni sullo scambio dei dati tra elaboratore e risorsa

### Device driver

Un device driver (o controllore di periferica), è un componente software di basso livello utilizzato per comunicare con le periferiche connesse al computer.

- **driver di sistemi embedded**: tutto il software è un unico programma compilato e caricato in ROM, il driver non è altro che una routine del programma che si interfaccia con l'hardware da pilotare
- **driver in kernel monolitici** (es. Linux): i driver sono moduli compilati insieme al kernel, se si vuole collegare una periferica per cui non è presente un driver, è necessario aggiungerlo e ricompilare il kernel (dalle nuove versioni non è più richiesta la ricompilazione), questo porta al vantaggio che i driver già presenti nel kernel sono testati dai programmatori del sistema operativo offrendo maggiore efficienza e affidabilità
- **driver in kernel ibridi** (es. Windows): i driver sono file binari caricati dinamicamente dal kernel al momento del bisogno, questa scelta permette una maggiore compatibilità e flessibilità nella gestione delle periferiche in quanto è sufficiente che il produttore fornisca il driver per la propria periferica e si è certi di poterla utilizzare, lo svantaggio è che i driver potrebbero non essere ottimizzati ed efficienti, non essendo parte del kernel

### Interrupt

La comunicazione tra controller e driver/kernel avviene tramite gli interrupt. Esiste un vettore con puntatori alle procedure da eseguire quando si riceve un interrupt (chiamate Interrupt Service Routines o ISR) e che gestiscono i flussi di dati tra risorse e sistema.

### Input/Output

- **gestione sincrona**: la CPU aspetta fino a quando il dato non è pronto in quanto è la CPU che gestisce i trasferimenti
- **gestione asincrona**: la CPU delega al DMA lo spostamento dei dati e nel frattempo può eseguire altre operazioni

### DMA, RDMA e GPU Direct Storage

Il DMA (Direct Memory Access) è un componente hardware che esegue il trasferimento di blocchi di dati dalla memoria secondaria o dal buffer locale di qualche dispositivo alla memoria principale. Viene gestito tramite interrupt e funziona come segue:

1. vengono specificati l'indirizzo, la dimensione dei dati sorgente e l'indirizzo di destinazione
2. il DMA si occupa del trasferimento, inviando un interrupt per ogni blocco trasferito, lasciando il processore libero di eseguire altri processi

Alcune varianti del DMA applicate a casi particolari:

- **RDMA o Remote Direct Memory Access** è un sistema che permette il trasferimento dei dati tra le RAM di diversi elaboratori collegati insieme, in ambito multicomputer con parallelizzazione
- **GPU Direct Storage** è un protocollo che prevede il trasferimento dei dati dalla memoria secondaria direttamente alla GPU, senza passare per la RAM, introdotto in Windows11 e inizialmente sviluppato per XBox.

## 1.6 Strutture dati del kernel

### Strutture dati classiche

- **array**: semplice struttura dati in cui ogni elemento è direttamente accessibile tramite un indice. La memoria principale è costruita come un array in cui si accede alle celle tramite un indirizzo di memoria
- **liste concatenate**: possono contenere dati di diversa natura e dimensione e sono talvolta utilizzate direttamente dagli algoritmi del kernel, ad esempio nell'allocazione concatenata di file
- **stack**: struttura LIFO (Last In First Out) utilizzata ad esempio per le chiamate a funzione (RDA) e il cambio di contesto
- **code**: struttura FIFO (First In First Out) utilizzata per gestire i documenti inviati ad una stampante o i processi in attesa di ottenere l'accesso alla CPU
- **alberi**: strutture basate sulla relazione causale padre-figlio, in genere si utilizzano gli alberi binari di ricerca ad esempio in Linux per l'algoritmo di scheduling della CPU

### Tabelle e funzioni hash

Una funzione hash è una funzione che trasforma ogni valore di chiave in un indirizzo o bucket.

- le funzioni di hash non sono iniettive per cui possono verificarsi **collisioni** se due chiavi distinte vengono mappate sullo stesso indirizzo, trovare funzioni che riducono al minimo le collisioni è molto complesso e molto dispendioso, per gestire le collisioni si utilizzano per ogni bucket delle catene o liste di trabocco (liste concatenate) per memorizzare le chiavi che collidono
- la **capacità** di un bucket è il numero di chiavi che può contenere (in caso di collisioni)
- l'**overflow** si verifica quando si tenta di inserire una chiave all'interno di un bucket già pieno
- l'**area di overflow** è un'area di memoria utilizzata per memorizzare i dati che hanno generato overflow
- l'**area primaria** è l'area di memoria con i bucket indirizzabili dalla tabella di hash

Per convertire una stringa alfanumerica  $s$  si sceglie una determinata funzione che mappa ogni carattere in un intero distinto  $s_i$  e, dopo aver scelto una base  $b$ , si associa il corrispettivo peso ad ogni carattere. Infine si sommano i risultati ottenuti:

$$k(s) = \sum_{i=0}^{n-1} s_i \cdot b^i$$

## 1.7 Protezione e sicurezza

### Protezione delle risorse

Il SO deve controllare che i programmi accedano solo alle risorse (file, segmenti di memoria, CPU, ...) di cui hanno ottenuto l'apposita autorizzazione. Inoltre deve verificare che solo gli utenti autorizzati accedano alle risorse di cui hanno accesso.

### Mode bit

Ogni operazione è associata ad un livello di protezione detto mode bit. Il mode bit è gestito via hardware e indica lo stato di protezione entro cui una determinata operazione può essere eseguita.

- **user mode** è uno stato di privilegio caratterizzato da un numero relativamente basso di privilegi verso la memoria, l'hardware e altre risorse
- **kernel mode** è lo stato di privilegio massimo riservato all'esecuzione del kernel. Il codice in linguaggio macchina eseguito in tale modalità ha accesso illimitato alla memoria, all'hardware e alle altre risorse

Se ci si trova in uno stato con alto livello di protezione (user mode) e si desidera eseguire un'operazione che richiede un livello inferiore (kernel mode), è necessario passare ad un mode bit inferiore tramite un processo di **trap**. Una volta completata l'operazione si effettua un processo di **return** per ripristinare il livello di protezione superiore. Ogni system call effettua il passaggio in modalità kernel; il ritorno dalla chiamata riporta il sistema in modalità utente.

## Sicurezza - Protezione da attacchi informatici

Il SO deve implementare dei sistemi di difesa per contrastare attacchi interni ed esterni classificati in:

- **denial of service**: malfunzionamento dovuto ad un attacco informatico in cui si esauriscono deliberatamente le risorse di un sistema di calcolo che fornisce un servizio, fino a renderlo non più in grado di erogare il servizio
- **trojan**: programmi che hanno una funzione conosciuta legittima e una funzione dannosa nascosta
- **worm**: malware (software usato per disturbare le operazioni svolte da un computer, rubare informazioni sensibili, accedere a sistemi informatici privati, o mostrare pubblicità indesiderata) in grado di autorePLICarsi
- **virus**: porzioni di codice dannoso che si legano ad altri programmi del sistema per diffondersi, un virus può anche essere un malware

Gli attacchi sono svolti da:

- **hacker**: pirata informatico che si impegna nell'affrontare sfide intellettuali per aggirare o superare creativamente le limitazioni di accesso ai sistemi per esplorare, divertirsi, apprendere, senza creare danni reali
- **cracker**: chi si ingegna per eludere blocchi imposti da qualsiasi software al fine di trarne guadagno

## 1.8 Virtualizzazione ed emulazione

### Virtualizzazione in generale

La virtualizzazione/emulazione è una tecnica che permette di eseguire un sistema operativo come applicazione all'interno di un altro SO attraverso una macchina virtuale (VM). La VM crea un ambiente virtuale isolato che riproduce tipicamente il comportamento di una macchina fisica grazie all'assegnazione di risorse hardware (porzioni di disco rigido, RAM e CPU) gestite dall'hypervisor o Virtual Machine Manager (VMM).

Tra i vantaggi vi è il fatto di poter offrire contemporaneamente ed efficientemente a più utenti diversi ambienti operativi separati, attivabili su richiesta, senza sporcare il sistema fisico reale con partizionamenti del disco rigido oppure ricorrendo ad ambienti clusterizzati su sistemi server. Lo svantaggio è che, essendo le macchine virtuali isolate, non possono condividere le risorse nativamente, a meno che non si utilizzino determinati software come Virtio.

La virtualizzazione (per le CPU che la supportano) può essere vista come una terza modalità del mode bit con più privilegi dei processi utente in quanto lavora attivamente con le risorse assegnate, ma meno del kernel del SO ospitante in quanto deve sempre sottostare al kernel.

### Virtualizzazione vs emulazione

L'emulazione è il processo per cui vengono riprodotte risorse hardware simulate che non corrispondono necessariamente all'architettura reale dell'elaboratore (es. emulazione x86 su ARM). Questo processo è più costoso in quanto le istruzioni da eseguire sono interpretate, ma permette di avere a disposizione molte architetture differenti.

La virtualizzazione consiste nel riservare parte delle risorse dell'hardware reale per l'ambiente virtualizzato. In questo modo non è necessario tradurre le istruzioni da una architettura ad un'altra, rendendo l'esecuzione più rapida, ma è possibile eseguire solo codice compilato per l'architettura reale in possesso.

## 1.9 Sistemi embedded

I sistemi embedded sono sistemi ubiquitari, ovvero presenti ovunque (elettrodomestici, auto, robot aziendali), sono eseguiti su sistemi rudimentali e hanno le seguenti caratteristiche:

- sistemi standard che girano su elaboratori general-purpose con implementate specifiche funzionalità
- sistemi special-purpose che girano su microprocessori
- Application Specific Integrated Circuit (ASIC) senza un vero SO

Le applicazioni prevedono dei requisiti stringenti sul tempo con cui determinate operazioni devono essere portate a termine, ovvero sono sistemi hard real-time.

## 1.10 Sistemi operativi open source

I sistemi operativi open source sono sistemi che vengono distribuiti sia in formato binario compilato, sia in codice sorgente. Non è necessario effettuare il reverse engineering per comprendere il funzionamento del sistema. Si forma una comunità di programmatore e aziende che contribuiscono allo sviluppo, al debugging, all'assistenza e al supporto gratuito agli utenti. Il codice più sicuro e i bug sono scoperti e risolti velocemente.

### Richard Stallman, GNU e Linux

Richard Stallman nel 1984 da origine al progetto GNU (GNU is Not Unix), ovvero l'idea di ricreare un intero sistema operativo analogo a UNIX, ma libero, con licenza GLP (General Public Licence). La GPL indica che ogni componente software può essere utilizzata, modificata e distribuita mantenendo sempre la licenza GPL, ovvero non è possibile utilizzarlo in codice proprietario. Ad oggi il progetto GNU ha numeri strumenti compatibili con Unix, ma il kernel Hurt (GNU) è ancora in fase di sviluppo. Per i sistemi operativi liberi si utilizza il kernel Linux sviluppato da Linus Torvald usando strumenti di GNU, ma che non fa parte del progetto GNU.

### Open source e software libero

Il software libero (GLP) indica che può essere utilizzato, modificato e distribuito liberamente, sempre sotto licenza GPL, ovvero non può mai diventare software proprietario. Il software open source è analogo al software libero, solo che può essere anche utilizzato in progetti proprietari.

### Esempi di sistemi operativi open source / liberi

- **Linux**: open source con alcune distribuzioni libere e altre solo open source
- **Windows**: proprietario e closed-source
- **macOS**: ibrido in quanto basato su kernel Darwin open-source basato su UNIX BSD, con l'aggiunta di componenti proprietarie
- **UNIX BSD**: derivato da Unix, non è open source in quanto è richiesta la licenza, ma era inizialmente distribuito insieme al sorgente, esistono diverse distribuzioni (FreeBSD, NetBSD, ...)
- **MINIX**: sistema operativo sviluppato da Andrew Tanenbaum in supporto al suo corso universitario, ma impiegato a sua insaputa all'interno dell'Intel Management Engine, data la sua leggerezza

## 2 Ripasso Hardware

### 2.1 CPU

#### Struttura della CPU

La CPU o Control Processing Unit è l'unità di calcolo e controllo dell'elaboratore. È composta da:

- **Control Unit o CU**: unità di controllo che comprende il Program Counter e Instruction Register
- **Arithmetic Logic Unit o ALU**: unità che esegue operazioni matematiche e logiche
- **registri**: aree di memoria interne al chip, utilizzate nell'esecuzione delle istruzioni, come Program Counter (PC), Instruction Register (IR), Memory Address Register (MAR), Memory Data Register (MDR) e altri registri per gli operandi
- **bus dei segnali di controllo**: trasferimento unidirezionale di segnali di controllo dal processore alla memoria (read/write)
- **bus degli indirizzi**: trasferimento unidirezionale degli indirizzi dal processore alla memoria
- **bus dei dati**: trasferimento bidirezionale dei dati tra processore e memoria

#### Funzionamento

Alcune nomenclature sul funzionamento:

- **fetch-decode-execute**: ciclo di valutazione delle istruzioni in una CPU
- **data path**: processo che comprende il recupero dei dati dai registri, l'esecuzione dell'operazione da eseguire da parte della ALU e memorizzazione del risultato sul nuovo registro; il ciclo di data path corrisponde al ciclo di clock nelle architetture non parallelizzate
- **durata istruzione ISA**: ogni istruzione viene eseguita in un multiplo di cicli di data path

#### Prestazioni

- **tempo di esecuzione** dato dal  $T_{\text{clock}}$  tempo di un ciclo di clock o di data path, da  $N_i$  numero di istruzioni di tipo  $i$  e da  $CPI_i$  numero di cicli di clock per istruzioni di tipo:  $i$

$$T_{\text{exec}} = T_{\text{clock}} \cdot \sum_{i=0}^n N_i \cdot CPI_i$$

- **MIPS o Mega Instructions Per Second** indica il numero di milioni di istruzioni eseguibili in un secondo non è molto preciso in quanto non tiene conto delle istruzioni offerte, della variazione di  $CPI$  per le diverse istruzioni e della velocità del buffer, si calcola come segue:

$$M_{\text{ega}} I_{\text{nstructions}} P_{\text{er}} S_{\text{econd}} = \frac{\text{freq. di clock}}{10^6 \cdot CPI}$$

- **MFLOPS o Mega Floating Point operations Per Second** indica il numero di operazioni in virgola mobile a 32 bit che vengono eseguite in un secondo
- **LINPACK benchmark** misura le operazioni in virgola mobile a 64 bit
- **Speedup** misura l'aumento di prestazioni in percentuale

$$\text{SpeedUp} = \frac{\text{Prest.} A - \text{Prest.} B}{\text{Prest.} B} = \frac{T_B - T_A}{T_A} \quad \text{con} \quad \text{Prestazione} = 1/T_{\text{esecuzione}}$$

- **Legge di Amdhal** indica il miglioramento in funzione del ruolo del componente nel sistema con  $p$  tempo di utilizzo della parte migliorata e  $a$  accelerazione dovuta al miglioramento

$$T_{\text{finale}} = \frac{p}{a} T_{\text{iniziale}} + (1-p) T_{\text{iniziale}}$$

In generale per migliorare le prestazioni di una CPU è possibile agire:

- riducendo il numero di cicli per istruzioni ISA
- aumentando la frequenza di clock
- introducendo il parallelismo (a livello di istruzioni con pipeline e superscalari o a livello di core/CPU)

## 2.2 Memoria

### Gerarchia delle memorie

La memoria si classifica in una struttura gerarchica in ordine decrescente di prestazioni e costo:

1. registri (volatile)
2. cache su più livelli (volatile)
3. memoria principale (volatile), es. RAM
4. memoria a stato solido es. DRAM, NVRAM, SSD, Flash
5. disco rigido
6. nastri magnetici

### Gestione della memoria

Il SO è responsabile delle seguenti attività connesse alla gestione della memoria centrale:

1. tener traccia di quali parti della memoria sono attualmente usate e da chi (controllare l'accesso alla memoria e garantire la versione più recente di ogni file sul disco alle applicazioni che lo richiedono)
2. decidere quali (parti di) processi caricare in memoria quando vi è spazio disponibile
3. allocare e deallocare lo spazio di memoria secondo necessità

### Caching

Il concetto di caching consiste nel creare una copia di dati su un supporto di memoria più veloce, ad esempio dal disco alla RAM, o dalla RAM alla cache. Quando il computer deve accedere alla memoria, per prima cosa controlla i supporti più veloci (registri) e, in assenza del dato, si passa al livello inferiore (cache, RAM, HDD/SSD). Per una migliore prestazione si cerca di avere almeno tra l'80 % e il 99 % degli accessi dalla cache.

### Buffering

Il buffering è un processo per cui si utilizza un'area di memoria chiamata buffer per salvare temporaneamente i dati trasferiti tra dispositivi con velocità diverse (es. RAM e disco), i dati per dispositivi di output o le modifiche ai file prima del salvataggio su disco. In questo modo si migliora l'efficienza del trasferimento e salvataggio dei dati. Differisce dalla cache in quanto si trova nella RAM e funge da area in cui la CPU può memorizzare temporaneamente dati prima che essi vengano trasferiti o riprodotti.

### Spooling

Lo spooling è una tecnica che consiste nel mettere in coda (spool) le richieste di I/O in un'area di memoria dedicata in modo che il sistema operativo le riesca a gestire in modo sequenziale. In questo modo è possibile gestire più attività di I/O contemporaneamente, migliorando l'efficienza del sistema.

### I/O Interleaving

L'I/O Interleaving consiste nel trattare le operazioni di lettura o scrittura che coinvolgono più dispositivi di I/O in modo sequenziale, alternando le operazioni tra di essi anziché completarle su un dispositivo prima di passare al successivo. Questo permette al sistema di lavorare, ad esempio su settori di dischi diversi contemporaneamente, riducendo il tempo complessivo necessario per completare l'operazione. In generale, l'interleaving implica la disposizione non contigua dei dati per migliorare le prestazioni.

### Memoria cache

La memoria cache è una piccola memoria molto veloce e costosa affiancata alla CPU per ridurre notevolmente il tempo impiegato per gli accessi alla RAM. In pratica vengono memorizzate delle copie delle pagine in RAM (dette linee di cache) in modo da poterci accedere velocemente senza sprecare cicli di clock per leggerle dalla RAM. Il processore tenta di leggere sempre i dati dalla cache, se il dato è effettivamente presente si ha un cache hit, altrimenti si ha un cache miss ed è necessario caricare in memoria la pagina con il dato richiesto.

Il principio di utilizzo della cache si basa su:

- **località spaziale**: ovvero se viene richiesto un dato, è probabile che verranno richiesti anche i dati limitrofi, infatti vengono caricate le intere pagine di RAM (delle linee di cache) e non i singoli dati
- **località temporale**: ovvero se in un istante si richiede un determinato dato, è probabile che verrà riutilizzato in un istante successivo, normalmente si usa la politica di rimpiazzo LRU

Spesso si utilizzano più livelli di cache per un migliore trade-off tra velocità e costi:

- **cache L1**: suddivisa in L1-I (istruzioni) e L1-D (dati) situata all'interno dei core (16KB to 128KB)
- **cache L2**: più lenta ed economica, cache condivisa tra i diversi core (256KB and 1MB)
- **cache L3**: più lenta ed economica, situata nella motherboard (2MB to 32MB)

### Gestione del file system

Per gestione del file system significa gestione dell'organizzazione dei file nel disco, il SO è responsabile di:

1. creazione e cancellazione di file e directory
2. supporto alle funzioni elementari per la manipolazione di file e directory
3. associazione dei file ai dispositivi di memoria secondaria
4. backup di file su dispositivi stabili di memorizzazione

## 2.3 Concetti di sistemi e reti

### Sistemi distribuiti

I sistemi distribuiti sono un insieme (eterogeneo) di calcolatori con memoria e clock indipendente (non condivisi) che sono connessi attraverso una rete di comunicazione di uno dei seguenti tipi (in base all'estensione):

1. Wide Area Network (WAN)
2. Metropolitan Area Network (MAN)
3. Local Area Network (LAN)
4. Personal Area Network (PAN)

La comunicazione avviene secondo un dato protocollo (TCP/IP è il più diffuso). Un sistema distribuito fornisce agli utenti l'accesso a varie risorse condivise di sistema in modo da consentire di accelerare l'elaborazione, aumentare la disponibilità di dati e migliorare l'affidabilità. In base al ruolo degli elaboratori nelle reti, queste vengono classificate in:

- **client-server**: i pc fungono da client che richiedono servizi al server
- **peer-to-peer (P2P)**: non esiste la distinzione tra client e server (es. Napster, eMule, servizi VoIP come Skype; bittorrent non è P2P puro perché necessita di server per connettersi alla rete)

### Cloud computing

Il cloud computing indica l'insieme di piattaforme e tecnologie che permettono di archiviare file o di sviluppare/utilizzare programmi e applicazioni direttamente sui server di chi fornisce il servizio, anziché sul proprio dispositivo. Pertanto il cloud computing è una tecnica che permette la fruizione di risorse computazionali, di storage e di applicazioni come servizi di rete. Alcuni esempi di cloud computing sono Google Drive ed EC2 (Elastic Compute Cloud) di Amazon. Le tipologie di cloud computing sono:

1. **cloud pubblico**: disponibile attraverso internet a chiunque si abboni al servizio
2. **cloud privato**: gestito da un'azienda ad utilizzo interno
3. **cloud ibrido**: comprende componenti sia pubbliche che private
4. **SaaS** (Software as a Service): applicazioni fruibili via internet (es. word processor, fogli di calcolo)
5. **PaaS** (Platform as a Service): ambiente software per usi applicativi via internet (server database)
6. **IaaS** (Infrastructure as a Service): server o storage accessibili via internet (spazio per backup)

## 2.4 Sistemi multiprocessore, multiscalare e parallelismo

### Introduzione al parallelismo

Caratteristiche dei diversi tipi di parallelismo:

- **loosely coupled**: poche CPU indipendenti collegate a bassa velocità
- **strongly coupled**: tanti piccoli componenti (ALU, core, ...) collegati ad alta velocità
- **course grained**: il software parallelizzato è grande e non richiede interazioni con altri software
- **fine grained**: si parallelizzano piccole istruzioni che richiedono interazioni costanti
- **interconnessione statica**: se determinata a priori e non cambia nel tempo
- **interconnessione dinamica**: se si basa sulle necessità del momento, con apparecchi come switch

### Sistemi superscalari - a parallelismo di istruzioni

Le architetture superscalari implementano un parallelismo a livello di istruzioni. Permettono di eseguire o iniziare a eseguire contemporaneamente più stati diversi (es. recupero operandi, decodifica, esecuzione, ...) di diverse istruzioni distribuite a diverse unità di elaborazione all'interno del singolo chip.

### Sistemi Hyper-Threading o multithreading

Nei sistemi Hyper-Threading o multithreading, i singoli core dispongono di due unità di elaborazione in grado di eseguire contemporaneamente due thread in parallelo sullo stesso core.

### Sistemi multicore

Un'architettura multicore si raggruppano diverse unità di calcolo (core) in un singolo chip. È più efficienti, perché le comunicazioni, sul singolo chip, sono più veloci e un chip multicore usa molta meno potenza di diversi chip single core. I diversi core nel singolo chip hanno un livello di cache condiviso (L2 o superiore).

### Sistemi con coprocessori

Un'architettura con coprocessori consiste nell'affiancare un'altra unità di elaborazione esterna al processore per svolgere compiti specifici come elaborazione grafica (GPU), processi di rete (schede di rete), crittografia (criptoprocessori).

### Sistemi multiprocessore

Un'architettura multiprocessore è costituita da più processori su una unica scheda madre che condividono la stessa memoria. Bisogna fare attenzione nella gestione del traffico dei dati tra i diversi processori e all'assegnazione delle risorse. L'elaborazione può essere:

- **asimmetrica**: ad ogni processore viene assegnato un compito specifico e un processore principale sovrintende all'intero sistema
- **simmetrica**: ogni processore è abilitato all'esecuzione di tutte le operazioni del sistema

### Sistemi multicomputer

Un'architettura multicomputer è una struttura costituita da più elaboratori (ciascuno indipendente con la propria CPU e memoria) collegati insieme, di solito condividono la stessa memoria secondaria. La gestione del traffico dei dati tra i diversi processori è particolarmente ostico da gestire, però hanno il vantaggio di essere più facili ed economici da implementare rispetto ai sistemi multiprocessore. Esistono alcuni indici di classificazione dei multicomputer:

- **grado o fanout**: numero di interconnessioni di ogni unità, serve per calcolare il fault tolerance
- **diametro**: distanza massima tra due nodi qualsiasi della rete
- **dimensionalità**: numero di assi diversi su cui si sviluppa la rete
- **scalabilità**: proporzionalità tra il numero di processori e le prestazioni della rete

Alcuni esempi di interconnessioni:

- **stella**: tutti i nodi sono connessi ad un nodo centrale, diametro = 2, dimensione = 0
- **albero**: struttura ad albero, diametro = 2h, dimensione = 0
- **interconnessione completa**: tutti i nodi sono connessi tra di loro, diametro = 1, dimensione = 0
- **anello**: i nodi sono posizionati su una circonferenza: diametro =  $2P/2$ , dimensione = 1
- **griglia**: nodi posizionati in una griglia, diametro =  $2(l - 1)$ , dimensione = 2
- **toroide2D**: griglia con interconnessioni tra nodi esterni, diametro =  $l$ , dimensione = 2
- **cubo**: nodi disposti su un cubo, diametro = 3, dimensione = 3
- **iper cubo**: nodi disposti su un ipercubo, buon compromesso di scalabilità in quanto il diametro aumenta logaritmicamente in base al numero di processori, diametro = 4, dimensione = 4
- **toroide3D**: nodi disposti su una griglia tridimensionale, in quanto è un buon compromesso tra diametro della rete e numero di connessioni, diametro = ..., dimensione = 3

## Sistemi cluster

Un cluster è un particolare tipo di multicomputer con particolare attenzione alle interconnessioni e alla coordinazione delle diverse unità. Per usufruire delle potenzialità dei cluster in ambito High Performance Computer (HPC), è necessario che i processi siano scritti per essere parallelizzati. In base alla gestione delle interconnessioni si differenziano in:

- **clustering asimmetrico**: un calcolatore rimane nello stato di attesa attiva mentre gli altri eseguono le applicazioni, il calcolatore in stand by controlla gli altri nodi e si attiva nel caso di malfunzionamenti
- **clustering simmetrico**: tutti i nodi eseguono le applicazioni e si controllano reciprocamente, attraverso ad esempio il protocollo NUMA (Non Uniform Memory Access system)

## Speedup per aumento delle CPU

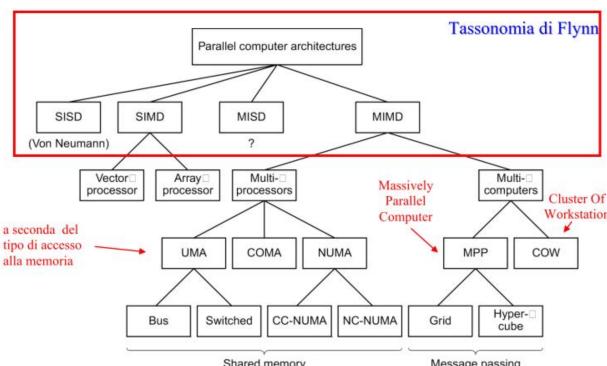
Il tempo di esecuzione di un frammento di codice (parzialmente o totalmente parallelizzato) in un'architettura in grado di eseguire operazioni parallele viene:

$$S_{\text{speedUp}} = \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{n}{1 - (n - 1)f} \quad T_{\text{par}} = f \cdot T_{\text{seq}} \quad \frac{(1 - f)T_{\text{seq}}}{n}$$

$T_{\text{seq}}$  = tempo exec. con 1 CPU  
 $T_{\text{par}}$  = tempo exec. con  $n$  CPU  
 $n$  = numero CPU  
 $f$  = frazione sequenziale del codice

## Classificazione tassonomica di Flynn (1972)

- **SISD**: single instruction sequence, single data sequence
- **SIMD**: single instruction sequence, multi data sequence
- **MISD**: multi instruction sequence, single data sequence
- **MIMD**: multi instruction sequence, multi data sequence



## 2.5 Coprocessori: GPU, TPU, FPGA

### Evoluzione della GPU per general purpose calculation GP-GPU

All'inizio le GPU erano specifiche per eseguire calcoli relativi alla parte grafica (colore dei pixel, ...) usando apposite unità di elaborazione grafica come i pixel shader. Nel 2006 grazie all'architettura CUDA nella scheda grafica GeForce 8800 GTX di NVIDIA anche le schede grafiche sono equipaggiate con apposite ALI in modo da poter eseguire aritmetica a singola e doppia precisione, accesso in lettura e scrittura alla memoria centrale e gestione diretta delle memorie cache come la shared memory.

### Calcolo ibrido GPU+CPU

Progettare cluster per il calcolo ad alte prestazioni basato sulla struttura ibrida CPU+GPU è più complesso, ma permette di rendere molto efficiente in termini di tempo l'esecuzione di molte applicazioni di carattere scientifico, matematico, fisico o ingegneristico. Infatti la CPU consiste di pochi core ottimizzati per l'elaborazione sequenziale, mentre una GPU ha un'architettura massicciamente parallela che consiste di migliaia di core molto efficiente progettati per trattare task multipli simultaneamente. Alla CPU si lascia la parte di codice in cui vengono gestite le risorse hardware e software del sistema, mentre alla GPU si lascia la parte puramente matematica di analisi dati.

### Tensor Processing Unit - TPU

Google ha progettato e realizzato un processore proprietario denominato TPU e dedicato alle architetture delle reti neurali per l'apprendimento approfondito (Deep Learning Neural Networks). È stato chiamato così perché sfrutta la libreria TensorFlow sviluppata da Google per DeepLearning.

La TPU è un ASIC (Application Specific Integrated Circuit), ovvero un processore creato per svolgere specifici compiti in maniera estremamente ottimizzata. I processori TPU, progettati per l'apprendimento automatico, offrono prestazioni migliori per watt e numero di transistor rispetto alle CPU e GPU grazie all'alta specificità, a una tolleranza maggiore agli errori computazionali e a circuiti più semplici.

### Field Programmable Gate Array - FPGA

I FPGA sono array di componenti hardware programmabili singolarmente detti Configurable Logic Blocks (CLB) interconnessi tra di loro. In questo modo si può ottenere implementazioni hardware di algoritmi e reti logiche aggiornabili senza dover però sostenere gli elevati costi di fabbricazione di un chip. Grazie all'implementazione hardware hanno un elevato speedup.

## 3 Concorrenza

### 3.1 Processi e Thread

Un processo è un’istanza attiva di un programma che comprende tutte le informazioni necessarie per il suo funzionamento, risiede quindi in RAM. Il programma è un insieme di codice e strutture eseguibile da un elaboratore ed è passivo. Un programma può avere più istanze di esecuzione, ovvero più processi.

#### Contesto di un processo o Process Control Block - PCB

Un processo è composto dal codice con le istruzioni da eseguire e dal contesto di esecuzione (o Process Control Block - PCB). Il PCB è una struttura dati contenente tutte le informazioni per il sistema operativo per gestire un processo durante il suo ciclo di vita, è salvato in RAM ed è composto da:

- identificatore (PID)
- stato del processo
- contesto della CPU (registri e program counter)
- informazioni sulla memoria
  - base e limite della memoria dedicata al processo
  - tabella delle pagine o dei segmenti
  - heap (memoria dinamica) e stack (variabili locali e RDA)
  - initialized e uninitialized data segment
- informazioni sulle risorse
- informazioni di scheduling
- informazioni di comunicazione tra processi

#### Multiprogrammazione e ruolo del SO nella gestione dei processi

La multiprogrammazione consiste nell’avere contemporaneamente più job (o processi) salvati in RAM in modo da poterli alternare tra loro nel caso in cui uno di essi sia in “pausa” mentre attende una risorsa. Il SO si deve occupare di:

1. creazione e cancellazione di processi (utente e di sistema)
2. sospensione e riattivazione di processi (attraverso scheduling della CPU e swapping)
3. fornire meccanismi per la sincronizzazione di processi, la comunicazione fra processi e la gestione del deadlock (evitare che processi si blocchino in attesa di risorse)

#### Stati di un processo

- **new**: quando un processo è stato appena avviato
- **ready**: quando ha i dati, ma è in coda di esecuzione
- **running**: quando è in esecuzione
- **waiting**: quando sta aspettando le risorse
- **terminated**: quando ha terminato l’esecuzione

Il passaggio di un processo dallo stato di ready a quello di running è gestito dall’algoritmo di scheduler dispatch. Quando un processo passa da running a ready, il PCB di tale processo viene salvato in RAM per poter essere ripristinato la volta successiva che torna in esecuzione. Lo stato terminated si verifica per terminazione normale e restituzione del controllo al SO, terminazione anomala (es. eccezioni, ...), uso scorretto di risorse o chiamata a istruzioni non valide (trap).

## Scheduling dei processi pt. 1

Lo scheduling dei processi è un meccanismo messo in atto per massimizzare l'utilizzo della CPU, che consiste nel passare rapidamente dall'esecuzione di un processo al successivo, garantendo il time sharing. È svolto dal CPU scheduler (modulo del sistema operativo) che sfrutta tre code:

1. **job queue**: insieme di tutti i processi presenti nel sistema
2. **ready queue**: insieme dei processi ready, prossimi all'esecuzione, che risiedono in memoria centrale
3. **code dei dispositivi**: insieme dei processi in attesa di un dispositivo di I/O

## Thread

Un thread (filo) è l'unità base di esecuzione all'interno della CPU. Ogni processo deve essere suddiviso in uno o più thread per essere eseguito. Per gestirne l'esecuzione ogni thread è composto da:

- **Thread Identifier - TID**: identificativo di ogni thread analogo al PID
- **Thread Control Block - TCB**: dati di gestione del thread analogo al PCB
- **Thread Local Storage - TLS**: per memorizzare i dati del singolo thread (es. PC, stack, ...)
- riferimenti a codice, dati e risorse condivisi con altri thread dello stesso processo

I thread vengono utilizzati nelle architetture multicore perché permettono il parallelismo e la concorrenza in quanto è possibile eseguire in parallelo più thread diversi assegnati a diverse unità di calcolo. La condivisione dei dati tra più thread comporta la necessità di gestire correttamente la memoria condivisa. L'utilizzo di un TLS consiste nell'avere una copia dei dati condivisi in modo da lavorare localmente e aggiornare i dati solo alla fine dell'esecuzione del thread.

Per la corretta gestione dei thread da parte del kernel esistono due diverse mappature:

- **one-to-one**: ogni thread user-mode ha un analogo thread in kernel-mode (es. Windows e Linux)
- **many-to-many**: più thread kernel-mode possono gestire i thread in user-mode (poco diffuso)
- **misto**: usa sia la gestione one-to-one che la gestione many-to-many

Nei SO, i processi e threads sono organizzati gerarchicamente in una struttura ad albero padre-figli. In Linux esiste il processo `systemd` con PID=1 che è genitore di tutti i task (processi e threads). In fase di creazione di un nuovo task è possibile scegliere i dati che il nuovo task condivide con il padre (filesystem info, memory space, signal, files,...). Un processo figlio che non ha più il padre (perché ha terminato l'esecuzione) è detto zombie-defunct o orfano.

Linux impiega thread a livello kernel per la gestione dei dispositivi della memoria e delle interrupt e il thread `kthreadd` con pid 2 è genitore di tutti i kernel thread.

## 3.2 Concorrenza vs Parallelismo

### Confronto

- La **concorrenza** è la capacità di gestire più task (finalizzati a risolvere uno stesso problema) in modo apparentemente simultaneo, anche se eseguiti su un solo core. I diversi task vengono interrotti e ripresi, uno alla volta, molto velocemente.
- Il **parallelismo** è l'esecuzione effettiva e simultanea di più compiti su più core o processori. Necessita quindi di architetture parallele (multicore, multithreading, multiprocessori ...) per essere eseguito.

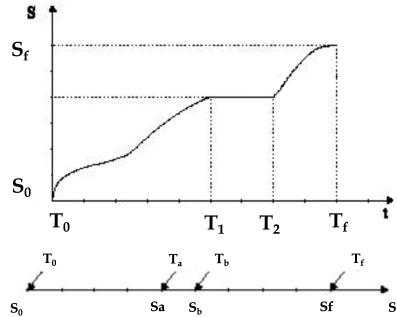
### Tipi di parallelismo

- **data parallelism**: la stessa operazione viene eseguita contemporaneamente sui dati, si partizionano i dati sui vari thread in modo che non ci siano conflitti e per accelerarne la computazione
- **task parallelism**: parallelizzare del codice tra thread diversi in modo da avere più processi che possono essere eseguiti se uno è in attesa di risorse

### 3.3 Diagrammi temporale

#### Diagramma dei tempi di esecuzione

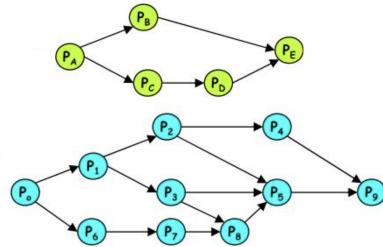
Il diagramma dei tempi di esecuzione permette di analizzare l'evoluzione di un processo in funzione del tempo. Siccome il tempo di esecuzione di un processo dipende da tantissimi fattori, tale grafico è sempre una approssimazione. Il plateau nel primo grafico indica che il programma è in attesa di risorse. Il grafico lineare, posto in basso, indica il tempo in cui verrà eseguito un processo e tornerà utile nello scheduling dei processi nella CPU.



#### Grafo di precedenza

Il grafo di precedenza serve per analizzare le dipendenze tra i vari moduli o subtask del processo, in modo da capire quali possono essere parallelizzate e l'ordine con cui devono essere eseguite. In questo modo si evita il problema delle approssimazioni sul tempo di esecuzione variabile.

Il tempo di esecuzione complessivo viene calcolato sommando i tempi dei processi in serie e sommando il massimo tra processi in parallelo.



#### Sistemi di processi

Un sistema di processi è un insieme di processi che si definisce:

- **closed system**: se esiste un processo iniziale e uno finale
- **open system**: se non esiste un processo iniziale o uno finale, ma è possibile trasformarlo in chiuso introducendo un processo iniziale e finale dummy

#### Massimo grado di parallelismo

Il massimo grado di parallelismo si intende la cardinalità del più grande sottoinsieme di processi tali per cui, presi due qualsiasi nodi del sottoinsieme, non esiste un percorso orientato che li connette. Ovvero sono il massimo numero di processi che verranno eseguiti in parallelo (che si trovano allineati verticalmente). In base all'architettura si può voler minimizzare il parallelismo per ottimizzare il calcolo in sistemi hardware limitati o massimizzare il parallelismo per massimizzare le prestazioni.

#### Interferenza e determinatezza

- un sistema di dice **determinato** se, dati due processi parallelizzati, la velocità e l'ordine di esecuzione non influenzano il risultato finale (non ci sono race conditions), altrimenti si dice **indeterminato**, un esempio di indeterminazione si verifica se entrambi i processi agiscono sulla stessa variabile senza appropriati controlli
- due processi si dicono **non interferenti** se uno è il successore dell'altro oppure se non si intersecano i codomini e nemmeno i domini con i codomini (ovvero se non vengono eseguiti in parallelo o se agiscono su variabili diverse)
- un sistema è determinato se e solo se è costituito da processi non interferenti
- due sistemi sono equivalenti se sono determinati, sono costituiti dallo stesso insieme di processi e se dallo stesso input producono lo stesso output

### 3.4 Risorse

La risorsa è un'astrazione per rappresentare una entità necessaria ad un processo per svolgere il proprio lavoro. Può essere sia output di altro processo (fondamentale per la continuazione del lavoro del processo in esame) che spazio di memoria. Se la risorsa non è disponibile, il processo ovviamente dovrà attendere per utilizzarla. Dividiamo le risorse:

- **condivisibili**: possono essere usate da più processi in parallelo (es. RAM condivisa)
- **non condivisibili**: non possono essere usate nello stesso tempo (es. stampante)
- **non consumabile**: può essere riutilizzata (es. core di una CPU può essere riutilizzato)
- **consumabile**: non riutilizzabile, in tal caso si parla di risorse esterne al sistema, (es. robot che può ricaricarsi da power pack separato dalla rete)

#### Prelazione o preemption

Una risorsa si dice preemptive (o con prelazione) se è possibile sottrarla forzatamente ad un processo in uso per assegnarla ad un altro processo. Ad esempio se si sottrae il core CPU ad un processo con bassa priorità nel caso in cui ce ne sia un altro con maggiore priorità.

#### Grafo di assegnazione delle risorse o grafo di Holt

Il grafo di assegnazione delle risorse è un grafo orientato composto da:

- $T = \{T_1, T_2, \dots\}$  insieme dei thread attivi nel sistema, rappresentato come nodi circolari etichettati con il rispettivo nome
- $R = \{R_1, R_2, \dots\}$  insieme delle risorse presenti nel sistema con le relative istanze  $W_i$ , rappresentate da rettangoli detti magazzini, con tanti pallini neri quante sono le istanze
- $T_i \rightarrow R_j$  arco di richiesta orientato indicante che il thread  $T_i$  ha richiesto la risorsa  $R_j$
- $R_j \rightarrow T_i$  arco di assegnazione orientato indicante che la risorsa  $R_j$  è stata assegnata al thread  $T_i$
- $T_i \dashrightarrow R_i$  arco di reclamo orientato indicante che  $T_i$  potrebbe richiedere  $R_i$  in futuro

Una risorsa  $R_i$  può avere tanti archi uscenti (di assegnazione) quante sono le istante  $W_i$ .

#### Costruzione del grafo delle risorse

Inizialmente si segnano tutte le risorse e tutti i thread e si aggiungono eventuali gli archi di reclamo. Ogni volta che un thread richiede una risorsa, si converte l'arco di reclamo in arco di richiesta. Se la risorsa è disponibile, l'arco di richiesta cambia verso e diventa arco di assegnazione. Se la risorsa è già occupata, il richiedente rimane in attesa ed eventualmente viene messo in una coda FIFO di attesa. Quando una risorsa viene liberata, l'arco di assegnazione torna ad essere un arco di reclamo.

## 3.5 Deadlock

### Definizione di deadlock

Un deadlock o stallo indica la situazione per cui si ha un insieme di thread bloccati in quanto ciascuno possiede almeno una risorsa ed attende di acquisirne altre, già allocate da altri thread. Gli stalli possono verificarsi spesso in quanto più thread possono competere per un numero limitato di risorse. Le situazioni (necessarie ma non sufficienti) per cui si può verificare sono:

- mutua esclusione delle risorse: una risorsa è utilizzabile solo da un processo contemporaneamente
- allocazione parziale: i processi allocano risorse in tempi diversi
- risorse senza prelazione: solo il processo che la ha acquisita può rilasciarla
- attesa circolare: ogni processo aspetta una risorsa che non verrà rilasciata (problema dei 5 filosofi)

### Definizione di livelock

Il livelock è una situazione di stallo attivo in cui i thread non sono effettivamente bloccati, ma non progrediscono. In genere si verifica quando un thread effettua ripetutamente un'azione che non ha successo, ad esempio quando due thread effettuano polling (busy waiting) per verificare lo stato dell'altro reciprocamente (livelock mutuo).

### Definizione di starvation

La starvation è una situazione in cui un processo o thread non riesce mai ad ottenere le risorse necessarie per essere eseguito, rimanendo in attesa (quindi attivo) per un tempo indefinito. Ad esempio quando un processo a bassa priorità non riesce mai ad essere eseguito data la costante presenza di processi più prioritari.

### Individuazione dei deadlock dal grafo delle risorse

Analizzando la presenza di cicli nel grafo delle risorse è possibile stabilire l'eventuale presenza di deadlock:

- se il grafo non contiene cicli, allora non si hanno deadlock
- se il grafo contiene cicli e se le risorse coinvolte hanno una singola istanza, allora si ha deadlock
- se il grafo contiene cicli e alcune risorse coinvolte hanno più istanze, potrebbe non esserci deadlock

### Algoritmo di individuazione del blocco critico - riduzione del grafo di Holt

In generale un sistema non è in stallo se e solo se il grafo di Holt è completamente riducibile. Un grafo di Holt è completamente riducibile se esiste una sequenza di passi di riduzione che elimina tutti gli archi del grafo. Una riduzione è possibile se un processo può acquisire tutte le risorse che necessita e di conseguenza potrà terminare. Se tutti i processi terminano allora il sistema non sarà in stallo. Per ridurre il grafo di Holt si ricorre al seguente metodo:

1. si scrivono le matrici delle risorse dichiarate  $A$ , delle risorse allocate  $B$  e delle risorse richieste  $C$ , in cui si hanno come colonne le risorse  $R_1, R_2, \dots$  e come prima riga i dati cumulativi del sistema  $S$  e come successive i vari processi  $P_1, P_2, \dots$  tali per cui  $A(i, j) = B(i, j) + C(i, j)$
2. si analizza solo la tabella  $C$  e se una certa riga  $P_i$  è minore di  $S$ , ovvero se le richieste di  $P_i$  possono essere soddisfatte, allora si cancella  $P_i$  e si incrementa la riga  $S$  con le risorse che vengono liberate
3. si procede con la cancellazione finché possibile e, se si riescano a cancellare tutte le righe allora il sistema non è in deadlock, altrimenti se alcune righe non sono eliminabili si verificherà un deadlock

## 3.6 Prevenzione del deadlock

### Allocazione globale

L'allocazione globale consiste nel far sì che un processo richieda tutte le risorse di cui ha bisogno in un'unica richiesta, prima di iniziare l'esecuzione. Se tutte le risorse non sono disponibili, il processo non parte, evitando così situazioni di attesa circolare.

### Allocazione gerarchica

L'allocazione gerarchica si basa sul fatto che tutte le risorse del sistema sono ordinate gerarchicamente, e ogni processo può richiedere risorse solo seguendo quell'ordine. Ovvero un processo può richiedere solo risorse con numero più alto di quelle che già possiede. In questo modo non possono verificarsi stalli dovuti al mutuo blocco di risorse (es.  $P_1$  ha  $R_1$  e chiede  $R_2$ , quando  $P_2$  ha  $R_2$  e chiede  $R_1$ )

### Algoritmo del banchiere

Sfrutta un principio simile al metodo di riduzione del grafo di Holt, quando un processo richiede risorse:

1. verifica che le risorse richieste rientrano in quelle dichiarate inizialmente dal processo
2. verifica che le risorse richieste siano inferiori di quelle disponibili
3. simula l'assegnazione di risorse decrementando quelle disponibili e se ci si trova ancora in uno stato sicuro (ovvero se il grafo è riducibile)
5. se sì, concede le risorse, altrimenti nega temporaneamente la richiesta

### Ripristino dei deadlock

Le operazioni per ripristinare uno stato attivo del sistema si chiama recovery e può prevedere:

- un rollback, ovvero alcuni processi rilasciano le risorse acquisite per permettere agli altri di proseguire (usato specialmente nei database come MySQL dove è possibile il rollback)
- la kill dei thread bloccati (secondo una certa priorità) ed eventualmente tutti i sottothread che dipendono da essi con possibilità di dover riavviare l'intero sistema, metodo chiamato algoritmo dello struzzo, usato dalla maggior parte dei sistemi operativi

### Gestione dei deadlock

In genere l'algoritmo del banchiere, sebbene sia ottimo, è molto dispendioso e non viene utilizzato nei sistemi attuali. Vengono invece impiegati sistemi di:

- prevenzione come allocazione gerarchica, alloc. globale, rilascio automatico di risorse (prelazione)
- rilevamento con la riduzione del grafo di Holt, eseguito con frequenza variabile in funzione di scelte progettuali quali la frequenza stimata di verifica di un deadlock, CPU in stato di sospensione, richieste non soddisfatte
- recovery attraverso rollback (se possibile) o applicazione l'algoritmo dello struzzo

## 3.7 Race condition e sezioni critiche

### Definizione di race condition

Una race condition indica la situazione in cui due processi paralleli e concorrenti modificano contemporaneamente una risorsa condivisa (es. variabili in memoria) e il risultato dipenderà dall'ordine con cui sono eseguiti.

### Istruzioni atomiche

Se più processi modificano le stesse variabili si possono verificare race condition date dal fatto che la modifica fatta da un programma può essere sovrascritta da quella dall'altro programma. Per evitare questa situazione si utilizzano le istruzioni atomiche ovvero istruzioni che vengono completate senza subire interruzioni che possono comportare a data race.

Alcune architetture implementano istruzioni atomiche via hardware ad esempio `TestAndSet` per controllare e modificare il contenuto di una parola in memoria o `CompareAndSwap` per scambiare due parole in memoria.

### Sezioni critiche

Se non è possibile completare la computazione utilizzando solo istruzioni atomiche, si utilizzano le sezioni critiche, ovvero si impongono dei vincoli a determinate regioni del codice affinché si abbia l'esclusività nel tempo della loro esecuzione. Le sezioni critiche hanno le seguenti proprietà:

- mutua esclusione: solo un processo per volta può accedervi
- progresso: la scelta del processo che accederà alla regione critica non può essere rimandata all'infinito
- attesa limitata: un processo che richiede l'accesso alla sezione critica non può attendere all'infinito

### Sezioni critiche e kernel con/senza prelazione

In un kernel senza prelazione, i thread del kernel non possono essere interrotti, per cui le race condition su dati del kernel vengono azzerate (su sistemi single core). Nel caso in cui più processi concorrenti contengano system call, tali system call verranno gestite individualmente e sequenzialmente.

In un kernel con prelazione, i thread del kernel possono essere interrotti da altri thread e, se non gestiti correttamente, si può incorrere in race condition. D'altra parte i kernel preemptive sono necessari in sistemi real-time (che devono rispondere a eventi entro limiti di tempo precisi e garantiti) o nei sistemi time-sharing (in cui si condividono le risorse tra più processi sfruttando il round robin e i quanti di tempo) siccome si diminuiscono i tempi di risposta.

Il kernel di Windows XP/2000 è (in parte) senza prelazione, ma i sottosistemi di Windows 2000 eseguiti in modalità utente hanno prelazione. Linux, da 2.6 introduce kernel con diritto di prelazione (sebbene sia personalizzabile attraverso una flag).

### Implementazione di sezioni critiche

- via hardware: nei sistemi non paralleli è sufficiente impedire che il codice nelle sezioni critiche venga interrotto ad esempio bloccando temporaneamente le interrupt
- via software: con i lock mutex, semafori e monitor implementati dal kernel

## 3.8 Lock Mutex

I lock mutex (MUTual EXclusion) sono strumenti software per controllare l'accesso alla regione critica. In entrata di una regione critica si chiama la funzione `acquire()` che aspetta in busy waiting finché non è possibile acquisire il flag, mentre in uscita della regione critica si effettua la `release` in cui si libera il flag acquisito in ingresso. Le due funzioni sono implementate via hardware, fornite dall'architettura, o sono atomiche per evitare data race.

```
acquire() {           |   release() {  
    while(!available); /* busy wait */ |       available = true;  
    available = false;      |   }  
}                      | }
```

## 3.9 Semafori

### Semafori contatore

I semafori contatore hanno un funzionamento simile ai lock mutex, solo che basandosi su un contatore, permettono ad un certo numero  $s \geq 1$  di processi di entrare in una determinata sezione critica. Per utilizzare un semaforo è necessario prima dichiararlo, specificandone il valore iniziale  $s$ . Le operazioni di `acquire` e `release` sono sostituite con `wait` (che attende che il contatore sia positivo prima di decrementarlo) e `signal` (che reincrementa il contatore), sempre gestite in maniera atomica.

```
semaphore mysem = 2; // dichiarazione di un semaforo con valore s = 2
-----
wait(s) {
    while(s <= 0); /* busy wait */
    s--;
}
-----| signal(s) {
        s++;
    }
}
```

I semafori sono particolarmente utili per gestire le risorse con più di una istanza (es. parcheggi disponibili e occupati in un parco macchine). Se il contatore può assumere solo i valori 0 e 1, i semafori associati sono detti semafori binari ed equivalgono a dei lock mutex.

### Semafori privati

I semafori privati sono classici semafori con la caratteristica che solo un processo (proprietario) è abilitato ad eseguirne la `wait`, mentre qualsiasi processo può eseguirne la `signal`.

### Come evitare busy waiting

Si osserva che, per come definito sopra, le istruzioni di `wait` e `signal` sfruttano il busy waiting, sprecando cicli della CPU. È possibile evitare ciò inserendo i processi in attesa in una coda, in modo da risvegliarli ogni volta che viene fatta una `signal`. In questo caso il contatore può assumere valori negativi e il relativo modulo è il numero di processi in coda. (negli esercizi si userà l'implementazione precedente)

```
typedef struct {
    int value;           // contatore
    struct pcb *list;   // lista con i PCB
} semaphore;
-----
wait(semaphore *s) {
    s->value--;
    if (s->value < 0) // se ho terminato le risorse
        block();        // aggiunge il processo a s->list e blocca l'esecuzione
}
-----
signal(semaphore *s) {
    s->value++;
    if (s->value <= 0) // se ci sono processi sospesi
        wakeup(P);      // risveglia il processo in s->list
}
```

### Deadlock e starvation

Il deadlock si verifica se un insieme di processi stanno aspettando un signal che solo uno di loro può fornire, ma che è bloccato. In genere si verifica quando due programmi acquisiscono due semafori in ordine invertito, per cui uno aspetta che l'altro rilasci il secondo semaforo e l'altro aspetta che il primo rilasci il primo semaforo.

La starvation si verifica se un processo in coda attende per un tempo indefinito in coda ad un semaforo senza venir mai svegliato. In genere si verifica se la lista dei processi in attesa è di tipo LIFO, al posto di FIFO, oppure se c'è un disaccoppiamento tra `text` e `text`.

### Implementazione dei semafori

- **OpenMP**: API multipiattaforma per la creazione di processi paralleli sia per normali computer che per supercomputer, le regioni critiche sfruttano la direttiva `#pragma omp critical { ... }`
- **Intel Threading Building Block - TBB**: libreria in C++ per sviluppare programmi paralleli

## 3.10 Monitor

### Condition variables

Le condition variables sono particolari oggetti in grado di interrompere l'esecuzione di un processo e di risvegliarlo non appena una determinata condizione è verificata. Di seguito un esempio in C++:

```
1 std::condition_variable my_condition_variable; // dichiarazione
2 my_condition_variable.wait(lock, []{ return condizione_logica; }); // wait
3 my_condition_variable.notify_one(); // notify_one, conosciuta come signal
4 my_condition_variable.notify_all(); // notify_one, conosciuta come broadcast
```

- **wait**: se la condizione è true, allora non fa nulla, altrimenti mette il processo in **wait** e libera il lock, quando riceve una **notify**, la condizione viene riverificata e se rimasta false, allora torna in attesa, altrimenti si riprende il lock e continua con il resto dell'esecuzione
- **notify\_one**: risveglia un processo in attesa, non forzandone la riattivazione in quanto spetta alla **wait** di verificare la condizione
- **notify\_all**: risveglia tutti i processi in attesa (analogo alla **notify\_one**)

### Monitor e condition variables

I monitor sono strumenti alternativi ai semafori che semplificano la gestione della mutua esclusione sacrificando in parte la parallelizzazione. Grazie all'utilizzo dei mutex, solo un processo per volta può accedere alle loro procedure. La differenza principale rispetto ai semafori è che non vengono effettuati nessun incremento o decremento di contatori, ma vengono utilizzate le condition variable per mettere in attesa e risvegliare i processi. Di seguito un esempio di implementazione di un buffer utilizzando i monitor in C++:

```
1 class MonitorBuffer {
2 private:
3     std::queue<int> buffer;    // buffer
4     const size_t maxSize = 5; // buffer size
5     std::mutex mtx;           // mutex per garantire la mutua esclusività
6     std::condition_variable notEmpty; // condition variable 1
7     std::condition_variable notFull; // condition variable 1
8
9 public:
10    void produce(int item) {
11        std::unique_lock<std::mutex> lock(mtx); // acquisisce il mutex
12
13        // aspetta finché non si hanno posti liberi nel buffer
14        notFull.wait(lock, [this]() { return buffer.size() < maxSize; });
15
16        buffer.push(item); // inserisce il prodotto
17        notEmpty.notify_one(); // sveglia un consumatore
18        // mutex rilasciato in automatico uscendo dallo scope grazie a unique_lock
19    }
20
21    int consume() {
22        std::unique_lock<std::mutex> lock(mtx); // acquisisce il mutex
23
24        // aspetta finché non si hanno prodotti nel buffer
25        notEmpty.wait(lock, [this]() { return !buffer.empty(); });
26
27        int item = buffer.front(); buffer.pop(); // rimuove il prodotto
28        notFull.notify_one(); // sveglia un produttore
29        return item; // restituisce il prodotto
30        // mutex rilasciato in automatico uscendo dallo scope grazie a unique_lock
31    }
32};
```

Si osserva che i processi risvegliati dai notity non vengono subito attivati perché devono prima aspettare che i mutex vengano liberati con l'uscita dalle procedure del monitor. Vengono sì svegliati, ma passano all'esecuzione solo dopo che il programma che li ha chiamati libera il mutex terminando.

## Problema dei 5 filosofi

Nel problema dei 5 filosofi si ha in una tavola rotonda in cui sono seduti attorno 5 filosofi intervallati da una forchetta per un totale di 5 forchette. I filosofi possono pensare o mangiare, per pensare non è richiesta nessuna risorsa, per mangiare devono aver acquisito entrambe le forchette che hanno accanto. Di seguito un esempio di codice, si nota che non è possibile che si verifichi deadlock perché acquisiscono entrambe le forchette insieme e non c'è l'*acquire-one-and-wait-the-other*. È comunque possibile che si verifichi starvation se un filosofo non riesce mai a mangiare.

```
1 #include <iostream>, <thread>, <mutex>, <condition_variable>, <array>
2
3 enum State { THINKING, HUNGRY, EATING }; // stati dei filosofi
4
5 class DiningPhilosophers { // e' monitor
6 private:
7     std::mutex mtx;                                // mutex per accesso monitor
8     std::array<State, 5> state;                    // stato dei filosofi
9     std::array<std::condition_variable, 5> cond;    // condition variables
10
11    // funzioni di utilita'
12    int left(int i) { return (i + 4) % 5; }
13    int right(int i) { return (i + 1) % 5; }
14
15    // se i filosofi a destra e a sinistra non stanno mangiando, cambia stato al
16    // filosofo in centro ed effettua una signal su tale filosofo, altrimenti non
17    // fa nulla e il thread rimane a dormire
18    void test(int i) {
19        if (state[i] == HUNGRY && state[left(i)] != EATING && state[right(i)] != EATING){
20            state[i] = EATING; cond[i].notify_one();
21        }
22    }
23
24 public:
25     DiningPhilosophers() { state.fill(THINKING); } // inizializzazione con costruttore
26
27     void pickup(int i) {
28         std::unique_lock<std::mutex> lock(mtx); // mutex bloccato
29         state[i] = HUNGRY; // imposta filosofo su affamato
30         test(i); // verifica disponibilita' immediata delle risorse, se libere le prende
31         cond.wait(lock, [this, i] { return state[i] == EATING;}); // attesa di risorse
32     }
33
34     void release(int i) {
35         std::unique_lock<std::mutex> lock(mtx); // mutex bloccato
36         state[i] = THINKING; // cambia stato del filosofo sazio
37         test(left(i)); // signal su quello di sinistra
38         test(right(i)); // signal su quello di destra
39     }
40 }
41
42 // funzione che esegue ciascun filosofo per pensare, prendere, mangiare e liberare
43 void philosopher(int i, DiningPhilosophers &D) {
44     auto think = [&]() { std::cout << "Philosopher " << i << " thinking\n"; };
45     auto eat = [&]() { std::cout << "Philosopher " << i << " eating\n"; };
46
47     for (int iter = 0; iter < 10; iter++) { // ciclo effettivo
48         think(); D.pickup(i); eat(); D.release(i);
49     }
50 }
51
52 // main che fa partire i thread dei vari filosofi
53 int main() {
54     DiningPhilosophers D;           // monitor dei 5 filosofi
55     std::vector<std::thread> threads; // thread per ogni filosofo
56     for (int i = 0; i < 5; ++i)      // lancia i thread
57         threads.emplace_back(phiosopher, i, std::ref(D));
58     for (auto &t : threads)          // aspetta la fine di tutti i thread
59         t.join();
60     return 0;
61 }
```

## Allocazione risorse con tempo massimo di utilizzo

Si vuole creare un monitor che gestisca l'assegnazione di risorse a singola istanza tra più processi tenendo conto di un tempo massimo di utilizzo. Di seguito un esempio in pseudocodice dalle slides:

```
1 class ResourceAllocator {
2     condition_variable x; // condition variable
3     boolean busy; // flag per capire se risorsa e' gia' utilizzata
4
5     void acquire(int time) { // acquisisce la risorsa
6         if (busy) x.wait(time) // aspetta per un tempo pari a time
7         busy = TRUE; // la acquisisce indipendentemente dal
8     }
9
10    void release() { // libera la risorsa
11        busy = FALSE;
12        x.signal();
13    }
14
15    void initialisation_code() { // codice di inizializzazione del monitor
16        busy = FALSE;
17    }
18 }
```

Si osservano alcune criticità:

- sono gli utilizzatori che devono preoccuparsi di chiamare release dopo il tempo massimo impostato, altrimenti la risorsa verrà assegnata contemporaneamente a qualcun altro
- il codice non gestisce eventuali false wakeup della condition variable

## Altri esempi visti nelle slides

Mailbox e parcheggio macchine ...

## 3.11 Parallelismo e concorrenza in Linux

Prima di Linux 2.6, il kernel non aveva prelazione per cui per creare delle sezioni critiche venivano disabilitati gli interrupt per un breve periodo di tempo. Poi è stata introdotta la prelazione in kernel mode e nella API POSIX sono stati definiti alcuni strumenti per la gestione della concorrenza (interi atomici, spinlock, mutex, semafori, condition variables e rilevamento deadlock).

### Interi atomici

Strumenti per agire con operazioni atomiche su interi (ad esempio sui contatori)

- `atomic_t counter` dichiarazione della variabile intera atomica `counter`
- `atomic_set(&counter,5) ↔ counter = 5;`
- `atomic_add(10, &counter) ↔ counter += 10;`
- `atomic_sub(4, &counter) ↔ counter -= 5;`
- `atomic_inc(&counter) ↔ counter++;`
- `atomic_read(&counter)` lettura atomica della variabile intera atomica `counter`

### Lock mutex

Consiste in una variabile booleana che viene gestita attraverso le chiamate a sistema `mutex_lock()` e `mutex_unlock()`. In C/C++ i mutex sono gestiti con:

- `pthread_mutex_t mtx`: dichiarazione del mutex `mtx`
- `pthread_mutex_init(&mtx,NULL)`: inizializzazione del mutex
- `pthread_mutex_lock(&mtx)`: esegue il lock (eventualmente attendendo) del mutex `mtx`
- `pthread_mutex_unlock(&mtx)`: esegue l'unlock del mutex `mtx`

## Spinlock

Gli spinlock sono mutex che effettuano busy waiting mentre sono in attesa di catturare il lock. Sono preferiti ai mutex quando la sezione critica è estremamente breve, oppure in presenza di sistemi che devono gestire I/O (driver dei dispositivi, o Interrupt Service Routine (ISR)). Nei sistemi single core sono estremamente sconsigliati perché rubano cicli di clock alla funzione che attendono che finisca.

### Semafori in user mode

Esistono due tipi di semafori in Linux: i semafori con nome (utilizzabili anche da processi non legati tra loro) e semafori senza nome (condivisi solo tra processi imparentati).

- `sem_t sem`: dichiarazione del semaforo senza normalmente `sem`
- `sem_init(&sem, int pshared, int value)`: inizializzazione del semaforo
- `sem_wait(&sem)`: esegue il wait sul semaforo `sem`
- `sem_post(&sem)`: esegue il signal del semaforo `sem`
- `sem_destroy(&sem)`: elimina il semaforo `sem` liberando le risorse
- `sem_t *sem = sem_open("/mysem", O_CREAT, 0644, 1);` creazione di un semaforo con nome che avrà le sue operazioni che noi non approfondiamo

Esistono dei semafori utilizzabili nel kernel, solo che ad oggi si preferiscono i mutex o gli spinlock

### Condition variables

Le condition variables sono associate ai lock mutex:

- `pthread_cond_t cond_var`: dichiarazione della condition variable `cond_var`
- `pthread_cond_init(&cond_var, NULL)`: inizializzazione della condition variable
- `pthread_cond_wait(&cond_var, &mtx)`: esegue il wait di `cond_var` rilasciando il mutex `mtx`
- `pthread_cond_signal(&cond_var)`: esegue la signal della `cond_var`

Di solito si esegue la `pthread_cond_wait(&cond_var, &mtx)` all'interno di un ciclo `while` con la condizione da verificare per il risveglio in quanto la condizione potrebbe non essere verificata oppure potrebbero verificarsi false wakeup.

### Deadlock detection

Per tracciare tutte le assegnazioni di risorse e rilevare i deadlock si utilizza `lockdep`. Siccome l'esecuzione del kernel con `lockdep` attivo è molto rallentata, si utilizza solo in fase di debug.

## 3.12 Memory Barrier

Le memory barrier sono strumenti che forzano l'esecuzione delle store e delle lock richieste dal programma, prima di eseguire le istruzioni successive. Serve quindi per forzare l'ordinamento delle letture e scritture in memoria. È possibile utilizzarle sia attraverso primitive hardware che primitive software. In genere si utilizzano in processi concorrenti lock-free con le ottimizzazioni del compilatore attive: in alcuni casi i compilatori invertono l'ordine delle load e store, con la MB è possibile forzare l'ordine stabilito per evitare data race.

## 4 Scheduling CPU

### 4.1 Introduzione

#### Alcuni concetti introduttivi

- **meccanismo di interruzione:** procedura per cui l'elaboratore interrompe l'esecuzione di un processo, ne salva in contesto in memoria (specialmente il PC per sapere dove ripartire) e carica il contesto di un altro processo in memoria avviandone l'esecuzione (meccanismo di interruzione è anche detto cambio di contesto)
- **istruzioni privilegiate:** insieme di istruzioni eseguibili solo dal sistema operativo per gestire alcuni compiti riservati al kernel (abilitare/disabilitare interruzioni, accedere ai registri di protezione della memoria, gestione delle eccezioni, arresto della CPU)
- richiamo sulla parte di processo, thread, preemptive e non preemptive, ... in particolare c'è da fare attenzione in caso di **scheduling preemptive su processi con dati condivisi** in quanto l'interruzione durante la gestione di dati condivisi può comportare risultati imprescindibili, specialmente in caso di modifiche dei dati di sistema da parte di un kernel preemptive

#### Kernel

Il kernel è la parte del sistema implementata in parte in hardware e in parte in linguaggi a basso livello (assembly, C), è composto da tre sezioni:

- **First Interrupt Handler - FLIH:** gestione iniziale di tutte le interruzioni (determinare l'origine dell'interruzione e attivare il segmento di codice opportuno)
- **dispatcher:** assegnazione dei core ai processi in base all'output dello scheduler che, attraverso l'algoritmo di scheduling, sceglie quali processi mandare in esecuzione
- **sincronizzazione:** gestione degli strumenti di sincronizzazione (semafori a contatore, con code FIFO, con stack LIFO, con liste di priorità, concorrenza nell'esecuzione di wait e signal, ... )

#### Scheduler

Algoritmo di alto livello che sceglie i processi da mandare in base alla priorità scelta, eventualmente la priorità può essere dinamica (dipendere dal tempo). L'ordinamento con priorità sfrutta la struttura di heap per migliore efficienza. Alcuni indici delle caratteristiche degli algoritmi di scheduling:

- **throughput:** produttività o numero di processi completati in un arco di tempo
- **CPU-burst:** ciclo di utilizzo puro della CPU di un processo
- **I/O-burst:** ciclo di attesa di un input/output di un processo
- **tempo di turnaround:** durata dell'esecuzione di un processo
- **tempo di attesa:** tempo di attesa nella ready queue
- **tempo di risposta:** tempo tra richiesta e risposta
- **latenza di dispatch:** tempo necessario al dispatcher per cambiare contesto

#### Grafico di Gantt

Il grafico di Gantt è un diagramma dei tempi di esecuzione in cui si segnano su una linea del tempo appositamente numerata, il susseguirsi dei vari processi in esecuzione sulla CPU. Vengono in particolar modo indicati il tempo di arrivo, di inizio e di fine di ogni processo in modo da poter facilitare il calcolo del tempo di turnaround, tempo medio di attesa e di risposta.

#### Scheduling dei sistemi real-time

I sistemi real time sono caratterizzati da processi invocati a intervalli periodici  $T_i$  con durata di esecuzione conosciuta  $P_i$ , tali per cui è necessario che un determinato processo termini le sue esecuzioni prima di essere reinvocato. In generale un sistema real-time si dice schedulabile se esiste almeno un modo per pianificare ottimamente i processi, ovvero se (gli algoritmi di scheduling avranno un upper bound leggermente inferiore):  $\sum_{i=1}^m T_i/P_i \leq 1$

I sistemi real-time si dividono in:

- **hard real time**: devono garantire il rispetto delle scadenze di esecuzione per i processi critici
- **soft real time**: richiedono che ai processi critici sia assegnata una maggiore priorità rispetto ai processi di routine (senza nessuna garanzia sui tempi di completamento)

### Gestione processi in iOS

Inizialmente era permesso di eseguire solo una applicazione in foreground e nulla in background. Da iOS 4 è stata introdotta la possibilità di eseguire un unico processo in foreground controllabile tramite GUI e altri servizi in background (senza accesso al display).

### Gestione processi in Android

Un'app, per funzionare il background deve utilizzare un servizio, ovvero un componente applicativo separato eseguito per conto di un'app. I servizi non hanno un'interfaccia utente e sono in parte forniti da sistema con la classe Services. I processi, inoltre sono classificati in: processo foreground, processo visibile (esegue task del processo in foreground), processo di servizio (svolge task in background ma evidente per l'utente), processo in background (svolge task non visibili all'utente), processo vuoto (non ha componenti attive associate ad app).

### Gestione processi in Chrome Browser

Il browser Chrome è un sistema multiprocesso con tre categorie di processi: browser (interfaccia utente, I/O dal disco e dalla rete), renderer (rendering delle pagine HTML, Javascript, immagini, uno per ogni scheda aperta, eseguiti in sandbox per ridurre problemi di sicurezza) e plug-in (estensioni per semplificare determinati compiti es. Flash, Windows Media).

## 4.2 Comunicazione tra processi

La comunicazione tra processi detta Inter Process Communication (IPC) serve per condividere informazioni e coordinare sottotask. I mezzi utilizzati sono la memoria condivisa (più rapida, ma con rischio di data race) o attraverso messaggi (più sicuro, ma più lento perché si sfruttano le system call). Nei sistemi multicore si sfruttano i messaggi in quanto la memoria condivisa può avere inconsistenze dovute alla suddivisione in gerarchie di cache.

### Ordinary pipes

Le pipes (in Windows dette anonymous pipes) sono vie di comunicazione unidirezionali tra processi padre e figlio. Il produttore scrive i dati nella write-end della pipe e l'utilizzatore legge i dati dalla read-end della pipe.

### Named pipes

Le named pipes chiamate anche FIFO sono pipes bidirezionali che sfruttano determinati tipi di files (FIFO) per la comunicazione tra più processi indipendenti (non serve la relazione padre-figlio e possono coinvolgere più di due processi).

### Mailboxes

Una mailbox (o porta) è una struttura per la comunicazione tra processi/thread concorrenti, basata sull'invio e ricezione asincrona di messaggi. Ogni porta ha un unico ID e i processi, per comunicare, devono condividere una stessa porta. Se più processi condividono la stessa mailbox, un messaggio potrà raggiungere solo di essi. I messaggi risiedono in code temporanee e si ha una disaccoppiamento tra mittente e destinatario. Alcune opzioni di gestione:

- **synchronous**: il mittente si blocca finché il destinatario non riceve un messaggio, il destinatario si blocca finché non riceve un messaggio
- **asynchronous**: senza interruzione tra sender e receiver, lo scambio avviene attraverso un buffer, sfruttando memoria dinamica condivisa o semafori (memoria statica)

## 4.3 Algoritmi di scheduling

### First Come First Served - FCFS

La CPU viene assegnata al processo che la richiede per primo. Si basa sull'utilizzo di una coda FIFO come ready queue. Può provocare l'effetto convoglio, aumentando parecchio il tempo medio di attesa dei vari processi da eseguire, se il primo che si impossessa della CPU la tiene occupata per tanto tempo.

### Shortest Job First - SJF

La CPU viene assegnata al processo più corto. Si sfrutta una coda con priorità in cui i processi con priorità maggiore sono quelli con tempo di esecuzione minore. Si necessita, quindi, di conoscere i tempi di esecuzione di ogni processo. Lo svantaggio è che si possa verificare starvation se un processo lungo rimane in coda siccome arrivano sempre processi più rapidi. Una possibile soluzione è implementare un algoritmo di aging per aumentare la priorità man mano che i processi rimangono in coda.

### Round Robin - RR

Vengono assegnati ad ogni processo dei quanti di tempo di esecuzione della CPU (10-100 ms) e i processi si turnano a vicenda. Dopo che un processo ha trascorso il suo quanto di esecuzione, è forzato a rilasciare la CPU (prelazione). La lunghezza del quanto di tempo permette di ottimizzare le prestazioni: per  $q$  grandi si ha comportamento simile a FCFS, per  $q$  bassi si rischia di perdere più tempo nel context switch che nell'effettiva esecuzione. In generale RR è più efficiente di SJF, nonostante abbia più context switch. Se in uno stesso istante un processo termina il quanto e un altro arriva nella coda di attesa è indifferente se il processo che prende l'esecuzione sia quello già in coda (FIFO) oppure quello appena arrivato (LIFO) in quanto nella realtà è altamente improbabile che ciò avvenga.

### Shortest Process Next - SPN

La CPU viene assegnata al processo con tempo di esecuzione minore, a parità di valore si usa la strategia FCFS. In una realizzazione non-preemptive si favoriscono i processi più brevi.

### Shortest Remaining Time - SRT

La CPU viene assegnata al processo con tempo di esecuzione rimanente inferiore. Problema fondamentale è determinare la durata dei processi, per effettuarla vengono eseguite stime predittive (algoritmi diversi basati sullo storico di processi simili). SRT equivale a SJF con prelazione.

### Highest Response Ratio Next - HRRN

Una volta definito il fattore di risposta come rapporto tra tempo di turnaround e tempo di esecuzione, si assegna la CPU il processo che in un certo istante ha il fattore di risposta maggiore. In questo modo si dà alta priorità ai processi brevi e si ha aging dei processi lunghi in attesa da tempo nella ready queue.

### Scheduling proporzionale alla frequenza - per sistemi real-time

Applicato ai sistemi real-time si sceglie di mandare in esecuzione quello che ha frequenza più elevata, interrompendo eventualmente altri processi (con prelazione). Il carico può essere gestito solo se vale:

$$\sum_{i=1}^m \frac{T_{\text{esecuzione},i}}{P_{\text{periodo},i}} \leq m_{\# \text{ processi}} \cdot (2^{1/m} - 1)$$

### Earliest Deadline First - EDF - per sistemi real time

Applicato ai sistemi real-time si sceglie di mandare in esecuzione quello che ha la scadenza più immediata, interrompendo eventualmente altri processi (con prelazione). Il caso ottimale si riesce a portare il consumo della CPU al 100%, ovvero l'upper bound è pari a 1.

## Scelta dell'algoritmo migliore

Per scegliere quale algoritmo implementare si può procedere in tre modi:

1. **modello deterministico**: se si è in grado (avviene raramente) di sapere a priori ogni processo eseguito dalla CPU con relativi tempi di arrivo e burst time, allora è possibile calcolare deterministicamente i tempi medi di risposta per ogni algoritmo e scegliere quello minore (e migliore)
2. **rete di code**: poco usato, consiste nel rappresentare il sistema come una rete di server in cui ogni componente (CPU, dispositivi I/O) è un server con una sua coda, analizzando le distribuzioni dei processi nelle varie code è possibile calcolare le performance di ogni dispositivo e scegliere l'algoritmo migliore (spesso usando distribuzioni matematiche e metodi matematici complessi e non reali); si può ricorrere eventualmente alla formula di Little:

$$n_{\text{processi in coda}} = \lambda_{\text{richieste in arrivo nel tempo}} \cdot W_{\text{tempo medio di attesa}}$$

3. **simulazione**: ovvero si confrontano i risultati dei diversi algoritmi in applicazioni simulate per scegliere quello migliore

## 4.4 Code multiple

È possibile creare più ready queue (code multiple) per gestire processi di ambiti diversi (foreground e background) in modo da poter scegliere un algoritmo migliore per ciascuna delle due categorie dei processi (RR per foreground e FCFS per background). Poi è necessario effettuare lo scheduling tra due code con ad esempio time slice a priorità fissa (80% foreground e 20% background).

Un processo viene assegnato ad una determinata coda in base a regole scelte dal sistema e può rimanere fisso su una coda o migrare tra una coda e l'altra. In questo ultimo caso (code multiple con feedback) è necessario fornire degli algoritmi che gestiscono la migrazione dei processi verso una coda con più priorità o meno priorità.

## 4.5 Scheduling multiprocessore

### Architetture omogenee

Si ipotizza che le unità di elaborazione siano omogenee (identiche), altrimenti diventa molto complesso ottimizzare la scelta di quale core assegnare ad un processo in base al tipo di processi e ai core disponibili (es. efficiency vs performance core).

### Multielaborazione asimmetrica vs simmetrica

- **multielaborazione asimmetrica** consiste nell'avere un processore detto master server che coordina gli altri processori, in questo modo solo il master accede i dati del sistema senza doverli condividere
- **multielaborazione simmetrica** (Symmetric Multi Processing - SMP) consiste nell'avere più core indipendenti, ciascuno con il proprio scheduler e i processi sono raggruppati in una coda comune o in code separate per ogni core; nel caso di coda comune serve fare attenzione a possibili race condition sull'accesso della coda; è usato da Windows, Linux e macOS in quanto più veloce ed efficiente

### Predilezione del processore vs bilanciamento del carico

- **predilezione del processore** consiste nel mantenere un processo in esecuzione sempre sullo stesso processore, in modo da ottimizzare l'utilizzo della cache; in Linux attraverso la system call `sched_setaffinity` è possibile impostare **strong affinity** per evitare di far migrare il processo tra processori o **soft affinity** per cui sotto determinati carichi è possibile per un processo migrare in altri processori
- **bilanciamento del carico** consiste nel ripartire uniformemente il carico per ogni processore (nei sistemi con ready queue comune diventa automatico); si distingue in migrazione guidata (push migration) se un processo periodicamente bilancia il carico dei diversi processori o migrazione spontanea (pull migration) se un processore inattivo sottrae carico ad uno in sovraccarico; Linux le implementa entrambe

## Gestione della memoria

Si parla di stallo di memoria quando la CPU è in attesa che dei dati dalla memoria, spesso si perde fino al 50% del tempo. È bene scegliere lo scheduling adatto in base anche al tipo di memoria disponibile. In architetture di calcolo separate NUMA (Non Uniform Memory Access) in cui è possibile accedere alla memoria di altre CPU, ma a costo elevato, è bene ridurre la migrazione dei processi.

## 4.6 Nei sistemi operativi attuali

### Linux pre 2.5

Si utilizzano code multiple con feedback gestite con RR e ha prelazione legata alla priorità. La priorità è legata alla percentuale di utilizzo della CPU di un dato processo, più tempo utilizza la CPU, minore sarà la priorità. È presente aging in quanto processi mai eseguiti hanno priorità maggiore.

### Linux post 2.6.23

Per ottimizzare la gestione multicore si utilizza il Completely Fair Scheduler (CFS). I processi vengono divisi in due classi **real time** e **default** con priorità propria. Il tempo di utilizzo della CPU viene suddiviso in latenza obiettivo, ovvero in intervalli di tempo entro cui un qualsiasi processo viene eseguito. La latenza obiettivo si ripartisce in maniera proporzionale alla priorità dei processi (si usa % di tempo e non RR a quanto fisso). La priorità è inversamente proporzionale al tempo virtuale di utilizzo detto **vruntime**. Il tempo virtuale di utilizzo è un accumulatore del normale tempo di utilizzo combinato con un fattore di decadimento detto **nice**:

$$\text{vruntime} += \Delta t \cdot \frac{\text{nice}_0}{\text{nice}}$$

I vari processi sono salvati in un red-black tree ordinato in base al loro **vruntime** (**vruntime** minore in cima). Ogni volta che un processo interrompe la sua esecuzione si calcola l'incremento del **vruntime**, si aggiorna il red-black tree con tutti i processi e si assegna l'esecuzione al processo in cima alla lista. I fattori di **nice** variano da **nice<sub>-20</sub>** (di minima priorità) a **nice<sub>19</sub>** (di massima priorità) con **nice<sub>0</sub>** di priorità neutrale.

Lo scheduler CFS supporta il bilanciamento del carico, è compatibile con NUMA e cerca di ridurre al minimo la migrazione dei thread. Per ridurre le migrazioni si dividono i core in cluster e si cerca di mantenere i processi all'interno dei cluster.

Con l'avvento delle architetture ibride (Intel Alder Lake o Apple M) è stato necessario apportare delle modifiche al kernel con Linux 5.16 in modo da scegliere meglio il tipo di core in base al tipo di processo.

### Windows 10

Win10 utilizza 32 code con priorità proporzionale al numero (da 0 a 31). L'esecuzione viene assegnata mediante RR ai processi appartenenti alla coda con priorità maggiore con prelazione a favore dei thread con maggiore priorità, ovvero un thread verrà eseguito solo se non ci sono altri thread con priorità maggiore. La priorità è gestita dinamicamente per evitare aging.

### Windows 11

Win10 non riconosce le diverse architetture dei core, per cui su un processore ibrido pecca di ottimizzazione. In Win11 è stata introdotta la gestione delle architetture ibride in modo da ottimizzare non solo le performance, ma anche l'efficienza del sistema. Il sistema di scheduling si basa sull'Intel Thread Director che permette di fornire al SO dati sull'esecuzione dei diversi thread e suggerisce (attraverso modelli di AI pre-learned) quale mandare in esecuzione. A differenza di Win10, in Win11 dispone di uno scheduler per i compiti grafici della GPU bypassando la CPU grazie anche al GPU Direct Storage.

## 5 Gestione memoria principale

### 5.1 Concetti base

- **librerie dinamiche** permettono al processo in esecuzione di caricare la libreria richiesta nel momento in cui serve, ed essendo condivisa, se il modulo è già in memoria (perché usato da altro processo) allora non deve essere caricato di nuovo
- **accesso alla memoria:** il processore accede in maniera immediata solo ai registri e alla memoria primaria (ed eventuali cache nel mezzo), per accedere alla memoria secondaria serve interfacciarsi con driver appositi in quanto gestito come dispositivo di I/O

#### Memoria del task manager di Windows

- **cache:** area di memoria con dati vecchi che possono essere usati per altri processi ma vengono mantenuti in memoria per un eventuale uso futuro
- **memoria compressa:** memoria non usata e compressa per occupare meno spazio
- **pool di paging o memoria di swap:** area di memoria salvata in un disco diverso dalla RAM in cui vengono salvati alcuni dati dei processi in modo da liberare la RAM quando risulta piena (es. computer con poca RAM o esecuzione di programmi che richiedono tanta memoria)
- **pool non di paging (Wired Memory):** dati del kernel memorizzati in RAM che non possono essere swappati in memoria secondaria

#### Frammentazione

- **frammentazione interna:** quando un programma alloca più memoria di quella che effettivamente usa, ovvero si hanno aree di memoria riservate a programmi, ma non usate
- **frammentazione esterna:** quando in una generica memoria sono presenti molte aree libere di piccole dimensioni intervallate da aree occupate, il problema è che le piccole aree non sempre sono utili per allocare un processo

### 5.2 Gestione della memoria e degli indirizzi dei programmi

#### Tipi di indirizzo

- **indirizzi simbolici:** sono i nomi delle variabili utilizzati nel codice del programmatore
- **indirizzi rilocabili:** generati dal compilatore e dal linker, sono relativi ovvero sono offset dall'inizio del segmento in cui si trovano
- **indirizzi logici:** generati dal loader e sono frutto della rilocazione degli indirizzi relativi una volta che il programma viene caricato in memoria
- **indirizzi fisici:** generati dalla MMU o dal TBL e corrispondono all'indirizzo effettivo della memoria in cui è contenuto il determinato dato

#### Gestione della memoria dedicata ai processi e rilocazione degli indirizzi

Quando un processo viene caricato in memoria, gli viene assegnato uno spazio di memoria personale delimitato dall'indirizzo base e dall'indirizzo limite.

Gli indirizzi relativi (nel codice) vengono rilocati in indirizzi logici attraverso un mapping lineare solo quando il programma viene caricato in memoria in quanto è necessario conoscere l'indirizzo base:

$$\text{ind. logico} = \text{ind. base} + \text{ind. relativo} \quad \text{con } \text{base} \leq \text{ind. logico} \leq \text{limite}$$

Se l'indirizzo logico non è compreso tra la base e il limite, si verifica un errore di segmentation fault.

## Tipo di binding

- **binding in compilazione:** i programmi contengono già indirizzi fisici (codice assoluto) e, vengono quindi eseguiti sempre nella stessa area di memoria (caricamento statico) (es. processi kernel)
- **binding in caricamento:** il passaggio da indirizzi rilocabili a indirizzi logici avviene in fase di caricamento, il codice è detto rilocabile e il caricamento è statico (non rilocalizzato in runtime)
- **binding in esecuzione:** il passaggio da indirizzi rilocabili a indirizzi logici avviene in fase di esecuzione, ovvero il codice può essere spostato nella RAM e gli indirizzi devono essere dinamicamente rilocabili, il codice deve essere rilocabile e il caricamento si dice dinamico

## 5.3 Paginazione

### Introduzione

La paginazione consiste nella suddivisione della memoria RAM in pagine fisiche e dello spazio logico (la memoria indirizzabile dal processore) in pagine logiche della stessa dimensione. La dimensione può variare tra 512 byte e 1GB, attualmente i sistemi hanno pagine comprese tra i 4KB e i 4MB. La corrispondenza tra le pagine logiche e quelle fisiche avviene tramite la tabella delle pagine (una per ogni processo). Le pagine fisiche sono dette anche frame.

### Passaggio da indirizzo logico a indirizzo fisico con paginazione

Dato un sistema con paginazione, è possibile suddividere un indirizzo (logico o fisico) in indice di pagina (logica o fisica) e offset di pagina. Ad esempio per indirizzi a 32 bit con pagine da 4KB si avranno 12 bit di offset e i restanti 20 di indice di pagina. Per passare da un indirizzo logico con pagina logica ad uno fisico con pagina fisica basta sostituire l'indice di pagina logica con il corrispettivo indice di pagina fisica. La traduzione viene fatta in hardware dalla Memory Management Unit (MMU).

### Tabella delle pagine

La tabella delle pagine (memorizzata in un apposito spazio in RAM) contiene la corrispondenza tra le pagine logiche e le pagine fisiche di uno specifico processo. È composta una riga per ogni pagina logica (creata dal sistema) e ogni riga contiene una cella con l'indirizzo di pagina logica, una cella adiacente con l'indirizzo corrispondente di pagina fisica ed eventuali metadati (bit validità, bit modifica, bit protezione). La MMU ci accede attraverso due registri:

- Page Table Base Register - PTBR: inizio della page table (CR3 in x86, TTBR in ARM)
- Page Table Length Register - PTLR: lunghezza della page table

### Translation Lookaside Buffer - TLB o cache per la page table

Il Translation Lookaside Buffer o TLB è una memoria associativa (CAM o Content Addressable Memory) che funge da cache per la tabella delle pagine in modo da facilitare il calcolo per il passaggio da pagina logica o pagina fisica. È implementato come memoria associativa in modo da avere il risultato in tempo costante  $O(1)$ , contiene tra le 64 e 1024 righe della tabella delle pagine e si trova vicino alla cache L2. Per velocizzare i processi, la ricerca della pagina avviene in contemporanea nel TLB e nella page table, inoltre si utilizzano TLB differenti per istruzioni e dati.

Siccome si ha una specifica tabella delle pagine per ogni processo, ad ogni context switch sarebbe necessario aggiornare il TLB. Per risparmiare ciò, si utilizza una colonna aggiuntiva chiamata Address Space Identifier (ASID) che memorizza l'ID del processo che usa la determinata corrispondenza logico-fisico. In questo modo prima di effettuare la traduzione è sufficiente controllare che il PID in esecuzione e il ASID corrispondano. Nel caso in cui ciò non avvenga, viene restituito un TLB-miss.

### Trade-off tra frammentazione e accesso veloce

Se le pagine hanno dimensioni piccole, si riesce a ridurre la frammentazione, ma aumenta la dimensione della tabella delle pagine e si possono avere importanti TLB-miss che rallentano l'esecuzione. Viceversa con pagine di grande dimensione si ha una tabella delle pagine di minore dimensione, accessi più veloci (pochi TLB-miss), ma si va incontro a grande frammentazione.

## Page swap

In caso di memoria piena, è necessario spostare alcune pagine dalla RAM allo spazio di swap nell'hard disk secondo un preciso algoritmo di swap. Le informazioni se una determinata pagina è in RAM o è stata swappata, sono contenute nei dati aggiuntivi della tabella delle pagine.

## Gestione dei page fault

Ad ogni processo vengono assegnate le pagine richieste per la sua esecuzione. Se un processo tenta di accedere a pagine non assegnateli o non ancora caricate in memoria, viene lanciata un'eccezione di page fault. Se la pagina può essere caricata si effettua il page load (la pagina viene caricata dal disco alla RAM e viene aggiornata la tabella delle pagine) e il page fault viene gestito silenziosamente. Se la pagina è presente, ma l'accesso non è consentito, viene lanciato un segmentation fault che provoca l'arresto del processo in esecuzione.

## Metadati della tabella delle pagine

- **bit di protezione:** specifica se l'accesso alle pagine è in sola lettura o lettura + scrittura
- **bit di validità:** specifica se la pagina si trova o meno in memoria, se non è in memoria (page fault) si rimedia con un page load (se possibile) o con un seg fault
- altri metadati non approfonditi, come riservatezza al kernel, bit di modifica, copy-on-write, ...

## Come ridurre la dimensione delle page table

Si vuole mantenere la tabella delle pagine contigua (per poter usare l'offset), ma allo stesso tempo si vuole evitare tabelle di grandi dimensioni per evitare frammentazione. Per risolvere questo problema possono utilizzare le tabelle invertite, le tabelle di pagine hash e le tabelle di pagina gerarchiche.

### Tabelle invertite

Le tabelle invertite consistono in tabelle di pagina ordinate in base alla pagina fisica. Non si potranno avere, quindi, più pagine logiche che mappano la stessa pagina fisica e soprattutto la ricerca della pagina logica non sarà più  $O(1)$ . Per questo sono poco usate.

### Tabelle di pagina gerarchiche

La tabella di pagina si sviluppa su più livelli preservando continuità delle sottotabelle e riducendo la frammentazione. La tabella di primo livello (e le altre sottotabelle tranne l'ultima) contengono riferimenti alle altre sottotabelle, mentre l'ultima tabella contiene l'effettivo indirizzo fisico. In base al numero di livelli, si suddivide l'indirizzo di pagina in più parti ( $p_1, p_2, \dots, d$ ) in cui ogni parte rappresenta l'offset della tabella ad un determinato livello ( $p_1$  per il primo,  $p_2$  per il secondo, ...,  $d$  per l'ultimo).

Negli attuali SO da 32-64bit si utilizzano da 3 a 7 livelli di paginazione, in hardware con RAM lenta (ci sono molti più accessi) o con basse capacità computazionali e pochi registri, il costo della paginazione gerarchica è parecchio elevato.

### Tabelle di pagina hash

Le tabelle di pagina hash sfruttano le funzioni di hash per indirizzare una determinata pagina logica alla riga con la corrispondente pagina fisica. Ogni riga della tabella sarà una catena (per gestire collisioni) di pagine fisiche. Il vantaggio principale è la riduzione del numero delle entry della tabella delle pagine.

### Tabelle invertite con hash

Sono tabelle invertite in cui la ricerca della riga avviene tramite funzione hash e, in caso di collisioni, si utilizzano sistemi di open addressing con varianti di funzioni hash  $H(p), H'(p), H''(p)$ .

### Clustered page table

In una stessa riga della tabella delle pagine si memorizzano le pagine fisiche associate a pagine logiche adiacenti, per condensare più righe insieme e risparmiare spazio.

## 5.4 Segmentazione

### Introduzione

Si sfrutta il fatto che ogni programma è suddiviso in blocchi o segmenti logici (data, bss), text, stack, ...) dette unità logiche. È possibile suddividere la memoria in segmenti di dimensione variabile, corrispondenti alle unità logiche dei programmi e, conoscendo l'indirizzo di partenza e la lunghezza di ogni segmento è possibile convertire un indirizzo logico (segmento logico + offset) in indirizzo fisico (segmento fisico + offset) facendo attenzione che l'offset non superi la lunghezza del segmento.

### Tabella dei segmenti

In analogia alla tabella delle pagine, si ha una tabella dei segmenti per ogni processo. La tabella dei segmenti è indicizzata in base al segmento logico e contiene il corrispettivo indirizzo base del segmento fisico e la sua lunghezza. Ci si accede attraverso i due registri:

- Segment Table Base Register - STBR: inizio della segment table
- Segment Table Length Register - STLR: lunghezza della segment table

### Vantaggi e svantaggi rispetto alla paginazione

Il vantaggio principale rispetto alla paginazione è la grande riduzione di frammentazione interna (non era sempre detto che i processi usassero interamente tutte le pagine), lo svantaggio è che la conversione tra indirizzo logico a fisico è leggermente più complessa (somma e non semplice sostituzione di una parte dell'indirizzo).

### Segmentazione paginata

Nei sistemi a segmentazione paginata si ha che ogni segmento è diviso in pagine con una tabella delle pagine per ogni segmento. Partendo da un indirizzo logico (segmento + offset) si ottiene un Linear Space Address (indice page table + indice pagina + offset) che attraverso la paginazione diventa un indirizzo fisico.

## 5.5 Paginazione e segmentazione nei processori Intel e ARM

### Segmentazione Intel 1° versione

Nei primi processori Intel a 32bit, si poteva usare sia segmentazione che misto di paginazione e segmentazione. La segmentazione prevedeva:

- dimensione massima di ogni segmento di 64K → 16 bit offset massimo
- 8K segmenti riservati per dati locali (Local Descriptor Table - LDT) → 13 bit segmento logico
- 8K segmenti condivisi per dati globali (Global Descriptor Table - GDT) → 13 bit segmento logico
- 1 bit per classificare il segmento riservato o condiviso e 2 bit di protezione

Per cui si avevano al massimo 16K segmenti da 64K con offset da 16 bit, con 1GB di spazio indirizzabile.

### Segmentazione Intel 2° versione

La seconda versione permetteva di gestire completamente i 4GB di RAM, sviluppata per i processori Pentium da 32bit, in cui si hanno 20 bit di offset e un extra bit di granularità che permetteva di estendere l'offset di altri 12 bit per avere al massimo 32 bit complessivi di offset e di conseguenza segmenti da 4GB.

### Paginazione Intel Pentium 1° versione

Si potevano avere due tipi di paginazione, con massimo spazio indirizzabile di 4GB:

- paginazione su singolo livello con pagine da 4MB (10 bit di indice e 22 bit di offset) e tabella delle pagine con righe della page table da 32 bit
- paginazione su due livelli con pagine da 4KB (10 bit di indice 1° livello, 10bit di indice 2° livello e 12 bit di offset) con righe della page table da 32 bit

### **Paginazione Intel Pentium con Page Address Extension PAE**

Per riuscire a mappare più di 4GB di memoria RAM senza passare alle architetture a 64 bit, Intel ha sviluppato il Page Address Extension (PAE) estendendo l'indirizzo fisico di 4 bit per un totale di indirizzi fisici a 36 bit e 64GB di memoria indirizzabile. Si sfrutta la paginazione a tre livelli con:

- 2 bit per selezionare la entry nella Page Directory Pointer Table
- 9 bit per selezionare la entry nella page directory di primo livello
- 9 bit per selezionare la entry nella page directory di secondo livello
- 12 bit di offset

Avendo le tabelle delle pagine a 64 bit, è possibile memorizzare indirizzi di pagine fisiche fino a 36 bit. Questo permette all'intero sistema di lavorare con RAM superiori a 4GB, ma il singolo processo può ancora indirizzare 4GB. In alternativa è possibile usare le pagine da 2MB con singolo livello di paginazione con offset di 12+9 bit e indice di pagina di 9 bit.

### **Paginazione dei sistemi a 64bit**

Nelle architetture a 64bit si è esteso di gran lunga lo spazio indirizzabile con la possibilità teoria di usare al massimo 64 bit di indirizzo. Nelle applicazioni reali, si usano indirizzi fisici da 52 bit e indirizzi logici da 48 bit con PAE. Si usa paginazione a 4 livelli con indirizzi con la seguente suddivisione dei bit:

16 unused - 9 page map level 4 - 9 page dir. pointer table - 9 page directory - 9 page table - 12 offset

### **Paginazione nei sistemi ARM a 3264bit**

Nei sistemi ARM a 32 bit si ha paginazione su due o uno livelli con pagine, dette sezioni, da 4KB e 1MB o 16KB e 16MB. Nei sistemi a 64bit, invece, si ha paginazione su a massimo 4 livelli con pagine di dimensione variabile in base alla granularità:

- granularità 4KB: pagine da 4KB, 1MB, 1GB, indirizzo da L0[9], L1[9], L2[9], L3[9], offset[12]
- granularità 16KB: pagine da 16KB, 32MB, 512MB, indirizzi da L1[13], L2[13], L3[8], offset[14]
- granularità 64KB: pagine da 64KB, 512MB/1GB, con 2 o 3 livelli in base alle configurazioni

## **5.6 Memoria virtuale e swap**

### **Scopo della memoria virtuale e swapping e paging**

La memoria virtuale indica l'insieme della memoria RAM e di una sezione riservata del disco utilizzata in caso di riempimento della RAM che dal punto di vista dell'indirizzo logico appare omogenea. Questo permette trattare uniformemente RAM e disco e di delegare il compito di gestione e ottimizzazione dei processi in memoria a dei servizi del SO detti pager (gestione dei page fault) o swapper (sposta pagine inutilizzate da RAM al disco per liberare RAM). Esistono due tipi di swapper:

- swapper pigro: muove solo alcune pagine dei processi
- swapper tradizionale: muove tutte le pagine associate ad un processo

In Linux si usava una partizione di swap ad hoc, recentemente si può usare anche un file apposito. In Windows esiste un file apposito e recentemente è stata aggiunta la possibilità di swappare anche sui dispositivi mobile (iPad Mx).

### **Gestione di un page fault**

Come detto prima, un page fault si verifica se si tenta di accedere ad una pagina con bit validità nullo (pagina assente in RAM). Si procede verificando se il programma può effettivamente accedere alla pagina richiesta (analizzando il PCB), in caso affermativo la pagina corrispondente viene caricata, in caso negativo viene lanciato un segmentation fault. Se la memoria risulta già piena e non è possibile caricare la pagina richiesta, bisogna prima applicare un algoritmo di swap per sostituire una pagina non usata della RAM con quella nuova richiesta.

## Effective Time Access

Il tempo di acceso effettivo alla RAM deve tenere conto della possibile necessità di effettuare operazioni di page load o di swap in e swap out. È dato dalla seguente formula:

$$EAT = (1 - p_{\text{page fault}}) \cdot T_{\text{memory access}} + p_{\text{page fault}} \cdot T_{\text{page fault + swap out + swap in}}$$

## Algoritmi di swapping

- **First In First Out - FIFO:** la prima pagina ad essere caricata in RAM sarà la prima ad essere swappata, non tiene conto del fatto che magari la prima pagina è quella più utilizzata, inoltre è possibile che si verifichi la anomalia di Belady (approfondita dopo)
- **Algoritmo ottimo - OPT:** rimuove la pagina che non verrà utilizzata per più tempo, non ha la anomalia di Belady, ma non può essere implementato perché prevede la conoscenza del futuro
- **Least Recently Used - LRU:** rimuove la pagina che per più tempo non è stata utilizzata, non soffre dell'anomalia di Belady e per essere implementata ci sono 3 possibilità:
  - salvando l'orario (contatore clock) nella page table
  - usando uno stack per cui una pagina utilizzata viene tolta dal mezzo e rimessa in cima
  - usando un bit che indica se una pagina è stata utilizzata o meno negli ultimi 100 ms
- **Second Chance:** si utilizzano due extra bit di accesso e modifica che vengono posti a 1 rispettivamente in caso di accesso o di modifica e periodicamente vengono resettati a 0, le prime pagine eliminate saranno quelle con (0,0), poi quelle solo lette (1,0), poi quelle solo scritte (0,1), infine quelle sia lette che scritte (1,1)
- **Least Frequently Used - LFU:** viene sostituita quella con contatore di utilizzo minimo (scarse prestazioni in costo computazionale e efficacia)
- **Most Frequently Used - MFU:** viene sostituita quella con contatore di utilizzo più alto perché quelle con contatore basso sono appena state caricate in memoria (scarse prestazioni)

## Anomalia di Belady

L'anomalia di Belady consiste nel fatto che analizzando il grafico dei page fault in funzione dei frame disponibili (grandezze inversamente proporzionali) si nota che per alcuni valori se si aumentano i frame allora aumentano controintuitivamente anche i page fault. Questo fenomeno si verifica nel caso dell'algoritmo FIFO se l'algoritmo perde memoria delle pagine accedute in precedenza in caso di loop di richieste per cui un mancato allineamento comporta un continuo swap out.

## Pool frame liberi

Quando una pagina deve essere swappata nel disco per lasciare posto ad un'altra pagina, si effettua contemporaneamente la copia della pagina in una parte della memoria riservata detta pool di frame liberi e nell'hard disk. In questo modo è possibile caricare subito la nuova pagina dal disco sulla RAM e se la pagina swappata è richiesta entro un breve lasso di tempo, si trova ancora nella memoria tra i pool di frame.

## Ottimizzazione delle scritture

Quando si rileva un momento di inattività del dispositivo, un processo di paging si avvia per copiare le pagine modificate dalla RAM al disco in modo da velocizzare un eventuale swap futuro essendoci più pagine già copiate sul disco.

## Pagine Pinned

Le pagine pinned sono quelle obbligate a stare in memoria in uno specifico posto, ad esempio quelle utilizzate per copiare files dai dispositivi di I/O nel memory mapping dei dispositivi di I/O.

## 5.7 Gestione dei frame tra i processi attivi e Working Set

### Allocazione dei frame

Quando dei processi devono allocare i loro frame in memoria possono avvenire vari tipi di allocazioni:

- alloc. uniforme: il numero di frame è ripartito in maniera equa tra i processi
- alloc. proporzionale: il numero di frame è ripartito proporzionalmente alla dimensione dei processi
- alloc. con priorità: il numero di frame è ripartito proporzionalmente alla priorità dei processi

### Sostituzione dei frame

Quando dei frame devono essere sostituiti si parla di:

- sostituzione globale: se i frame swappati sono di un qualsiasi processo
- sostituzione locale: se i frame swappati sono dello stesso processo che richiede quello nuovo

### Thrashing

Il thrashing è una situazione in cui si verificano tantissimi page fault e l'utilizzo della CPU crolla sensibilmente perché è in continua attesa di ricevere dati.

### Località e working set

Il working set è l'insieme di pagine che vengono accedute in un intervallo di tempo  $\Delta$  detto finestra di working set. Il working set viene utilizzato per identificare le pagine da non swappare: si aggiungono  $n$  bit di riferimento per ogni riga della page table, quando una pagina viene visitata si pone il bit di sinistra a 1 e ogni  $\Delta/n$  viene mandato un interrupt che esegue uno shift dei bit di riferimento. In questo modo se almeno uno dei bit di riferimento è 1, allora la pagina fa parte del WS, altrimenti si può swappare.

### Frequenza di page fault

È possibile adottare un algoritmo che, fissata una soglia massima e minima di page fault, gestisce i frame associati ad un processo in maniera adattiva. Se il numero di page fault supera la soglia massima, l'algoritmo acquisisce nuovi frame, mentre se va sotto la soglia minima, rilascia alcuni frame.

### Correlazione invocazione di una funzione e page fault rate

Si osserva che quando in un programma viene invocata una funzione, spesso si ha un aumento di page fault dovuto al fatto che è necessario caricare le pagine per eseguire quella determinata funzione.

### Prepaging

Il prepaging consiste nel caricare l'intero working set all'inizio della chiamata di una funzione o programma per limitare i page fault per cambio di working set. Il working set cambia di dimensione in base al sistema e alla memoria disponibile.

### Dimensione delle pagine e TLB reach

Il TLB Reach è la dimensione della memoria raggiungibile dal TLB ( $\#$  pagine nel TLB  $\times$  dim. pagine). Idealmente il working set deve essere completamente contenuto in un TLB per evitare page fault frequenti.

### Copy On Write - COW

Quando si crea un processo figlio con una fork, la copia delle pagine del padre per il figlio avviene solo quando uno dei due processi effettua una modifica a tale area di memoria.

## 5.8 Allocazione della memoria del kernel

### Introduzione - contestualizzazione

Le strutture dati del kernel devono essere continue per cui la memoria del kernel non è soggetta a paging (ovvero a swap) e utilizza solo indirizzi fisici. Viene in genere allocata secondo due strategie l'allocazione a potenze di due e l'allocazione a lastra.

### Allocazione a potenze di due

L'allocazione a potenze di due: si allocano sempre pagine a blocchi di potenze di 2 per ogni processo e quando alcune pagine vengono deallocate, il segmento viene spartito a metà rimanendo sempre una potenza di 2.

### Allocazione a lastra

Nell'allocazione a lastra la memoria del kernel è suddivisa in cache che raggruppano diverse slab (non necessariamente continue) che memorizzano una stessa struttura dati. Esiste una unica cache per ogni struttura (es. cache dei semafori, cache dei descrittori dei files, ...). Le slab sono aree di memoria costituite da pagine continue che sono suddivise in slot grandi esattamente quanto l'oggetto che devono contenere (es. slab da 12KB formata da 3 pagine di 4KB che contiene 24 slot per 24 oggetti da 512B). In questo modo si riduce la frammentazione per l'allocazione di piccoli oggetti.

Le cache hanno anche il compito di gestire gli oggetti nella memoria, in particolare gli oggetti vengono tutti inizialmente allocati e preinizializzati pronti all'uso. Poi, una volta utilizzati non vengono deallocated, ma semplicemente marcati come liberi. In questo modo di elimina l'overhead di allocazione e deallocazione continua di oggetti, rendendo il sistema più efficiente. Lo slab allocator si occupa di fare ciò.

Siccome non sempre le slab si riempiono (specialmente se le strutture dati occupano una dimensione strana es. 592byte), la parte di slab che rimane vuota è utilizzata dallo slab allocator per memorizzare dati sul riempimento delle slab, oppure si usa per lo slab coloring. Lo slab coloring consiste nello sfalsare l'indirizzo di partenza delle slab (es. slab0 con offset 0B, slab1 con offset 64B, slab2 con offset 128B) in modo che quando vengono mappate su cache set-associative, si distribuiscono meglio su più set diversi e si evita di avere continue sovrascritture e continui cache miss.

## 5.9 Mappatura della memoria e gestione dell'I/O

### Mappatura di un file - Memory Mapping

Quando un processo richiede l'accesso a files di grandi dimensioni, tale file viene caricato solo parzialmente in memoria ed è gestito tramite page fault come se fosse un processo. Alcuni sistemi operativi obbligano l'accesso ai files tramite memory mapping ad esempio `mmap` in Linux o tramite le API in Windows come la funzione `CreateFileMapping`.

### Mappatura degli I/O

Anche i registri dei dispositivi di I/O vengono mappati in memoria e vengono letti attraverso la memoria primaria, con paginazione.

## 5.10 Gestione della memoria in Linux

### Allocazione a lastre, buddy, SLOB e SLUB per memoria kernel

- linux pre 2.2: usa il sistema buddy con allocazione a potenze di 2 per strutture dati e altro
- linux post 2.2: usa la allocazione a lastre per strutture dati, per il resto usa quella a potenze di 2
- linux recenti: usano anche SLOB (Simple List of Block, per embedded, con tre liste per oggetti piccoli, medi e grandi) e SLUB (allocazione a lastre ottimizzata per il multiprocessore)

### Gestione della memoria utente

Ci sono due liste, una con le pagine attive e una con le pagine non attive. Quando una pagina viene visitata, si sposta in cima alla lista delle pagine attive, di conseguenza le pagine poco visitate si spostano in fondo alla lista. Il daemon `kswapd` tiene bilanciate le due liste (sposta pagine inattive nella lista delle pagine inattive). Le pagine da rimuovere sono scelte tra la lista delle pagine inattive.

## 5.11 Gestione della memoria in Windows

### Gestione della memoria kernel

La memoria kernel è divisa in:

- paged pool - paged kernel memory: memoria che può essere eventualmente spostata nel file di paging `pagefile.sys` e contiene i dati di cui non si richiede accesso immediato
- non paged pool - non paged kernel memory: memoria che deve rimanere in RAM e che non può essere paginata (es. Interrupt Service Routine (ISR))

### Gestione della memoria utente

Il sistema impiega paginazione su richiesta via clustering, ovvero in caso di page fault vengono caricate in memoria sia la pagina richiesta che quelle limitrofe per evitare eventuali page fault. All'avvio di un processo gli viene assegnata la dimensione minima e massima del working set (tra 50 e 345) che specifica il minimo numero e il massimo numero di pagine caricabili in memoria da un determinato processo. Quando la memoria libera scende sotto una determinata soglia, il sistema riduce i working set per liberare memoria. Windows inoltre utilizza la memoria compressa in sostituzione allo swap. Questo permette di liberare memoria senza effettuare swap, riducendo i tempi di trasferimento tra disco e RAM.

## **6 Gestione file**

## 7 Affidabilità

### 7.1 Nozioni introduttive

#### Affidabilità

L'affidabilità di un sistema (o dispositivo) è la probabilità che esso funzioni correttamente per un certo tempo sotto certe condizioni imposte. In altre parole è la probabilità che il sistema non abbia guasti di un certo tipo nello svolgimento di una certa missione. Per facilitare il calcolo dell'affidabilità di sistemi complessi si può utilizzare la scomposizione gerarchica di un sistema in sottosistemi.

#### Tipi di errori e guasti

Gli errori o guasti si distinguono in:

- permanenti: se una volta verificati, rimangono presenti nel sistema finché non vengono riparati
- temporanei: se non si verifica sempre in tutte le condizioni e si classifica in
  - transitori: se accade solo una volta
  - intermittenti: se si presenta più volte in modo imprevedibile

Secondo lo standard IEEE un guasto può essere sia un difetto di un componente, sia un errore in una delle procedure che deve compiere (es. dati, operazioni, interferenze, ...) che si manifestano durante l'esecuzione.

#### Fault tolerance

Per fault tolerance si fa riferimento all'abilità di un componente di continuare il normale funzionamento nonostante la presenza di guasti. Per avere fault tolerance è necessario inserire forme di ridondanza che avranno costo economico e prestazionale. In genere si vuole incrementare la fault tolerance in risorse delicate da proteggere.

Per aumentare la fault tolerant di un SO si possono implementare alcune soluzioni come:

- isolamento dei processi: se un processo si blocca non blocca l'intero sistema
- controllo della concorrenza: evitare deadlock, livelock o altri
- virtual machines: per astrarre hardware ed eseguire i processi in sandbox
- checkpoints e rollback: per ripristinare il sistema se si verifica un errore
- ridondanza spaziale/fisica: impiego di più componenti che svolgono lo stesso compito e possono sostituire i colleghi in caso di rottura
- ridondanza temporale: riesecuzione di una funzione in caso si verifichi un errore (transitorio)
- ridondanza di informazioni: clonazione delle informazioni in modo da poter individuare e correggere i bit di errore

#### Affidabilità logistica

L'affidabilità logistica è la probabilità che non si verifichi nessun guasto di qualsiasi tipo.

#### Affidabilità di missione

L'affidabilità di missione è la probabilità che non si verifichino guasti con conseguenze gravi tali da pregiudicare la funzionalità del sistema. Si possono distingue in guasti significativi (che alterano le funzionalità del sistema ma non gli impediscono di compiere la missione) e guasti maggiori (che impediscono il completamento della missione).

#### Sicurezza

La sicurezza è la probabilità che non si verifichino guasti con possibili conseguenze catastrofiche con danni a persone, cose o al sistema stesso.

## 7.2 Formule matematiche

### Affidabilità - Reliability

Rapporto tra i dispositivi funzionanti ad un tempo  $t$  e i dispositivi funzionanti al tempo iniziale  $t_0$

$$R_{\text{affidabilità}}(t) = \frac{n_{\text{disp. funzionanti}}(t)}{n_{\text{disp. funzionanti}}(t_0)}$$

La funzione ha le seguenti proprietà:  $\begin{cases} R(t) = 1 & \text{per } t = 0 \\ 1 \geq R(t) \geq R(t + \Delta t) & \text{per } \Delta t \geq 0 \\ R(t) = 0 & \text{per } t = +\infty \end{cases}$

La derivata è chiamata densità di affidabilità  $r(t) = \frac{dR(t)}{dt}$

### Probabilità di guasto - Unreliability

Probabilità che si verifichi un guasto, complementare alla affidabilità:

$$Q_{\text{guasto}}(t) = 1 - R(t) = 1 - \frac{n_{\text{disp. funzionanti}}(t)}{n_{\text{disp. funzionanti}}(t_0)}$$

La funzione ha le seguenti proprietà:  $\begin{cases} Q(t) = 0 & \text{per } t = 0 \\ 0 \leq Q(t) \leq Q(t + \Delta t) & \text{per } \Delta t \geq 0 \\ Q(t) = 1 & \text{per } t = +\infty \end{cases}$

La derivata è chiamata densità di probabilità di guasto  $q(t) = \frac{dQ(t)}{dt} = \frac{d(1 - R(t))}{dt} = \frac{-dR(t)}{dt} = -r(t)$

### Mean Time To Failure - MTTF

Media delle ore di funzionamento prima di un guasto

$$\text{MTTF} = \frac{\text{somma ore di funzionamento di tutti i dispositivi}}{\text{numero di dispositivi}} \quad P_{\text{failure}} = \frac{1}{\text{MTTF}}$$

### Mean Time To Repair - MTTR

Media delle ore richieste per riparare i dispositivi, tempo di esposizione per cui un altro guasto può provocare perdita di dati

$$\text{MTTR} = \frac{\text{somma ore di riparazione di tutti i dispositivi}}{\text{numero di dispositivi}} \quad P_{\text{repair}} = \text{MTTR} \cdot P_{\text{failure}} \approx \frac{\text{MTTR}}{\text{MTBF}}$$

### Availability

$$\text{Frazione di tempo di funzionamento del dispositivo:} \quad A_{\text{availability}} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} = \frac{\text{MTTF}}{\text{MTBF}}$$

### Mean Time Between Failure - MTBF

Media delle ore di funzionamento tra un guasto e quello successivo, corrisponde alla somma tra MTTF e MTTR, siccome molte volte il MTTF  $\gg$  MTTR si utilizzano MTTF = MTBF.

### Mean Time to Data Loss - MTTDL

Media in ore per avere una perdita di dati (es. se si verifica un guasto al componente A mentre l'altro componente B è in riparazione):

$$\text{MTTDL} = \frac{1}{P_{DL}(t)} \approx \frac{1}{P_F(A) \times P_R(B) + P_F(B) \times P_R(A)}$$