

SISTEMI OPERATIVI

Prof. Natella Roberto – A.A. 2024/25

Luca Maria Incarnato

INDICE DEGLI ARGOMENTI

GESTIONE E SCHEDULING DEI PROCESSI E THREADS

1. STRUTTURA ED EVOLUZIONE DI UN SISTEMA OPERATIVO (p. 3)
2. RICHIAMI DI ARCHITETTURA DEI SISTEMI DI ELABORAZIONE (p. 8)
3. ARCHITETTURA DEI SISTEMI OPERATIVI (p. 17)
4. I PROCESSI E LA LORO GESTIONE (p. 25)
5. LO SCHEDULER E LO SCHEDULING (p. 32)
6. TECNOLOGIE MULTI – PROCESSORE (p. 40)
7. SCHEDULING IN SISTEMI OPERATIVI UNIX/LINUX E WINDOWS (p. 43)
8. I THREADS (p. 50)

PROGRAMMAZIONE CONCORRENTE

9. LA PROGRAMMAZIONE CONCORRENTE (p. 56)
10. SINCRONIZZAZIONE NEL MODELLO AD AMBIENTE GLOBALE (p. 62)
11. PROBLEMI NEL MODELLO AD AMBIENTE GLOBALE (p. 67)
12. I MONITOR (p. 75)
13. IL PROBLEMA DEL DEADLOCK (p. 83)
14. SINCRONIZZAZIONE NEL MODELLO AD AMBIENTE LOCALE (p. 93)

GESTIONE DELLA MEMORIA E DELL'I/O

15. INTRODUZIONE ALLA GESTIONE DELLA MEMORIA (p. 98)
16. LA MEMORIA VIRTUALE (p. 109)
17. LA GESTIONE DELLA MEMORIA IN LINUX E IN WINDOWS (p. 116)
18. I/O E GESTIONE DEI DISCHI (p. 127)
19. I/O SCHEDULING (p. 134)
20. DISCHI A STATO SOLIDO (p. 141)
21. IL FILESYSTEM (p. 143)
22. FILESYSTEM IN LINUX E IN WINDOWS (p. 152)
23. LE MACCHINE VIRTUALI E LA VIRTUALIZZAZIONE DELLA CPU (p. 159)
24. LA VIRTUALIZZAZIONE DELLA MEMORIA E DELL'I/O (p. 166)
25. LE TECNOLOGIE VMWARE E CONTAINER – BASED (p. 170)
26. ANDROID (p. 173)

APPROFONDIMENTI LINUX/UNIX

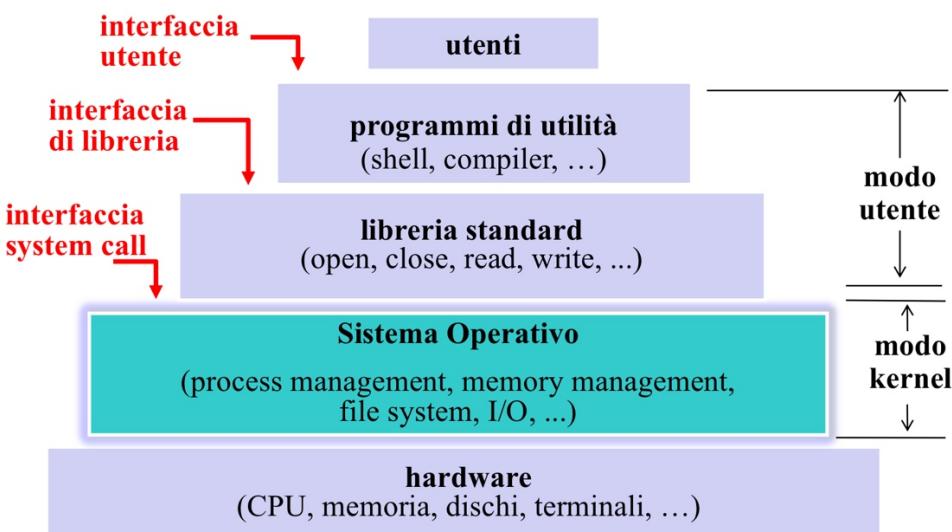
27. INTRODUZIONE LINUX/UNIX (p. 184)
28. IL CICLO DI SVILUPPO MODULARE (p. 189)
29. I PROCESSI IN UNIX (p. 193)
30. LA SHELL LINUX (p. 201)
31. SHARED MEMORY E SEMAFORI (p. 207)
32. POSIX THREADS (p. 214)
33. LO SCAMBIO DEI MESSAGGI (p. 218)
34. SVILUPPO DI MODULI PER IL KERNEL LINUX (p. 223)

GESTIONE E SCHEDULING DEI PROCESSI E THREADS

STRUTTURA ED EVOLUZIONE DI UN SISTEMA OPERATIVO

Un **Sistema Operativo** è un **componente software** di un sistema di elaborazione il cui compito principale è quello di **agire come intermediario tra gli utenti del sistema e le risorse hardware da impiegare per esaudire le loro richieste**; infatti, esistono principalmente **tre macro – componenti** in un sistema di elaborazione, che possono essere raggruppati in **tre livelli** (dal più basso al più alto):

1. **Livello hardware**, è l'insieme delle risorse e dei componenti fisici che eseguono le operazioni più elementari e comprende il processore, la memoria centrale, la memoria di massa e le periferiche di Input/Output;
2. **Livello del sistema operativo**, è l'insieme di tutti gli applicativi software che hanno il compito di gestire le risorse del livello hardware offrendo ai programmi applicativi un'interfaccia più semplice, generale e sicura da usare;
3. **Livello dei programmi applicativi**, è l'insieme di tutti gli applicativi software che possono essere utilizzati dagli utenti del sistema di elaborazione e che si appoggiano al Sistema Operativo per qualsiasi computazione.



Nella realtà, il **Sistema Operativo** non è impiegato solo per la gestione delle risorse, ma anche per gestire la sicurezza e i malfunzionamenti di un sistema di elaborazione: nascondendo la macchina fisica alle applicazioni, è possibile non solo decidere arbitrariamente quante risorse dare a quale applicativo ma anche impedire che questi accedano ad informazioni che devono rimanere private e che eventuali malfunzionamenti precludano l'accesso dell'utente al sistema stesso. Per i motivi appena enunciati, il Sistema Operativo è comunemente visto come un **insieme di software di gestione**.

Per poter eseguire un **programma**, con o senza l'impiego di un Sistema Operativo, è **necessario dedicare a quel programma una serie di risorse fisiche** ed eseguire una serie di operazioni preliminari come il caricamento in memoria delle istruzioni o l'assegnazione dell'entry point al program counter del processore. In un sistema di elaborazione in cui è **stato implementato un Sistema Operativo**, tutto ciò è svolto senza che il **programmatore debba intervenire** e, soprattutto, rende il **programma stesso trasportabile**; se, invece, lo stesso programma fosse realizzato per un sistema di elaborazione privo di Sistema Operativo, il **programmatore dovrebbe non solo occuparsi della gestione delle risorse e del caricamento in memoria ma anche di realizzare lo stesso programma applicativo per architetture che utilizzano componenti hardware diversi**. La

trasparenza al livello hardware che offre un Sistema Operativo è uno dei **principali vantaggi** del suo impiego, semplificando notevolmente la **programmazione commerciale** (quella che realizza applicazioni utente).

La **gestione delle risorse**, però, non si limita ai singoli applicativi ma anche alla **multiprogrammazione**; per multiprogrammazione si intende l'**insieme di tecniche che permettono a più utenti o a più applicazioni di accedere contemporaneamente alle stesse risorse hardware senza che uno debba attendere l'esecuzione degli altri**. Ad esempio, un utente avvia due programmi su uno stesso computer: il **Sistema Operativo**, a quel punto, non solo **si occupa** di associare ad ogni programma le risorse che necessita ma anche di **caricare in memoria alternatamente i due programmi in modo che sembri che vengano eseguiti contemporaneamente**.

Per quanto riguarda gli aspetti di **sicurezza**, altrettanto importanti quando si sviluppa un Sistema Operativo, si tenga in considerazione che è **necessario mantenere la riservatezza** (o privacy) e la **protezione delle informazioni**: un dato che non deve essere disponibile ad un programma applicativo non deve essere in alcun modo visibile ad esso, ma deve anche essere data la possibilità all'utente di decidere quali dati rendere disponibili a quali programmi. Tutto ciò è fatto con lo scopo di impedire a programmi potenzialmente malevoli (malware) di compromettere le funzionalità e l'accesso al sistema di elaborazione. Infine, si menzionano la **comprendere del funzionamento di un computer** e la **possibilità di diagnosticare meglio i problemi di affidabilità e prestazioni** come altri motivi per cui i Sistemi Operativi hanno l'importanza che nel mondo contemporaneo gli è data.

Nella pratica, il **Sistema Operativo agisce tramite le API** (Application Programming Interface) e le **system call** (o syscall); per API si intende l'**interfaccia di programmazione** che il Sistema Operativo offre agli applicativi utente per poter sfruttare le risorse del sistema, mentre le system call sono le operazioni stesse offerte dall'API ed indicate come quelle operazioni la cui esecuzione deve essere equivalente ad un'istruzione macchina (ovvero atomiche) ed è il motivo per cui sono chiamate anche **primitive**. Quindi, un programmatore applicativo utilizza le API, mentre un programmatore di sistema le crea.

Storicamente, i **sistemi in questione non hanno sempre avuto la struttura evidenziata**; inizialmente il costo delle materie prime e le tecnologie disponibili non permettevano sistemi **complessi ed articolati** come quelli odierni ma in concomitanza con il processo di miniaturizzazione (e quindi del progressivo abbassamento dei costi) e con l'avanzamento tecnologico sono stati resi possibili notevoli passi in avanti che hanno concretizzato tali sistemi. Dal punto di vista puramente numerico, i **passi in avanti eseguiti nell'arco di circa quarant'anni sono quantificabili con la seguente tabella**:

	1981	2016	Fattore migliorativo
MIPS	1	≈200000	200000
€/SPECInt	≈ 100K€	< 2	50000
DRAM size	128KB	16 GB	130000
Disk Size	10MB	1 TB	100000
Net Bwd	9600 bps	10 Gbps	1000000
Address bits	16	64	4
User/Machine	100	<1	100

In ordine cronologico, si sono susseguiti i seguenti tipi di sistemi:

- Single – User System

Caratterizzati dall'assenza di alcun Sistema Operativo, erano sistemi in cui era possibile l'esecuzione di un solo programma alla volta, il quale necessitava di essere manualmente caricato nella macchina attraverso ingombranti bobine di nastri magnetici contenenti le istruzioni da eseguire; si noti anche che era impossibilitata l'interazione dell'utente con il calcolatore durante l'esecuzione del programma.

Il problema principale di questi sistemi era la bassa percentuale di utilizzo delle risorse hardware, il che non permetteva alle aziende di coprire i costi della strumentazione in tempi ragionevoli; per percentuale di utilizzo si intende:

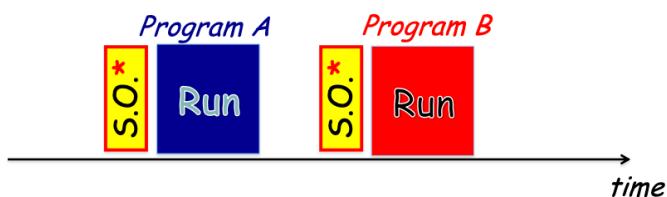
$$u\% = \frac{\text{tempo effettivo di uso dell'hardware}}{\text{tempo totale di utilizzo}}$$

Se il calcolatore richiedeva cinque minuti per poter caricare un programma, cinque per poterlo scaricare e cinque per poterlo eseguire, la percentuale di utilizzo era del 33%.

- Batch System

Il Batch System, o sistema a lotti, consiste nel raggruppare in lotti i "job", ovvero i programmi, di più utenti ed eseguirli in un'unica sessione in sequenza; la tecnologia che aveva permesso questo passo in avanti era quella delle schede perforate: schede rettangolari sui quali era possibile scrivere le istruzioni di un programma perforando in determinati punti la carta, garantendo un'ottimizzazione dello spazio decisamente più efficiente che con i nastri magnetici. Una volta caricati i lotti di job, l'utente la cui scheda doveva essere eseguita insieme alle altre doveva tornare dopo ore/giorni per ottenere i risultati, attendendo prima che tutte le schede fossero state elaborate; sono state limitate alcune problematiche dei sistemi Single – User ma rimane la non – interattività del sistema.

In questo tipo di sistemi si intravede per la prima volta un Sistema Operativo, un programma (monitor) residente nella memoria del computer incaricato di lanciare i job in sequenza:

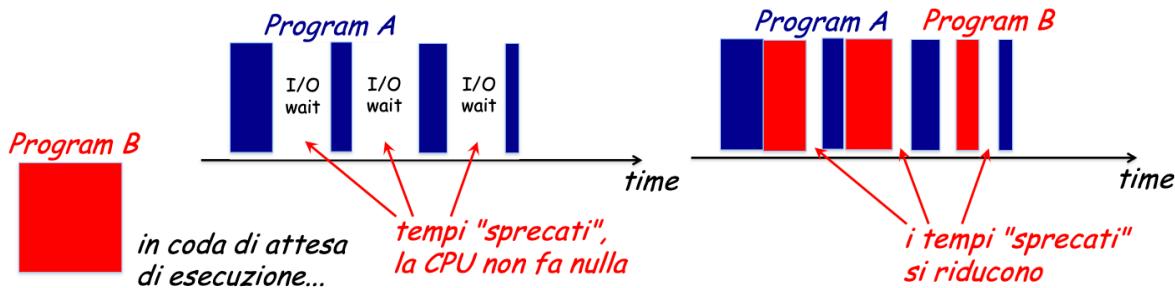


I sistemi in questione venivano detti anche monoprogrammati, dal momento in cui il calcolatore doveva prima terminare l'esecuzione di un programma prima di poter lavorare su un altro; questo svantaggio, seppur ancora limitante, veniva bilanciato dalla possibilità di ridurre l'interazione con la macchina al di fuori del tempo di esecuzione e di utilizzare in maniera più efficiente l'hardware.

- Sistemi multiprogrammati

Un ulteriore problema dei sistemi monoprogrammati era costituito dalla presenza di intervalli di tempo sprecati in cui la CPU era in attesa dell'I/O, senza la possibilità di eseguire altre istruzioni per tappare tali tempi morti. Come suggerisce il nome, un sistema multiprogrammato è realizzato in modo che in memoria RAM possano essere caricati più job contemporaneamente, che

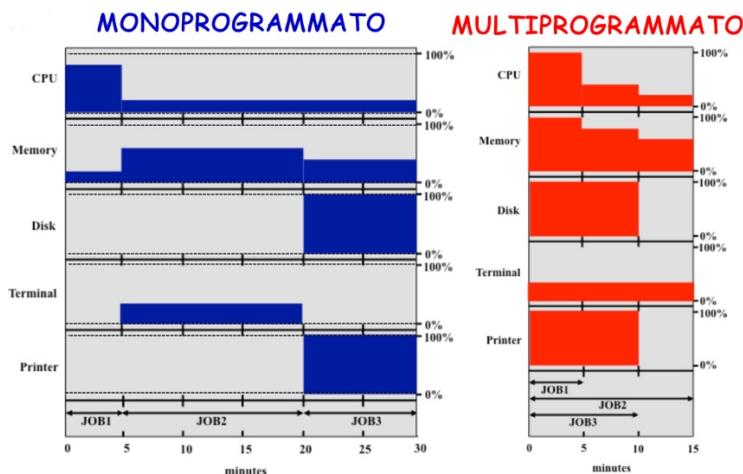
vengono **spezzettati ed eseguiti a tempi alterni**, nel frattempo che per l'altro programma si attendesse l'I/O:



Considerati i seguenti job:

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

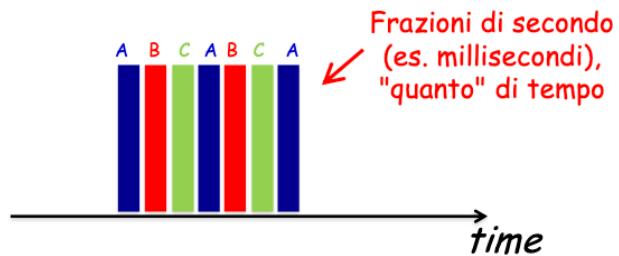
La differenza tra un approccio monoprogrammato e uno multiprogrammato può essere riassunta dai due grafici seguenti:



Dove si può facilmente apprezzare la riduzione del tempo totale di utilizzo della macchina.

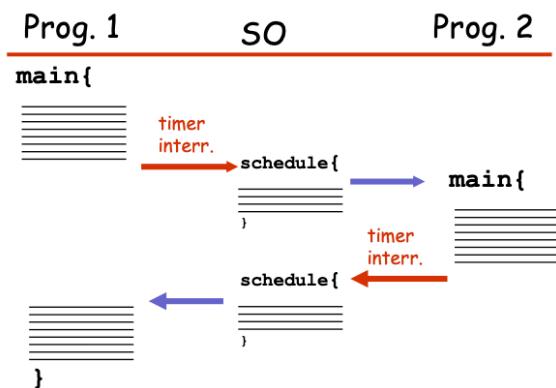
- **Time sharing**

Il time sharing estende ulteriormente il concetto di multiprogrammazione con l'aggiunta di **non costringere gli utenti ad attendere ore/giorni per avere dei risultati grazie alla possibilità di interagire con il programma in tempo reale**; questo tipo di sistemi sono costruiti in modo tale che **l'interruzione dei job avvenga forzatamente dal SO (preemption)**, anche se non fanno I/O, e in modo che sulla CPU si alternino con una frequenza molto maggiore (oltre le 100 volte al secondo):



Agli utenti, però, i programmi appaiono come se fossero eseguiti in contemporanea da processori diversi, nonostante lavorino con la stessa CPU.

Praticamente, il time sharing è reso possibile grazie all'innesto periodico di interrupt che spostano l'esecuzione da un programma ad un altro, salvando le informazioni essenziali di esecuzione in un secondo luogo per poi riprenderle all'interruzione che, successivamente, riattiverà il processo; l'operazione in questione prende anche il nome di **context switch** e, inevitabilmente, introduce un overhead di sistema:



Infatti, il context switch comunque prevede l'esecuzione di una subroutine, il cui tempo di esecuzione (sebbene piccolo) non è trascurabile; ad esempio, un context switch può richiedere fino ad un 1ms per essere eseguito che, con un periodo di interruzione è di 10ms, fa perdere circa il 10% del tempo di CPU al SO.

Tendenzialmente, l'approccio del Time Sharing si affianca al Batch multiprogrammato per la complementarietà dei relativi vantaggi e svantaggi:

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

- Personal Computer

Nei primi anni '80 l'accessibilità delle tecnologie ha reso sempre più diffuso l'accesso ad un calcolatore anche per usi domestici e privati, finché non è divenuto di uso comune il Personal

Computer, un sistema progettato per essere single – user e per porre enfasi sull’interfaccia utente e sull’interattività. Inizialmente, questi sistemi erano **mono – utente e mono – task** (come, ad esempio, MS – DOS) ma al giorno d’oggi c’è una prevalenza di sistemi **multiutente e Time Sharing** (come, ad esempio, Windows e MacOS).

- **Embedded e Mobile Computing**

In concomitanza con lo sviluppo e l’impiego di Personal Computer, **sono nate anche altre tecnologie, come l’Embedded e il Mobile Computing; il primo è impiegato perlopiù per applicazioni industriali e fa uso di SO di tipo real – time** (ovvero con applicazioni che lavorano con sistemi fisici e il cui funzionamento è legato al tempismo dei comandi), mentre **il secondo è un approccio riservato alla portabilità, alla sicurezza, all’efficienza energetica e ai multimedia**.

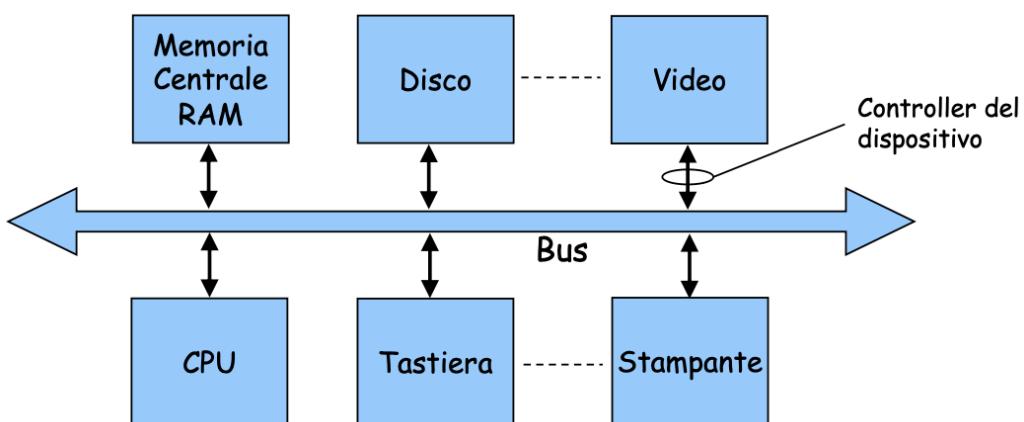
- **Distributed, Cloud Computing**

Questi approcci sono più moderni e si concentrano:

- Distributed Computing, sull’ulteriore **evoluzione nella virtualizzazione delle risorse** (il sistema è formato da gruppi, cluster, di macchine con un orchestratore che distribuisce le applicazioni degli utenti nel cluster);
- Cloud Computing, focalizzato sulla **disposizione dei data center “su domanda” da parte dei provider** (che si fanno pagare per la quantità di risorse usate);

RICHIAMI DI ARCHITETTURA DEI SISTEMI DI ELABORAZIONE

Poiché un Sistema Operativo implementato in un calcolatore elettronico è a stretto contatto con l’hardware del sistema, si può ben intuire che la sua comprensione non può prescindere dal funzionamento di un computer. Seguendo il **modello di Von Neumann**, un moderno calcolatore elettronico è individuato da **una CPU**, l’unità di calcolo elementare, da **una memoria centrale**, detta anche RAM, e da **una serie di dispositivi di ingresso e/o uscita**; ognuno di questi componenti è collegato agli altri tramite un **bus di sistema**, un “insieme di fili” su cui viaggiano i risultati dell’elaborazione e gli indirizzi verso cui questi sono diretti, divisi in pacchetti di 8/16/32/64 bit. Nella sua totalità, **un calcolatore elettronico si figura come segue**:



La CPU è il componente adibito all’esecuzione delle istruzioni di un programma conservato nella memoria centrale; durante tale esecuzione, il processore attraversa diverse fasi, le quali coinvolgono frequentemente l’uso di registri interni, unità di memoria piccole ma ad accesso

rapido con le quali è scongiurato ogni rallentamento dovuto al fetching in memoria. **I registri vengono categorizzati in registri generali** (o programmabili) e **in registri di stato e controllo**: i primi sono quelli utilizzati per contenere dati ed indirizzi di locazioni di memoria durante l'esecuzione del programma e servono solo come spazio predisposto al rapido accesso alle informazioni rilevanti al programma (come gli addendi per un programma che fa la somma), mentre **i secondi si occupano di tenere traccia dello stato del processore e dell'esecuzione del programma e non sono solitamente accessibili al programmatore**; i più importanti registri interni della CPU sono il **Program Counter (PC)**, lo **Status Register (SR**, o Program Status PS) e lo **Stack Pointer (SP)**.

Il processore, in un calcolatore moderno avanzato, oltre che ad accedere in memoria per prelevare le informazioni, può accedere alle periferiche di I/O (come una tastiera per prelevare l'input); tuttavia, l'accesso alla periferica non è diretto, dal momento in cui la CPU non conosce la struttura del sistema a priori e i due lavorano a frequenze e intensità diverse, ma accede ad un particolare circuito elettronico, il controllore, che è realizzato appositamente per interporsi tra periferica e CPU tramite il bus di sistema. In aggiunta, per poter realizzare una corretta comunicazione tra controllore e CPU, si installa in memoria un device driver, un software di controllo del dispositivo che, in esecuzione sulla CPU, rende noto a quest'ultima le operazioni eseguibili sulla periferica; per ora non è importante digredire più di tanto, si tenga solo in considerazione che quando si parla di **device driver** si fa riferimento ad una parte del Sistema Operativo.

Generalmente, un controller ha tre tipi di registro: dato, stato e controllo; il device driver legge il registro di stato (che è solo di lettura) per determinare lo stato della periferica (ad esempio, input da tastiera finito), mentre sul registro di controllo (che è solo di scrittura) può impartire comandi e, infine, sul registro di dato può leggere/scrivere per comunicare dati dalla/verso la periferica.

Prima di procedere a descrivere l'effettivo meccanismo con cui la CPU interagisce con i dispositivi di I/O, si vuole fare una **digressione sul modo in cui si accedono agli indirizzi e in cui si opera su una periferica**. Negli anni si sono consolidati principalmente due modi con cui le periferiche vengono mappate dalla CPU:

- **Port – mapped I/O**, con il quale la CPU legge e scrive sui registri del device con delle istruzioni macchina speciali e ad hoc per quella periferica (ad esempio, istruzioni come IN o OUT) ed è un approccio obsoleto e tipico dei vecchi sistemi con bus a parallelismo ridotto (ad esempio, 8 bit);
- **Memory – mapped I/O**, con il quale la CPU va a considerare i registri di memoria delle periferiche come delle aree di memoria centrale, permettendo così di dedicare degli indirizzi facilmente accessibili ad aree di memoria esterne e di utilizzare le stesse istruzioni che si utilizzano per l'accesso in memoria (come la MOV).

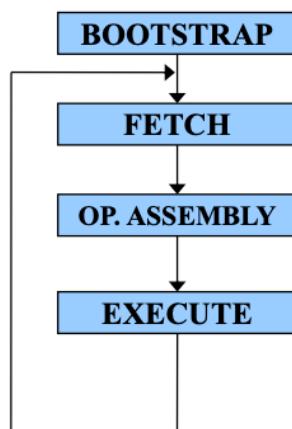
La differenza principale tra i due metodi è nel modo in cui è gestito lo spazio di indirizzamento: il primo metodo separa gli spazi di indirizzamento della periferica e della memoria RAM, mentre il secondo riserva una piccola quantità di indirizzi di quest'ultima ai registri della periferica, escludendo così la possibilità di accedervi. I due approcci, chiaramente, hanno i propri svantaggi ed i propri vantaggi: sebbene il primo approccio aggiunga un overhead in fase di programmazione, non preclude l'accesso ad alcun indirizzo di memoria RAM, cosa che invece accade utilizzando il secondo metodo; tuttavia, negli anni si è preferito un approccio semplificato, dal punto di vista della programmazione, al costo di una quantità comunque ridotta di registri non puntati da alcun indirizzo.

In aggiunta, è possibile anche implementare un dispositivo, il DMA (Direct Memory Access), che opera al posto della CPU i trasferimenti da e/o verso la memoria (o le periferiche) per evitare l'occupazione della capacità computazionale del processore durante il trasferimento di grandi blocchi di dati e il numero di interruzioni che vanno gestite.

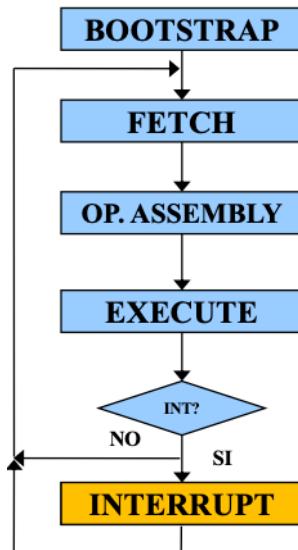
Le periferiche, in quanto componenti non direttamente adibiti al calcolo computazionale, lavorano con tempi di diversi ordini di grandezza superiori rispetto a quelli di una CPU; questa discrepanza porta con sé notevoli problemi di efficienza, soprattutto quando viene richiesta l'attenzione del processore per una qualsiasi operazione svolta dalla periferica: in tale situazione, la CPU dovrebbe entrare e permanere in un ciclo di attesa fintantoché non viene mostrato nel registro di stato il valore corrispondente alla possibilità di interagire. Si può facilmente intuire come questo modo di operare, detto polling, sia particolarmente inefficiente, dal momento in cui per intervalli di tempo non trascurabili, rispetto al proprio tempo di lavoro, la CPU è ferma e la sua capacità computazionale sprecata (questa condizione è denominata busy waiting); per questo motivo negli anni sono state sviluppate delle metodologie operative più efficienti del polling, dette meccanismi interrupt – based.

Implementando questo tipo di interruzioni, la CPU esegue il proprio lavoro su altre attività quando la periferica non è disponibile, evitando così il busy waiting, e nell'istante in cui questa lo diventa solleva un segnale di interrupt, con il quale il controllo della CPU è passato alla ISR (Interrupt Service Routine, un software che fa parte del driver e con il quale sono gestite le interruzioni di una particolare periferica). Essenzialmente, le interruzioni permettono di alternare la CPU tra l'esecuzione di programmi dell'utente e la gestione dei dispositivi di I/O, in modo che quando questi richiedono attenzione il processore si dedica temporaneamente ad essi; l'importanza delle interruzioni per la multiprogrammazione e del time sharing è facilmente intuibile, aumentando l'efficienza e l'utilizzo delle risorse, ed è il motivo per cui questi meccanismi sono approfonditi più nel dettaglio rispetto al polling.

Il ciclo della CPU di un calcolatore che segue la struttura di Von Neumann si compone, in una configurazione in cui è assente l'implementazione di interruzioni, nel modo seguente:



E si modifica quando si implementa la gestione delle interruzioni:



Durante la fase di esecuzione dell'istruzione, la CPU potrebbe ricevere un segnale di interruzione, con il quale il flusso del programma è reindirizzato alla ISR, la quale viene eseguita a partire dal ciclo successivo e finché non termina (tramite la generazione di un nuovo segnale che riporta il flusso del programma laddove si era fermato e il processore nello stato in cui si trovava); si noti che la CPU, in questo alternarsi di programmi, non è cosciente di tutte le operazioni e di tutti i salti che si stanno effettuando, si occupa solo di eseguire l'operazione che gli è portata da un registro di memoria (quale non gli è dato saperlo) durante la fase di fetch.

Le interruzioni possono essere categorizzate sul fatto che siano:

- **Sincrone o asincrone** rispetto al programma;
- **Richieste o subite** dal programma.

Analizzando le prime nel dettaglio:

- **Interruzioni asincrone**, (chiamate anche semplicemente interruzioni) sono richieste di “attenzione” da parte dei dispositivi di I/O e possono verificarsi in qualunque momento durante l'esecuzione del programma;
- **Interruzioni sincrone**, (chiamate anche semplicemente traps o eccezioni) sono auto – generate dalla CPU e si verificano dopo l'occorrenza di specifici eventi del programma in esecuzione;
 - Ad esempio, sono interruzioni sincrone l'accesso ad indirizzi di memoria “illeciti” o non allineati, come un indirizzo nullo o dispari per alcune architetture (come il M6800), oppure operazioni aritmetico – logiche limite, come la divisione per zero;
 - A queste interruzioni il Sistema Operativo risponde “uccidendo” il programma.

Mentre le seconde:

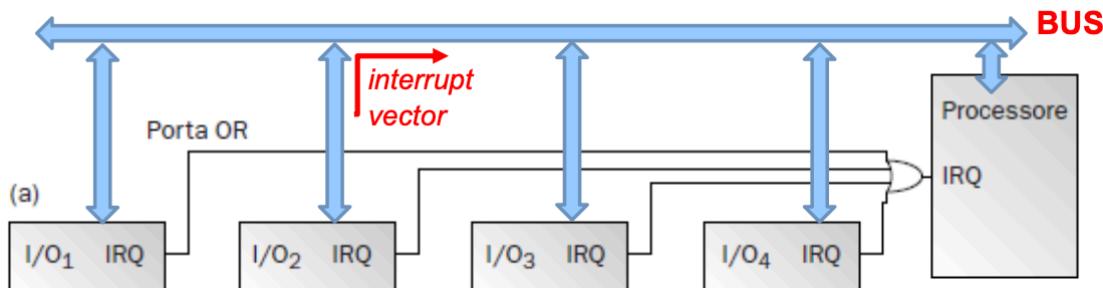
- **Interruzioni richieste**, sono interruzioni sincrone utilizzate volontariamente per innescare l'esecuzione del sistema operativo e, pertanto, vengono sollevate durante la fase di esecuzione dell'istruzione;
 - Ad esempio, breakpoint usati dai debugger per inserire istruzioni speciali, verificare il contenuto di alcune variabili prima o dopo una modifica o fermare l'esecuzione arbitrariamente;

- **Interruzioni subite**, sono quelle che rientrano nei casi precedenti, ovvero quelle che solleva un programma quando occorrono situazioni pericolose.

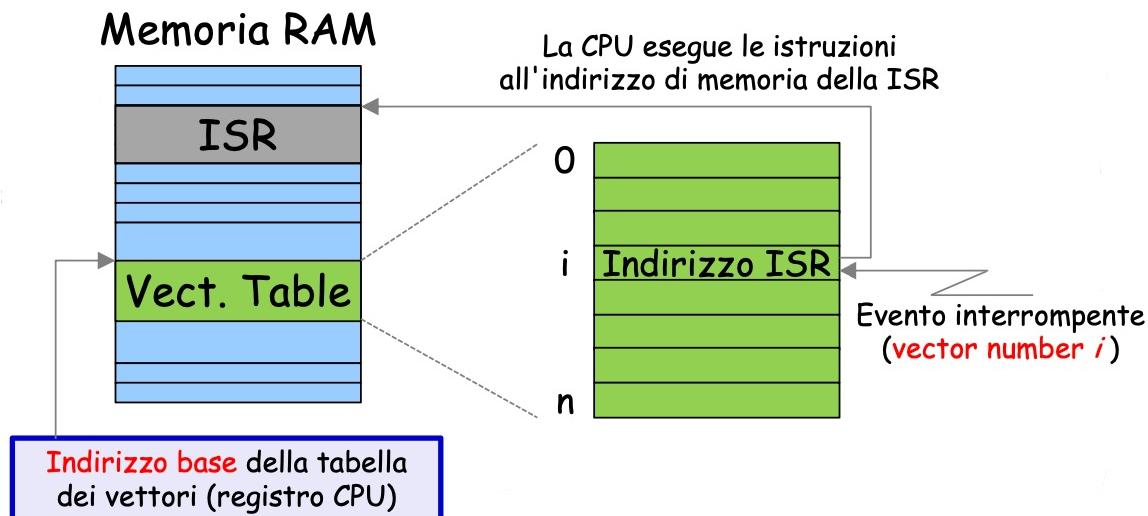
La lista di interruzioni più comuni e la loro classificazione è proposta di seguito:

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

Sulla base di quanto appena detto, si può intuire la possibilità di abilitare o disabilitare le interruzioni per mezzo di apposite istruzioni, Set Interrupt (SII) e Clear Interrupt (CLI), che forzano la CPU a dirigersi verso un'interruzione o ad ignorare le interruzioni occorse.



Poiché risulta impensabile che ogni singola periferica sia collegata alla CPU per mezzo di bus proprietari, lungo una stessa linea possono essere indirizzati più dispositivi. Sorge, così, il problema di identificare quale periferica ha sollevato una determinata istruzione; per fare ciò, la CPU distingue un dispositivo da un altro sulla base di un identificativo fornito dal dispositivo stesso, il **vector number**, che identifica anche la ISR da eseguire tra quelle memorizzate in una tabella, detta **interrupt vector table**.



Nel dettaglio, la pipeline di un'interruzione è la seguente:

1. All'occorrenza, un segnale di interrupt request (INT) viene inviato dalla periferica alla CPU;
2. La CPU finisce l'esecuzione dell'istruzione corrente;
3. La CPU verifica la presenza del segnale INT ed invia un segnale di conferma (ACK) al dispositivo per comunicare la presa in carico dell'interruzione;
4. La CPU salva sullo stack del sistema operativo le informazioni necessarie a riprendere, al termine dell'esecuzione della ISR, l'esecuzione del programma interrotto:
 - a. Program Counter (PC);
 - b. Stack Pointer (SP);
 - c. Status Register (SR);
5. La CPU seleziona la ISR corretta tramite il vettore di interrupt;
6. La CPU carica dal vettore di interrupt:
 - a. Il registro PC, con l'indirizzo iniziale della ISR;
 - b. Il registro PS, attivando anche la modalità supervisore della CPU;

(Si consideri che, **fino a questo momento, tutte le operazioni sono svolte dall'hardware** senza alcun intervento software).

7. La ISR salva lo stato del processore (come i registri – dato non salvati in precedenza dalla CPU) su uno stack;
8. La ISR gestisce l'interruzione interagendo con la periferica;
9. La ISR ripristina lo stato del processore (operazione inversa alla 7);
10. Il controllo ritorna al processo interrotto, con conseguente ripristino dei registri PC, SP e SR.

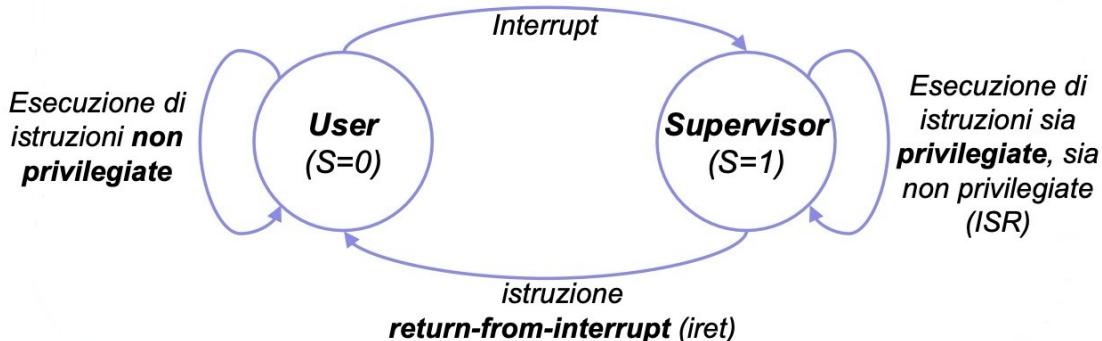
(Si consideri che **queste ultime operazioni sono svolte a livello software**).

In questa carrellata di operazioni è stata menzionata la **modalità supervisore**, una delle due possibili modalità di funzionamento di una CPU utile ed importante alla protezione delle risorse hardware; i moderni processori, infatti, possono lavorare in:

- **User mode** (o modalità non privilegiata);
- **Kernel mode** (o modalità supervisore/privilegiata).

Quest'ultima abilita le istruzioni per l'accesso alle risorse fisiche, dette **istruzioni privilegiate**, come le ISR che, altrimenti, non sarebbero accessibili per motivi di sicurezza e accessibilità (un

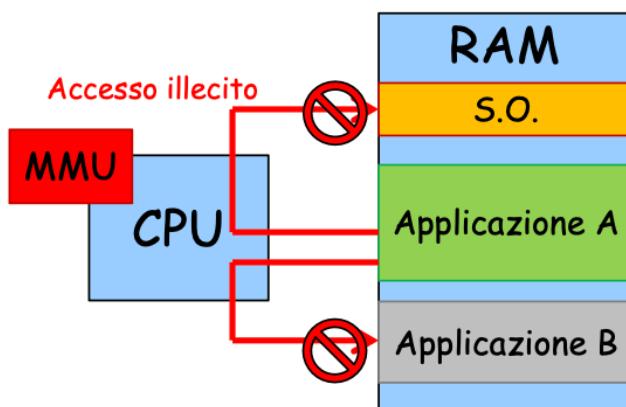
programma utente non deve poter manipolare le risorse hardware, altrimenti potrebbe comprometterne la funzionalità per altri). Lo stato supervisore/utente è individuato dalla CPU da un bit S nel registro di Stato; quando la CPU rileva e attiva un'interruzione, più precisamente all'avvio della ISR, imposta S = 1, portandosi in Kernel mode, mentre ritorna a S = 0 quando incontra l'istruzione di ritorno da ISR (IRET), portandosi di nuovo in User mode.



Degli esempi di istruzioni privilegiate sono le seguenti:

- SLI e CLI per l'abilitazione e la disabilitazione delle interruzioni;
- Modifica dei registri per la gestione degli interrupt (indirizzo base interrupt vector table);
- Istruzioni per la gestione della memoria;
- Istruzioni per la I/O port – mapped;
- Istruzioni per la modifica del registro di stato PS;
 - PS contiene anche il bit di stato utente o supervisore, non deve essere modificabile se non dal supervisore.

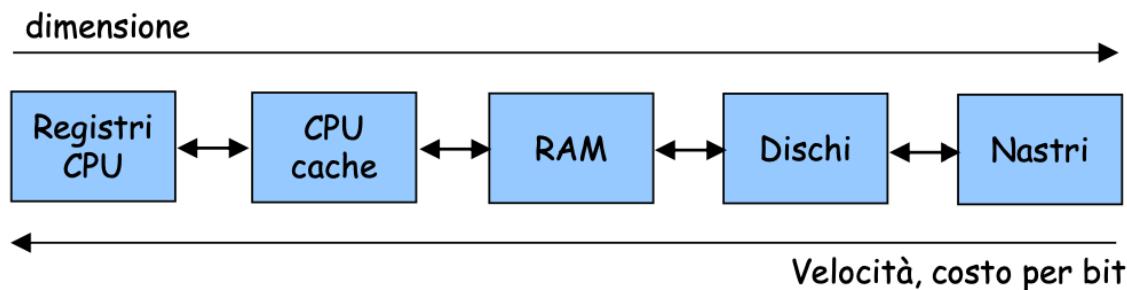
Nei sistemi multiprogrammati e multiutente, più applicazioni condividono la stessa area di memoria; la MMU (Memory Management Unit) affianca la CPU nella protezione di codice e dati del SO dalle applicazioni e di un'applicazione dalle altre:



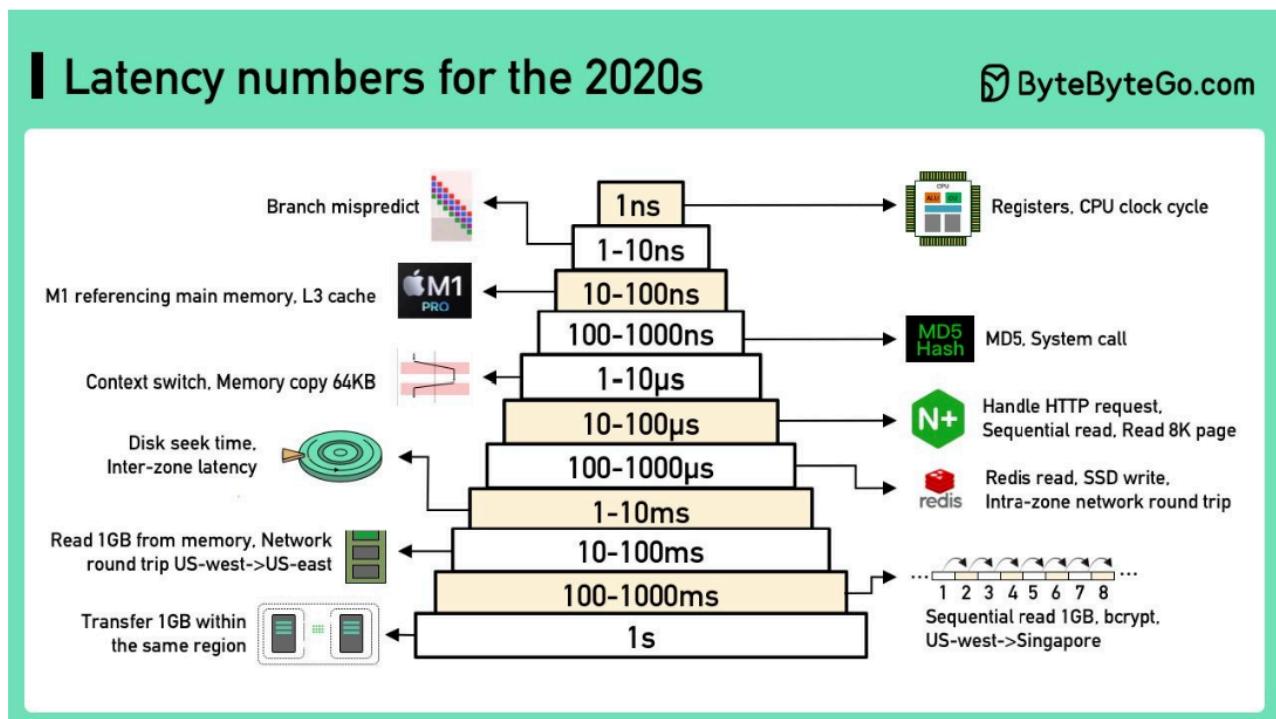
In un moderno calcolatore esistono tre tipologie principali di memorie:

- **Memorie Centrale** (o RAM), spazi di archiviazione volatile che possono essere acceduti direttamente dalla CPU;
- **Memorie Secondarie**, spazi di archiviazione non volatile con un'alta capacità di memorizzazione (Dischi magnetici, Memorie a stato solido, Nastri magnetici, ...);
- **Memorie Interne**, spazi di archiviazione con bassa capacità di memorizzazione ma elevata velocità di accesso (cache e registri interni).

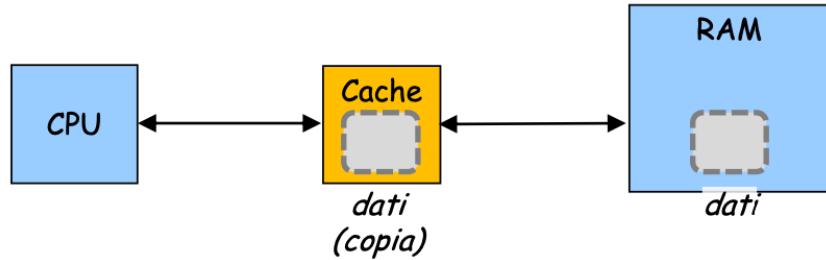
Tutte le memorie in questione possono essere organizzate in una gerarchia di memoria che segue tre caratteristiche: **velocità, costo e dimensione**. In particolare, **velocità e costo per bit sono inversamente proporzionali alla dimensione**, in una schematizzazione che segue la figura seguente:



La **necessità di avere più memorie**, con velocità (e quindi dimensioni) differenti, risiede nella **necessità di immagazzinare sempre più dati ma preservarne comunque la velocità di accesso**; infatti, se si aumentasse solamente la capacità della memoria centrale si arriverebbe ad un punto in cui la velocità di accesso risulta di troppo più lenta della velocità di lavoro della CPU, riducendo drasticamente la percentuale di utilizzo del componente. In genere, i **tempi di lavoro dei componenti di un computer sono riassunti dalla seguente tabella**:



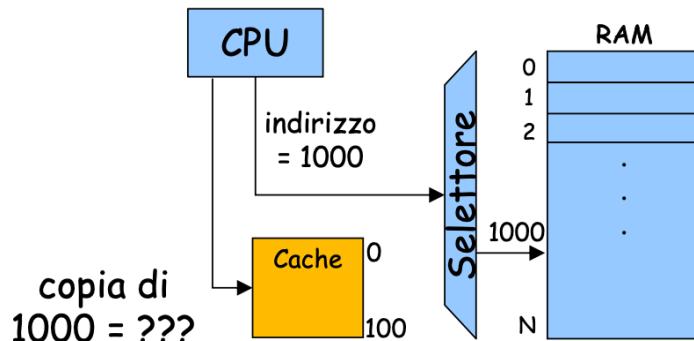
Si parla di **caching** in riferimento alla **pratica di conservare nelle memorie veloci una copia** (di un sottoinsieme) dei **dati delle memorie lente**; pertanto, se il dato è **presente in cache** (cache hit) vi si accede **velocemente** (la cache è la memoria più veloce), altrimenti (cache miss) si accede alla memoria centrale:



Il caching è efficace grazie alla validità dei principi di località:

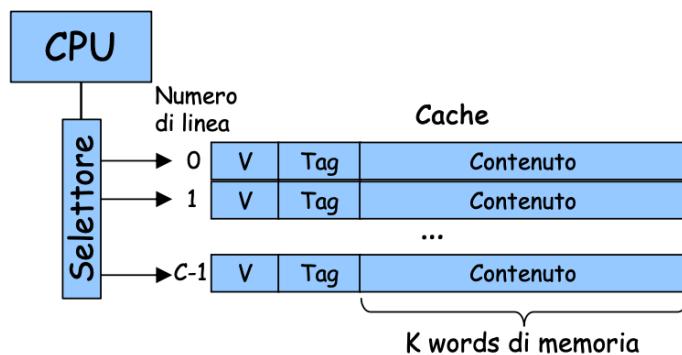
- **Località spaziale**, i programmi tendono ad accedere a dati vicini;
- **Località temporale**, i programmi tendono ad accedere più volte agli stessi dati.

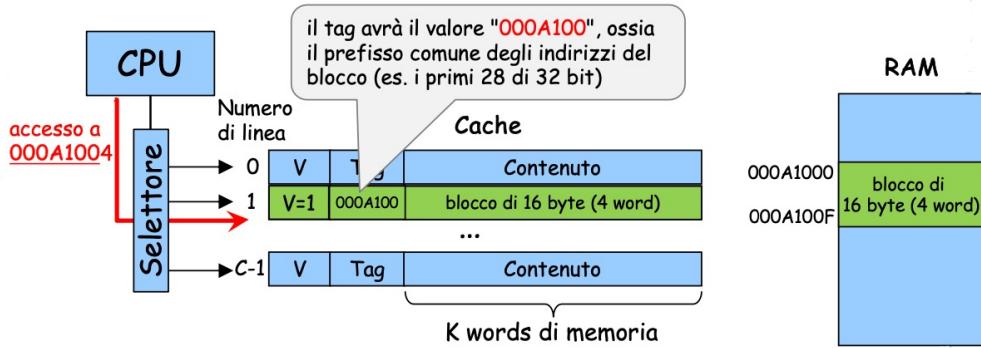
La RAM è una memoria abbastanza capiente (decine di GB) e, pertanto, il suo spazio di indirizzamento è un insieme di valori molto ampio; la CPU accede alla RAM in base alla posizione del codice/dei dati indicandone l'indirizzo (**selezione lineare**, tramite un DEMUX):



Di contro, la cache è una memoria molto più piccola e non è pratico usare l'indirizzo come "posizione" per trovare la copia del dato. La cache risolve il problema tramite un **principio di selezione associativa**, con cui si memorizza la coppia indirizzo – dato in posizione arbitraria e, per trovare il dato, si ricorre alla ricerca della coppia che contiene l'indirizzo indicato dalla CPU.

In genere, la cache è così ordinata:





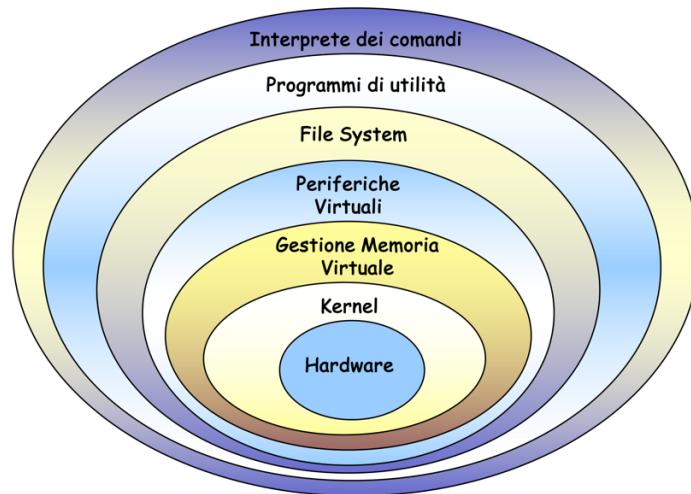
Si individuano:

- **Contenuto**, la copia di un blocco di dati contigui della memoria centrale;
- **V**, bit di validità che indica se la linea di cache è occupata;
- **Tag**, identifica il contenuto tramite un prefisso comune agli indirizzi del blocco (solitamente è l'indirizzo senza il LSB, 000A100F → 000A100).

L'accesso alla cache, con verifica del tag e validità del contenuto, **avviene in parallelo**. Il caching è un concetto che si applica a vari livelli, esistendo una cache della memoria centrale nella CPU, una del disco in memoria centrale e una di filesystem distribuiti in filesystem locali; inoltre, richiede una politica di gestione per la quale sono predefiniti dimensione della cache e dei blocchi, una funzione di mapping del blocco e algoritmi di sostituzione ma, nonostante ciò, non mancano problemi di coerenza.

ARCHITETTURA DEI SISTEMI OPERATIVI

Come si è già potuto osservare, un **Sistema Operativo assolve a due principali compiti: virtualizzare le risorse hardware** (in quanto fornitore di servizi), come il filesystem, i processi e le periferiche, e **gestire e coordinare i vari attori che sono coinvolti nella computazione**, insieme poi a meccanismi di protezione e comunicazione.



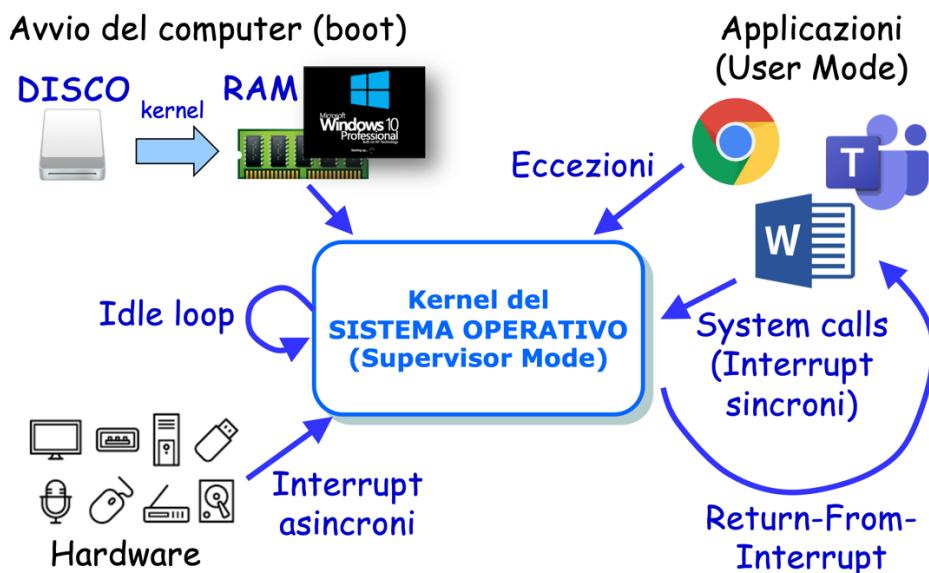
Il Sistema Operativo, quindi, può essere visto come una macchina virtuale che mette in comunicazione i livelli più alti di utilizzo di un computer con le risorse di basso livello.

L'elemento più importante in un Sistema Operativo è il **kernel** (nucleo), la parte del SO che risiede in memoria e che implementa e include le sue funzionalità fondamentali.

Al livello del kernel, la macchina virtuale realizzata dal Sistema Operativo contiene tante CPU quanti sono i processi (virtuali) attivi, non possiede meccanismi di interruzione e possiede istruzioni di sincronizzazione e scambio messaggi tra i processi che operano sui processori virtuali. Invece, a livello della gestione della memoria, la macchina virtuale realizzata dal Sistema Operativo consente di fare riferimento a spazi di indirizzamento virtuali, garantisce la protezione e consente (in alcuni casi) di ignorare se il programma e/o i dati in uso sono fisicamente residenti in memoria centrale o su memoria di massa. Invece, a livello della gestione delle periferiche, la macchina virtuale del Sistema Operativo dispone di periferiche dedicate ai singoli processi, maschera le loro caratteristiche fisiche e ne gestisce parzialmente i malfunzionamenti. Invece, a livello del filesystem, la macchina virtuale realizzata dal Sistema Operativo offre strutture logiche (come file e directory) per memorizzare blocchi di informazioni, ne controlla gli accessi e ne gestisce l'organizzazione fisica su memoria di massa.

A questo punto, sorgono spontanee due domande: **Qual è il flusso di controllo e come è invocato un Sistema Operativo? Come è composto l'interno di un Sistema Operativo?** Per rispondere è importante essere consapevoli del fatto che i Sistemi Operativi sono guidati dalle interruzioni (sincrone e asincrone), le quali guidano l'avvicendamento dei processi, e che gran parte del kernel viene eseguito come **Interrupt Handler (ISR)**.

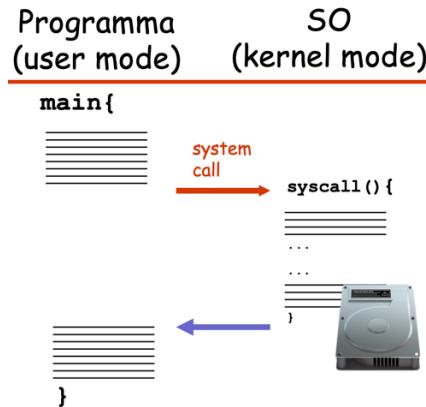
Nella sua interezza, il ciclo di esecuzione di un Sistema Operativo è il seguente:



All'avvio del computer, il kernel è trasferito dal disco alla RAM per poter essere eseguito e per poter rendere il computer operativo; dopodiché intervengono le interruzioni sincrone, asincrone o le eccezioni, sollevate dalle periferiche o dalle applicazioni (in User Mode), che richiedono l'operatività del kernel (in Supervisor Mode). Negli intervalli di tempo in cui non sono sollevate interruzioni o eccezioni, il kernel si trova in uno stato di **idle loop** in cui non esegue alcuna operazione ed attende una qualsiasi invocazione che lo renda operativo.

Per poter operare su una risorsa, i programmi devono effettuare una chiamata di servizio al Sistema Operativo, detta **system call**, la quale si configura come una richiesta di esecuzione di istruzioni privilegiate per conto del chiamante. Una **system call** corrisponde all'attivazione di

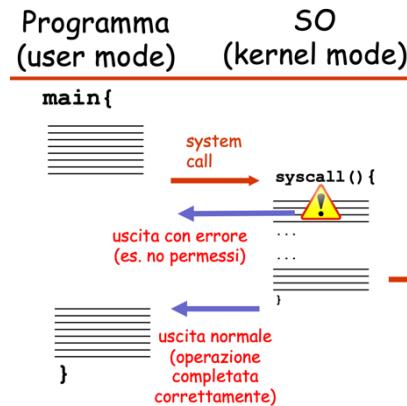
una parte del kernel a favore del processo chiamante, per espletare il servizio o per acquisire la risorsa richiesta.



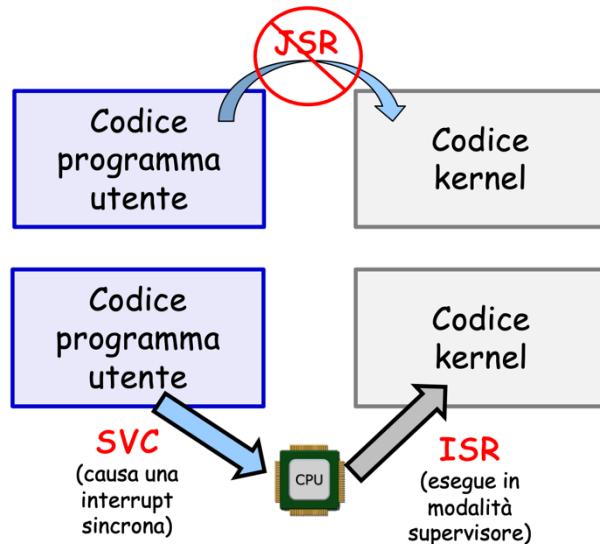
Le **system call** possono essere **categorizzate** come segue:

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

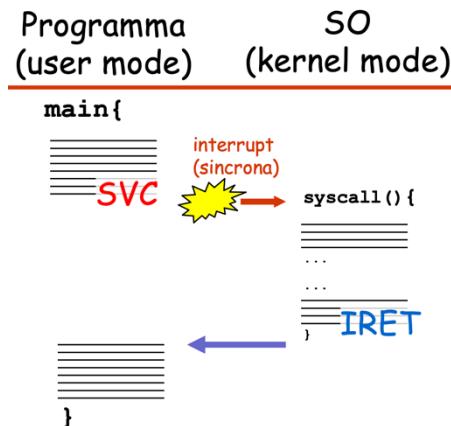
Il fatto che **il kernel sia l'unico programma fra tutti che può lavorare in modalità supervisore** (le applicazioni, potenzialmente buggate o malevole, lavorano sempre in modalità utente) **garantisce le proprietà di sicurezza del SO che sono state finora ostentate**; infatti, da questo punto di vista, **il kernel non è altro che un intermediario nell'accesso delle risorse**, in modo da poter **effettuare adeguati controlli di sicurezza e terminare la system call nel caso in cui il chiamante non avesse i permessi di accesso**.



Con queste informazioni, è chiara la necessità di un kernel: i programmi in modalità utente non possono invocare una system call con una semplice istruzione di salto (come una JSR) e neanche modificare da sé il proprio livello di privilegio (andrebbe in contraddizione con la definizione e l'utilità di livello di privilegio). Il programma, quindi, è obbligato ad utilizzare un'istruzione particolare, la SVC (Supervisor Call) che innesta una interrupt sincrona e l'esecuzione di una ISR, che può eseguire le operazioni privilegiate.



L'istruzione di macchina **SVC** innesca un interrupt sincrono, come già anticipato, ed in risposta il SO esegue la system call e svolge un'operazione per conto del programma chiamante, come un meccanismo di delega, per poi restituire il controllo all'applicazione con un'istruzione di ritorno **IRET**.



Il passaggio di parametri di ingresso e uscita alle system call deve seguire le regole e i protocolli imposti dal sistema operativo (Application Binary Interface); Linux adotta le seguenti convenzioni:

- Codice numerico della syscall: EAX;
- Parametri di ingresso:
 1. EBX;
 2. ECX;
 3. EDX;
 4. ESI;
 5. EDI;
 6. EBP
- Valore di ritorno: EAX.

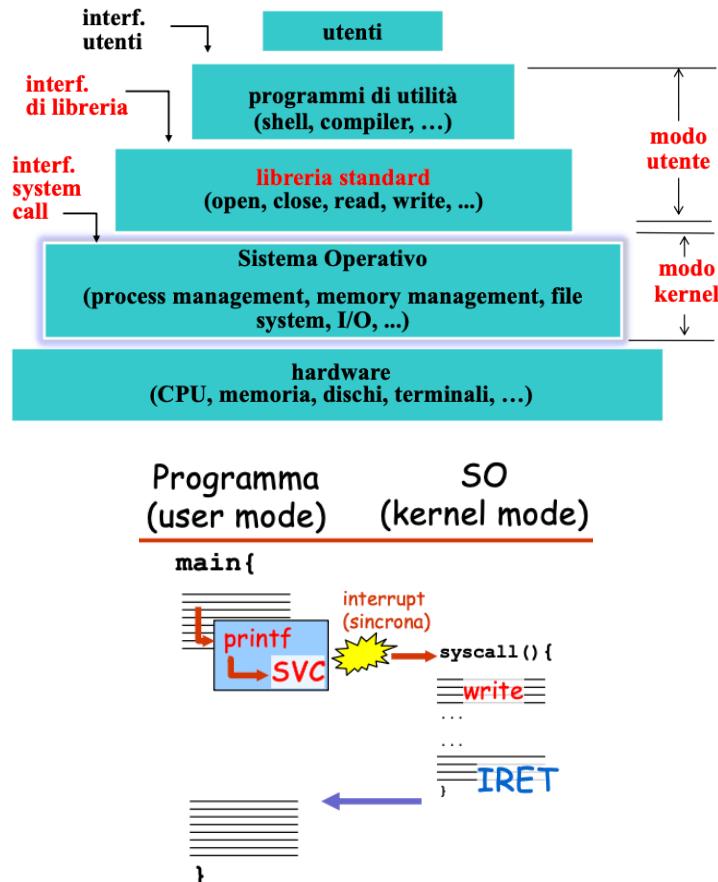
Ad esempio, in sistemi Linux su x86 (32 bit), la syscall per la funzione **write** è la seguente:

```
mov $4, %eax          //EAX = codice della syscall (4 = write)
mov $1, %ebx          // EBX = file destinazione (1 = il terminale)
```

```
mov $stringa, %ecx // ECX = indirizzo della stringa ("Hello World!")
mov $6, %edx        // EDX = lunghezza della stringa (6 char)
int $0x80           // syscall per sistemi a 64 bit
```

Tipicamente, i linguaggi di programmazione forniscono routine di interfaccia che trasformano una tradizionale chiamata di procedura in una **SVC**, facilitando la chiamata e lo scambio di parametri; ad esempio, nel linguaggio C la funzione di libreria standard `write()` nasconde la chiamata di sistema scritta nell'esempio precedente:

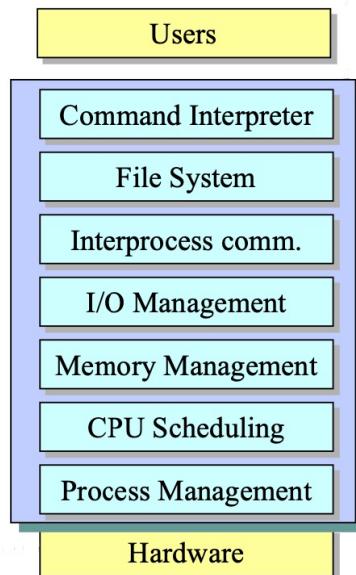
```
write (STDOUT, "Hello World!", len);
```



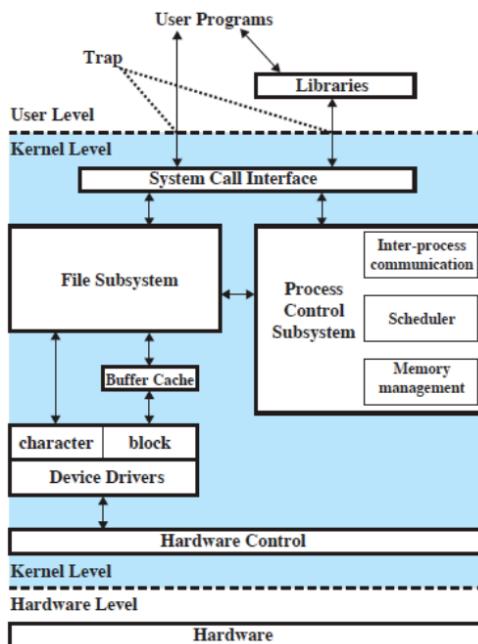
Sono proprio le librerie standard che, agendo da interfaccia tra i programmi utente e il sistema operativo e operando nel modo utente, permettono di semplificare le chiamate di sistema; infatti, è la funzione di libreria che usa la **SVC** per la chiamata di sistema, liberando il programmatore high level di questa responsabilità.

Il Sistema Operativo, come si è potuto intuire, è un software di notevole delicatezza e complessità, nonché di importanti dimensioni; i primissimi sistemi operativi mai sviluppati erano costituiti da un unico programma, senza particolari suddivisioni, composto da una “raccolta di funzioni” scritte in linguaggio macchina e ad ognuna delle quali corrispondeva una system call. Questo approccio prende il nome di **architettura monolitica** ed è particolarmente inadeguato per i **complessi sistemi moderni**, basti pensare che lo IBM OS – 360, SO batch multiprogrammato costituito da oltre un milione di istruzioni macchina, ha continuato a produrre errori per molto tempo durante tutto l’arco della sua vita.

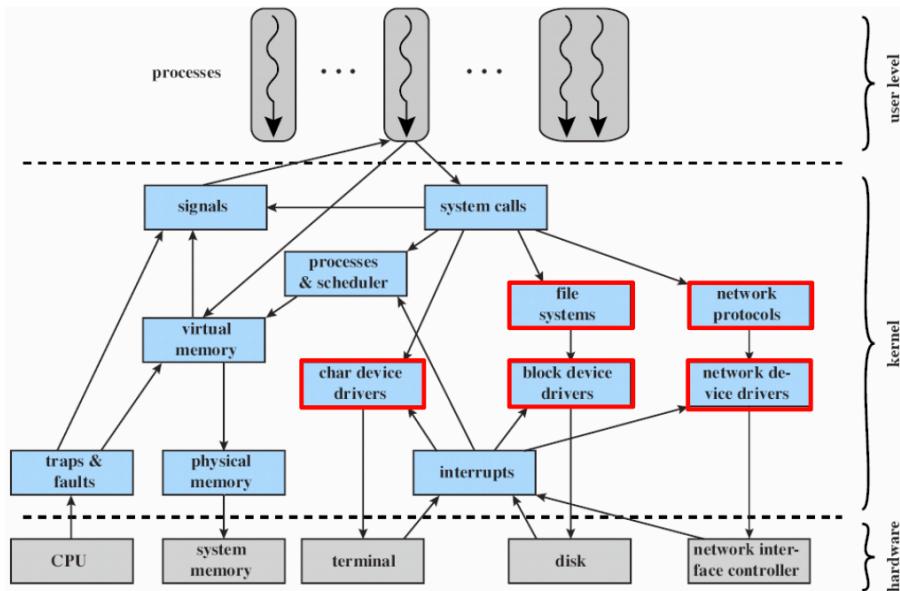
La progettazione del SO richiede di applicare gli approcci propri dell'ingegneria del software allo scopo di garantire correttezza, modularità, facilità di manutenzione, efficienza di esecuzione e sicurezza. **I SO ad architettura modulare suddividono il codice in più moduli**, le cui interfacce ne descrivono le funzionalità e il cui corpo le implementa; i vantaggi di questo approccio si evidenziano nel ridotto impatto di una modifica ad un modulo e nella possibilità di caricare in memoria i soli moduli necessari.



Un esempio di SO ad architettura modulare è UNIX:



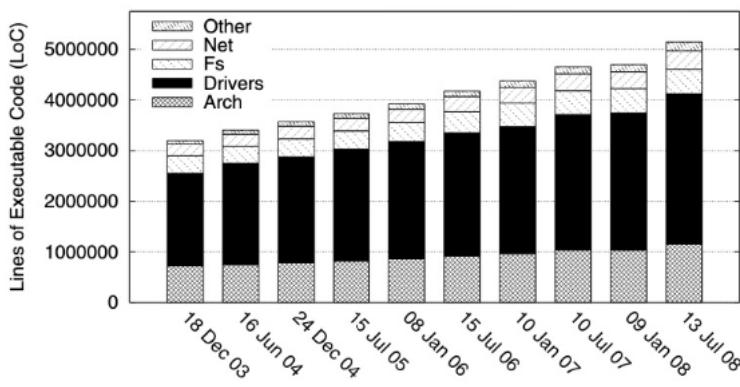
Oppure Linux:



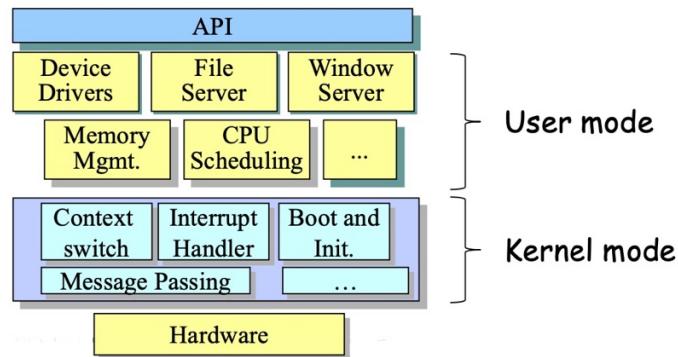
Quelli evidenziati in rosso sono i moduli caricabili che possono essere collegati/scollegati dal kernel a tempo di esecuzione.

Un aspetto rilevante nella progettazione di un SO è la **suddivisione dei componenti tra lo user – space e il kernel – space**: i primi sono eseguiti come se fossero delle applicazioni utente, quindi in **modalità non privilegiata**, mentre i secondi godono della **modalità supervisore**; generalmente, spostare i moduli dal **kernel – space** allo **user – space** favorisce prestazioni (nonostante le chiamate di sistema sono molto più lente delle comuni chiamate a funzione), **sicurezza e modularità**, visto che **un bug o una vulnerabilità in qualunque componente del kernel può danneggiare l'intero sistema** (con la famosa Blue Screen Of Death, BSOD, come risultato).

I device drivers, componenti del sistema operativo precedentemente menzionati, si occupano della gestione dei singoli dispositivi (attraverso le ISR), sono tipicamente moduli kernel caricati in memoria su domanda e rappresentano la maggior parte del sistema operativo (circa il 70% delle linee di codice di Linux) ed una delle principali cause di malfunzionamento.

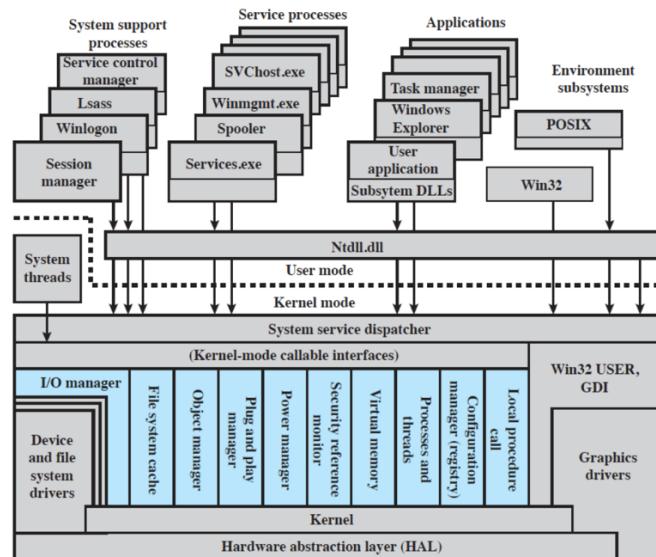


Un’ulteriore tipo di **architettura** dei SO è quella cosiddetta **a microkernel**, nel quale sono implementati solo i meccanismi essenziali (come la comunicazione tra processi) mentre le politiche di gestione sono specificate all'esterno del kernel, in **processi di sistema** (user mode):

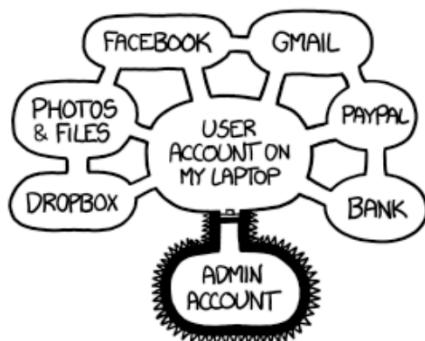


Un esempio di SO ad architettura a microkernel è MINIX 3; le risorse sono gestite da processi di sistema, detti server (file server, terminal server, printer server, ...) e quando un processo utente (client) deve usare una risorsa (come un file) invia una richiesta ad un server tramite meccanismi di comunicazione forniti dal microkernel stesso.

L'architettura di Windows è ibrida, non è puramente modulare né a microkernel e molti componenti non orientati alle performance (come gestione di sessioni, autenticazioni, servizi di sistema, ...) sono spostati nello user – space. Volendo visualizzarne la struttura:



Ad oggi, i sistemi operativi sono piuttosto sicuri, così tanto che nel caso in cui un laptop fosse rubato sarebbe più facile leggere le email personali, rubare denaro o l'identità piuttosto che installare driver senza il permesso del proprietario.



I PROCESSI E LA LORO GESTIONE

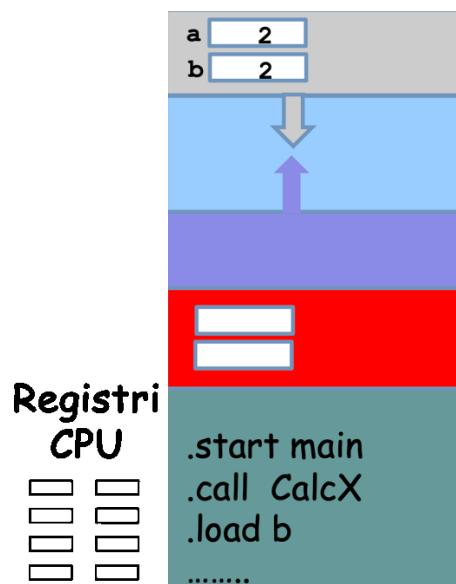
Un programma è definito come la codifica di un algoritmo in un linguaggio di programmazione, in modo da rendersi eseguibile da un calcolatore; questa definizione, classica e più volte utilizzata, è una descrizione statica delle elaborazioni da eseguire. Una migliore definizione si poggia sull'idea di processo, l'unità base di esecuzione di un Sistema Operativo che identifica le attività dell'elaboratore relative ad una specifica esecuzione di un programma; il processo, a differenza del programma, è un'entità dinamica, definita sia dal programma che dal suo contesto di esecuzione, e poi processi differenti possono eseguire più istanze di uno stesso programma.

La necessità di ridefinire il programma nel concetto di processo risiede nello pseudo-parallelismo che accompagna i Sistemi Operativi multiprogrammati; infatti, passando da un programma all'altro con rapida frequenza (non trattandosi di un parallelismo non è possibile eseguire contemporaneamente due operazioni), il Sistema Operativo deve salvare lo stato della macchina quando questa viene interrotta per far eseguire altre operazioni. Con questa struttura, è possibile affermare che ad ogni singolo processo è associata una CPU virtuale, contenente ogni registro del processore nella sua forma logica (e quando questa coincide con quella fisica, si può dire che il processo è in esecuzione). La parte del Sistema Operativo a cui è assegnato il compito di realizzare la virtualizzazione della CPU prende il nome di kernel (o nucleo).

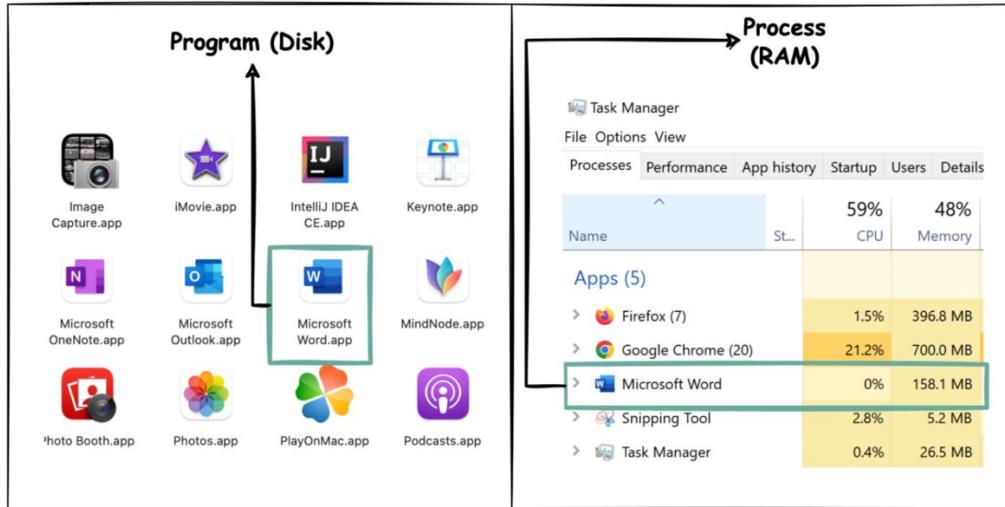
Se il programma assume una forma di questo tipo:

```
void Calc(int b) {  
    ...  
    if (b == 1) {...}  
    ...  
}  
  
int main() {  
    int a = 2;  
    Calc(a);  
}
```

Un processo è invece il contesto in cui è eseguito:



Con, in ordine dall'alto al basso, stack, heap, area dati, area codice. Nella pratica, questa distinzione è visualizzata ricorrendo alla differenza tra la libreria app e il gestore delle risorse:



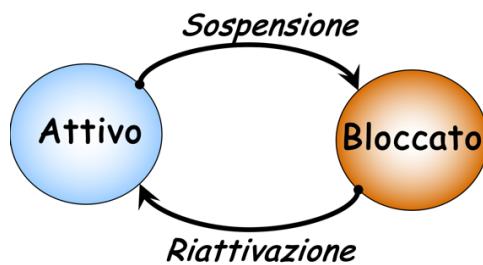
Lo stato di un processo rappresenta un'astrazione del suo contesto di esecuzione ed eventuali sue transizioni possono essere dovute a:

- L'attività corrente del processo;
- Eventi esterni asincroni con l'esecuzione del processo.

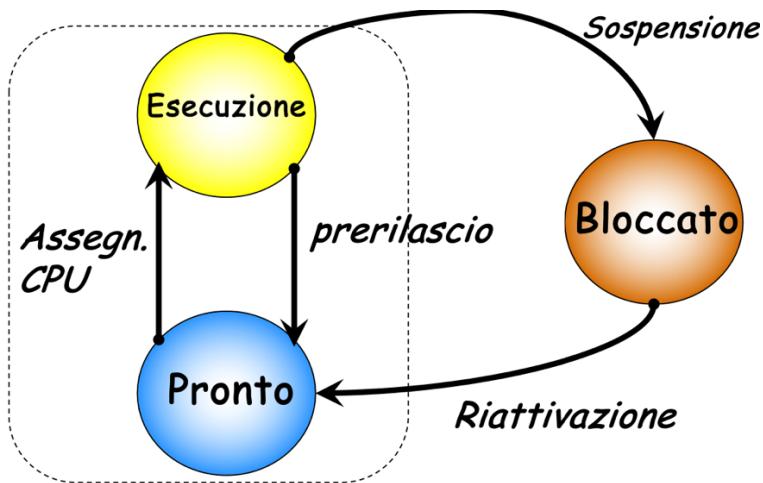
In prima approssimazione, un processo può trovarsi in uno solo fra due stati, il cui cambiamento è identificato con il nome di **transizione di stato**:

- **Attivo**, il processo è in esecuzione sulla CPU;
- **Bloccato**, il processo è in attesa di un evento (come un evento di I/O, un messaggio o una sincronizzazione).

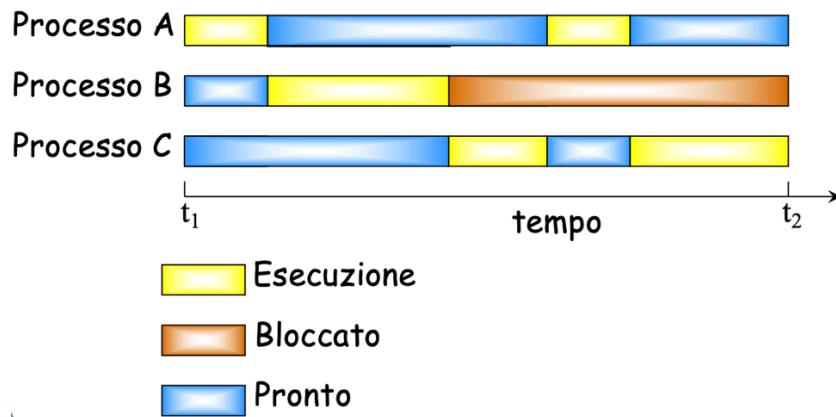
Il passaggio fra questi due stati avviene tramite le operazioni di sospensione e di riattivazione:



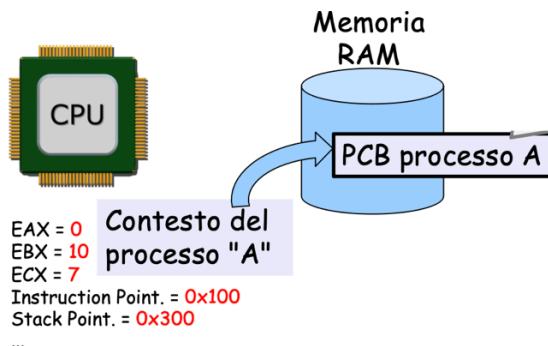
Il problema di questo modello, detto **modello a due stati**, è che presuppone l'esistenza di tante CPU quanti sono i processi di esecuzione; pertanto, sorge la domanda: come gestire il caso in cui si dispone di una sola CPU ma si sta lavorando con molti processi attivi o, generalmente, quando il numero di processi nel sistema supera il numero di unità di elaborazione? Risulterebbe necessario distinguere un processo pronto (ad essere eseguito) e un processo in esecuzione, e a quel punto si andrebbe ad usare il **modello a tre stati**, per il quale:



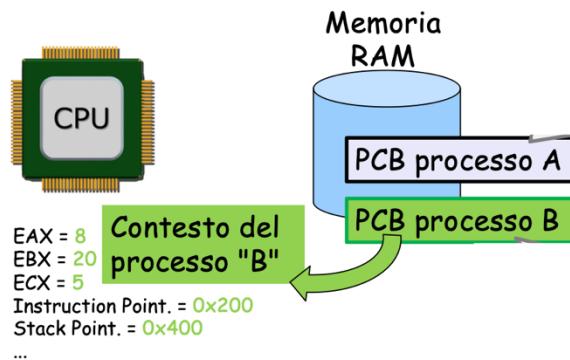
Un primo esempio concreto di processore che adotta questo modello può essere visualizzato dal seguente schema:



Per adottare questo modello è necessaria la comprensione del concetto di **contesto di un processo**, inteso come le **informazioni contenute nei registri del processore** (Program Counter, Stack Pointer, Registri general-purpose, Registri di gestione della memoria, ...), perché l'**insieme di operazioni**, descritte graficamente nell'immagine, **eseguite dal SO per il prerilascio di un processo** prendono il nome di **Context Switch** (o cambiamento di contesto). Durante un Context Switch il **Process Control Block (PCB)**, una struttura dati del SO che **contiene le informazioni sul contesto di un processo**, è salvato in memoria in attesa di essere ripristinato quando il processo viene ripreso dal processore:



Nel frattempo, il **PCB (Process Control Block)** di un secondo processo (già in memoria) viene **installato nei registri del processore** in modo da poter essere eseguito correttamente:



Descritto il modo in cui il cambio di contesto avviene, si elenchino i momenti in cui un Context Switch può effettivamente avvenire:

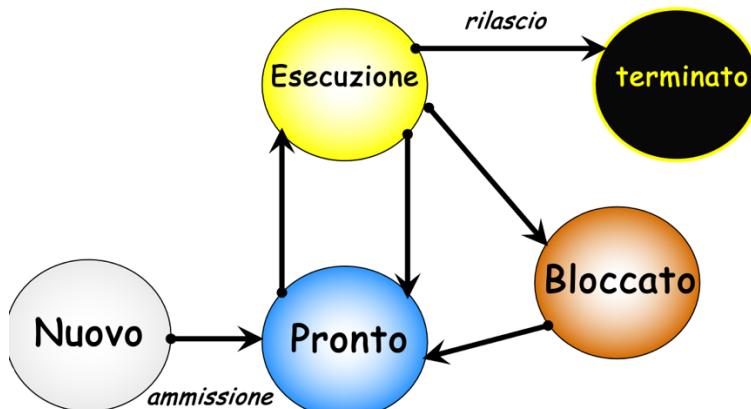
- **A seguito di un Timeout**, perché il tempo assegnato ad un processo può essere scaduto;
- **A seguito di una system call**, in cui il processo richiede un servizio al SO;
- **A seguito di interruzioni di I/O**;
- **A seguito di memory faults**, ovvero quando il processo accede ad un indirizzo di memoria non valido;
- **A seguito di una trap**, un'eccezione della CPU.

Riassumendo, il Context Switch è il risultato di tre operazioni:

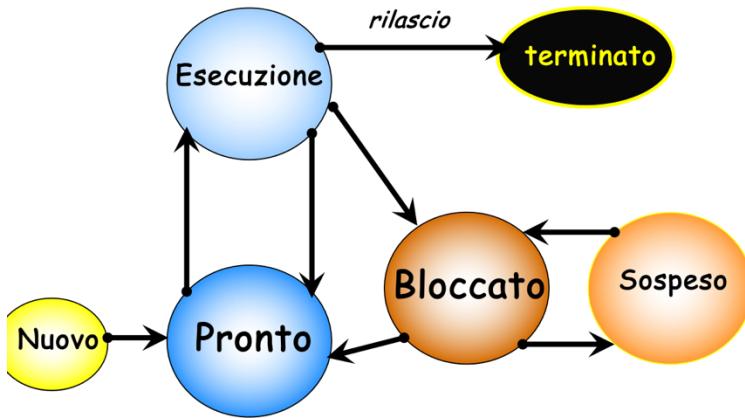
1. **Salvataggio dello stato**, si salva una copia del contesto del processo prelazionato nel PCB;
2. **Scheduling CPU**, si sceglie il prossimo processo (tra quelli pronti) da porre in esecuzione;
3. **Ripristino dello stato**, si copia il contesto del processo scelto dal suo PCB ai registri della CPU.

A questo punto, considerando anche la possibilità che più processi possano essere nello stesso momento nello stato pronto, si pone il problema di scegliere quale processo pronto vada messo in esecuzione; la scelta in questione è compiuta dallo **scheduler**, che fa parte generalmente del kernel del SO.

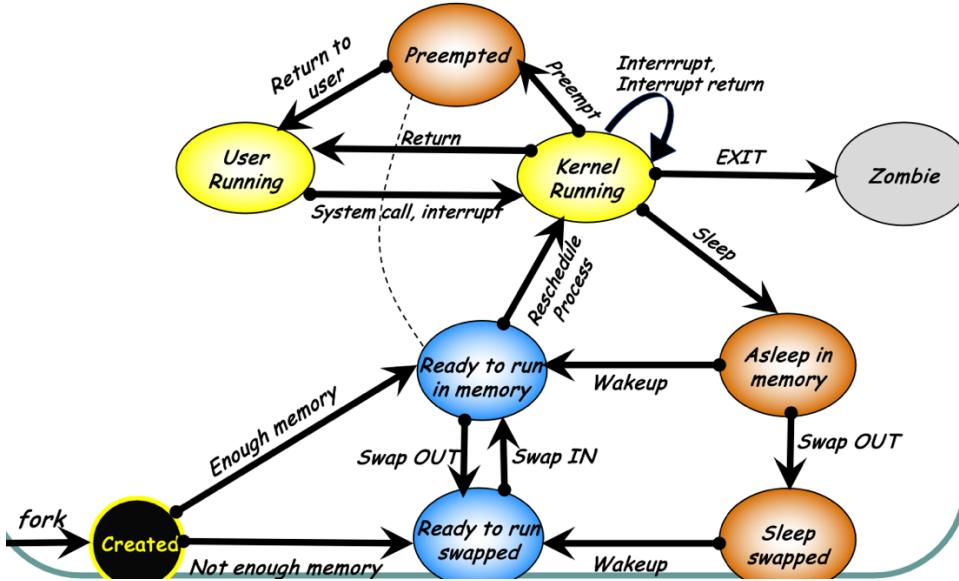
Il modello può essere ampliato con l'introduzione di due nuovi stati: lo stato “nuovo” corrisponde ad un **processo appena creato** e può avvenire (ad esempio) per la richiesta da parte di un processo già esistente, mentre lo stato “terminato” corrisponde ad una **terminazione del processo**, che può essere **normale** (il processo esegue una chiamata al SO per indicare il completamento delle proprie attività) o **anomala** (può essere provocata da un utilizzo scorretto delle risorse). Il **modello**, detto a 5 stati, può così essere visualizzato:



I SO prevedono anche la possibilità di spostare temporaneamente un processo dalla RAM alla memoria secondaria, operazione chiamata **swapping** e dedita alla liberazione dello spazio in memoria per altri processi; affinché lo swapping sia possibile, per alcuni sistemi è necessario implementare un ulteriore stato, detto stato “sospeso”, dando origine al modello generale:



Ovviamente quanto detto in questa sede è semplificato per ovvi motivi e lo schema degli stati di un processo può raggiungere livelli di complessità decisamente più importanti; come, ad esempio, lo schema degli stati di un processo UNIX:



Salta all'occhio la presenza di uno stato finora non menzionato, lo **stato “zombie”**. Esso è definito come **lo stato in cui si trova un processo terminato ma che non può essere ancora terminato perché la sua immagine in memoria è ancora necessaria**; ad esempio, un processo padre avvia un processo figlio per delegargli delle attività, il processo figlio termina prima del padre ma non può essere eliminato, perché quest'ultimo ancora deve arrivare al punto in cui le informazioni raccolte sono utilizzate, e quindi diventa zombie.

Con Linux è possibile usare il comando **top** per visualizzare il **task manager del computer**, mostrando periodicamente un elenco di informazioni sui processi che in un dato momento sono presenti nel sistema:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1149	so	20	0	3625080	304004	118296	S	1,7	14,9	0:20.99	gnome-shell
901	so	20	0	229372	60880	37780	S	0,3	3,0	0:04.74	Xorg
1371	so	20	0	292696	42436	31040	S	0,3	2,1	0:12.24	vmtoolsd
1939	so	20	0	816712	52104	39404	S	0,3	2,6	0:03.13	gnome-terminal-
3434	so	20	0	12000	3996	3212	R	0,3	0,2	0:00.14	top
1	root	20	0	102132	11604	8400	S	0,0	0,6	0:03.03	systemd
2	root	20	0	0	0	0	S	0,0	0,0	0:00.01	kthreadd
3	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	rcu_par_gp
6	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	kworker/0:0H-kblockd
7	root	20	0	0	0	0	I	0,0	0,0	0:10.44	kworker/0:1-events
9	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	mm_percpu_wq
10	root	20	0	0	0	0	S	0,0	0,0	0:00.24	ksoftirqd/0
11	root	20	0	0	0	0	I	0,0	0,0	0:00.65	rcu_sched
12	root	rt	0	0	0	0	S	0,0	0,0	0:00.06	migration/0
13	root	-51	0	0	0	0	S	0,0	0,0	0:00.00	idle_inject/0
14	root	20	0	0	0	0	S	0,0	0,0	0:00.00	cpuhp/0
15	root	20	0	0	0	0	S	0,0	0,0	0:00.00	kdevtmpfs
16	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	netns
17	root	20	0	0	0	0	S	0,0	0,0	0:00.00	rcu_tasks_kthre

R (per running) indica i processi “pronti” e in “esecuzione”, mentre **S** (per sleeping) indica quelli in attesa di I/O.

Si può anche utilizzare il comando **ps aux** per stampare un’istantanea dei processi e le loro informazioni:

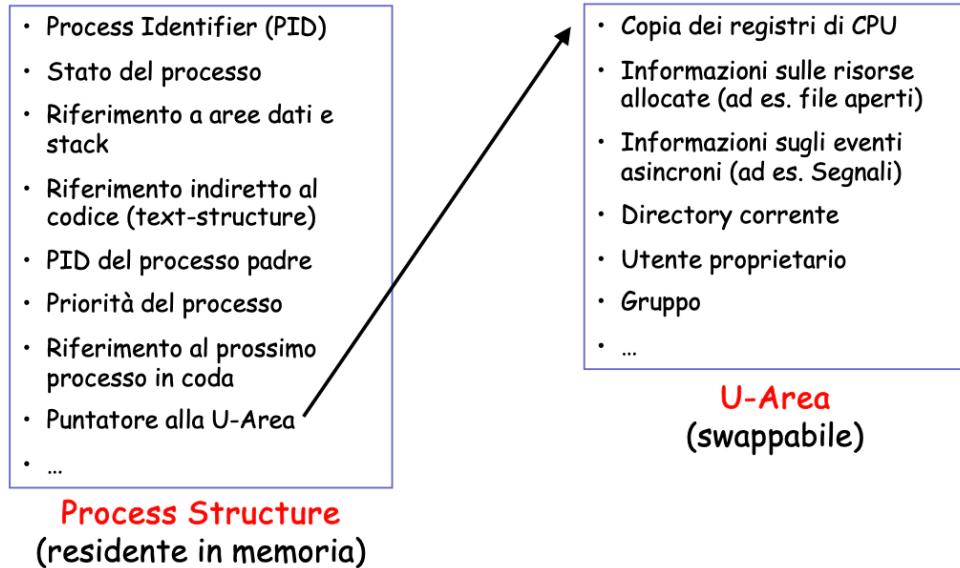
- **Identificatore (PID);**
- **Stato;**
- **Il terminale assegnato;**
- **Il tempo di CPU utilizzato;**
- **Il nome del comando** corrispondente al processo.

Di default sono mostrati solo i processi che sono “visibili” all’utente, ovvero che usano un terminale, ma con l’opzione **x** possono essere mostrati anche quelli senza terminale (processi in background o “daemon”).

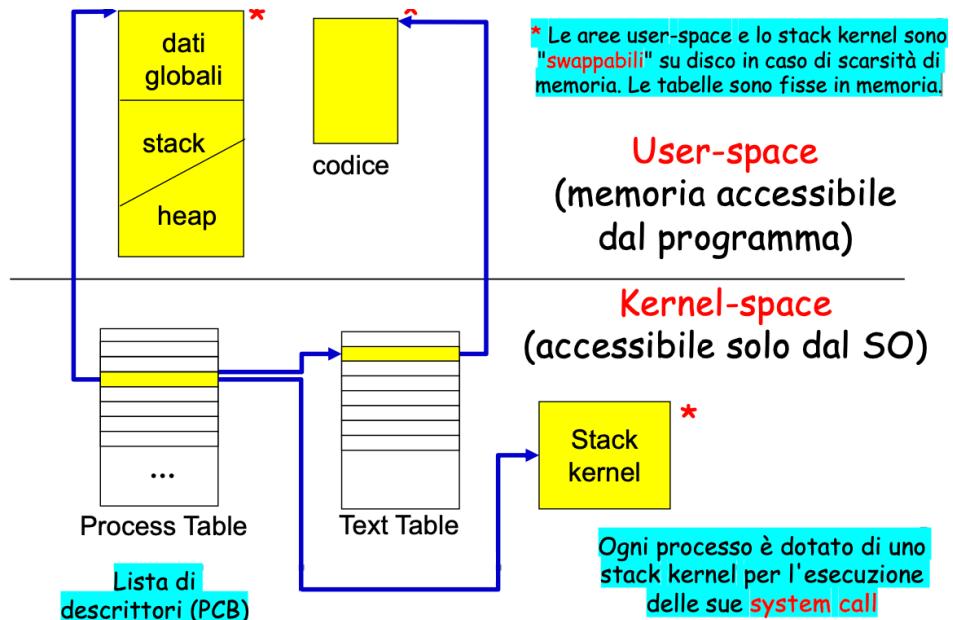
Ad ogni processo, come è stato già introdotto, è associata una struttura dati (come una struct in linguaggio C), detto **Descrittore del Processo** (o PCB come precedentemente annotato) e sono raggruppati in una apposita tabella dei processi (Process Table). Le informazioni da registrare nel descrittore possono essere così classificate:

- **Nome del processo;**
- **Stato del processo;**
- **Modalità di servizio** (ad esempio, priorità o deadline);
- **Informazioni sulla gestione della memoria;**
- **Contesto del processo** (copia dei registri della CPU);
- **Utilizzo delle risorse;**
- **Identificatore del processo successivo.**

In UNIX (U – area corrisponde allo user space):



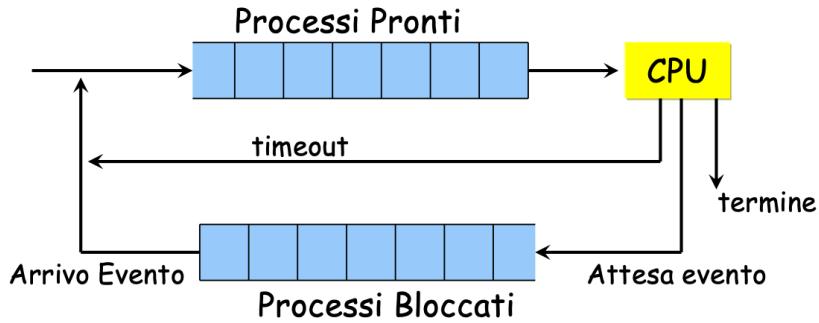
L'immagine di un processo, invece, si configura come segue:



Il SO tiene traccia dei processi nel sistema utilizzando delle code:

- Coda (una o più) dei processi pronti (o runqueue);
- Coda (una o più) dei processi bloccati, ovvero quelli in attesa di eventi.

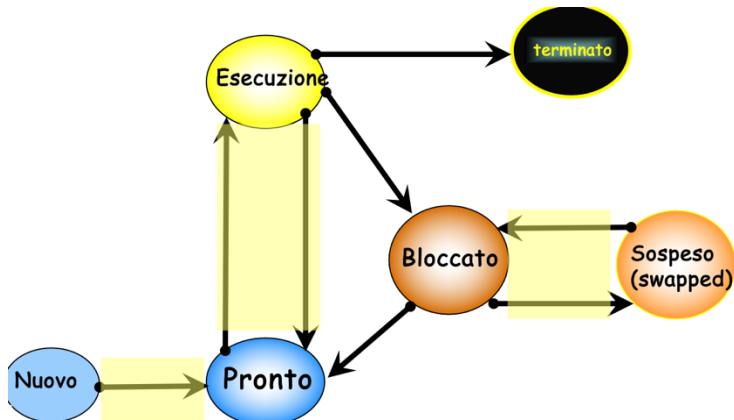
Il tutto si configura come segue:



Alla **creazione di un nuovo processo**, esso viene **posizionato in coda tra i processi pronti**; quando arriva il momento di essere rimosso dalla coda, il **prerilascio** da parte del SO forza il processo in esecuzione nello **stato di pronto**. Eventuali operazioni di I/O pongono il processo nello **stato bloccato** e lo fanno **accodare agli altri processi bloccati**; infine, in seguito al verificarsi dell'evento atteso, il processo bloccato transita nello **stato di pronto** e si comporta come se fosse stato appena creato. A partire da quanto detto, la domanda che sorge è: **un processo sbloccato va messo in testa o in coda tra i processi pronti?** La risposta viene fornita dallo **scheduler** e dal paradigma con cui è implementato nel SO.

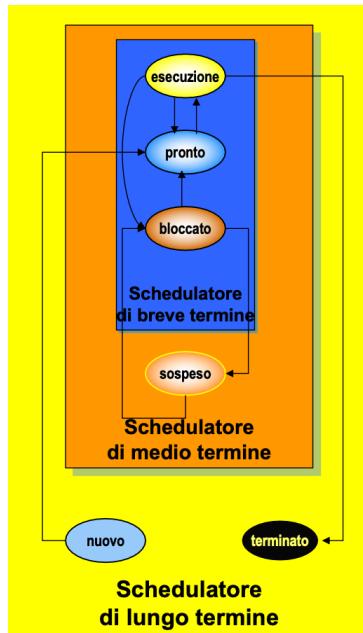
LO SCHEDULER E LO SCHEDULING

Lo **scheduler** è la parte del SO preposta all'**assegnazione di risorse a favore dei processi**, mentre un **algoritmo di scheduling** seleziona il **processo assegnato da una coda sulla base dei criteri** in esso specificati. I momenti in cui uno scheduler lavora sono quelli evidenziati nella figura seguente, prendendo come esempio il modello generale:



Esistono **diverse tipologie di scheduler**:

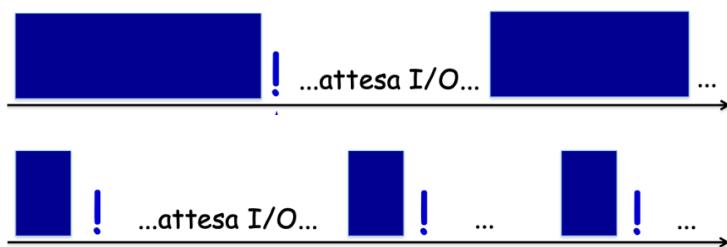
- Scheduler a **breve termine** (o della CPU);
- Scheduler a **medio termine** (o di swap, o swapper);
- Scheduler a **lungo termine** (o di job).



Lo scheduler a lungo termine determina quale programma caricare nella coda dei processi pronti del sistema con lo scopo di regolare la quantità di processi del sistema, agendo così nel grado di multiprogrammazione, tra uno stato nuovo ed uno pronto. I possibili criteri per la costruzione di un algoritmo di scheduling sono i seguenti:

- FIFO (First In First Out);
- Priorità;
- Tempo di esecuzione presunto;
- Requisiti di I/O;
- Tempo presunto di CPU.

Ad esempio, per aumentare l'efficienza d'utilizzo delle risorse, uno scheduler di lungo termine può ammettere nel sistema un numero comparabile di processi CPU – bound e I/O – bound. Per **processi CPU – bound** si intendono quei processi che effettuano poche chiamate di sistema e che tendono ad occupare la CPU per lunghi periodi (se il SO non li interrompe), sono tendenzialmente applicazioni di batch e calcolo numerico; i processi **I/O – bound**, invece, sono quelli che fanno un frequente uso di chiamate di sistema, usando poco la CPU per poi mettersi subito in attesa di I/O, sono tendenzialmente i programmi interattivi come browser o editor di testo. La differenza tra questi due tipi di processi può essere visualizzata come segue:



Le porzioni piene di grafico corrispondono agli intervalli di tempo in cui il processo utilizza le risorse della CPU.

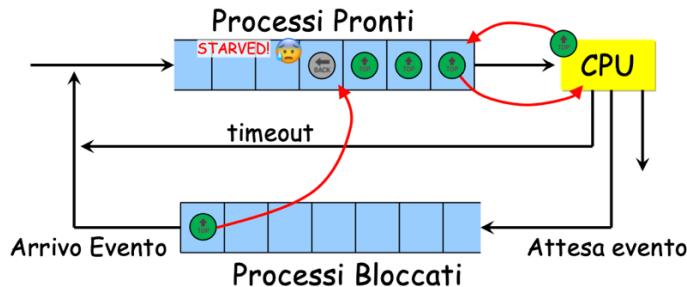
Lo scheduler di medio termine ha il compito di trasferire temporaneamente processi (o parte di essi) dalla memoria centrale alla memoria di massa (swap – out) e/o viceversa (swap – in), con lo scopo di liberare spazio nella memoria centrale e di rendere possibile il caricamento di altri

processi, ovvero di **gestire efficientemente la memoria**, agendo tra i processi bloccati e quelli sospesi.

Lo **scheduler** di breve termine ha il compito di **scegliere un processo pronto a cui assegnare la CPU** ed è quello **attivato più frequentemente**, quando il processo in esecuzione si interrompe (ad esempio, per un'interruzione del clock, una system call, ...), agendo **tra processi in esecuzione e pronti**. Quindi, come si può intuire, **questo tipo di scheduler mira ad ottimizzare dei criteri di prestazione**, quali:

- **Criteri user – oriented**, per i quali la percezione dell'utente e dei singoli processi ne fa da padrone e definiscono:
 - **Tempo di risposta**, il tempo tra l'invio di una richiesta dell'utente e l'inizio della sua elaborazione;
 - **Tempo di turnaround**, il tempo tra l'invio di un processo al sistema e la sua terminazione (include il tempo di risposta, il tempo speso in esecuzione e il tempo speso in attesa);
 - **Deadlines**, una scadenza di completamento indicata dall'utente e per la quale va massimizzata la percentuale di scadenze rispettate.
- **Criteri system – oriented**, per i quali il punto di vista delle risorse e dello stato amministratore del sistema è al primo posto e definiscono:
 - **Throughput**, la produttività in termini di numero di processi completati per unità di tempo (come 10 esecuzioni/ora);
 - **Utilizzo della CPU**, come la percentuale di tempo in cui la CPU risulta occupata;
 - **Fairness**, il criterio per cui (a meno di indicazioni esplicite per l'utente) tutti i processi vanno trattati allo stesso modo, senza forzarne alcuno in condizione di attesa indefinita (starvation).

Un **processo** si definisce **in starvation** quando può attendere per un tempo arbitrariamente lungo ed è una condizione detta anche “attesa infinita”; la starvation equivale all’assenza di garanzie di esecuzione del processo nel prossimo futuro (non va confusa con attesa infinita) e può essere visualizzata come segue:

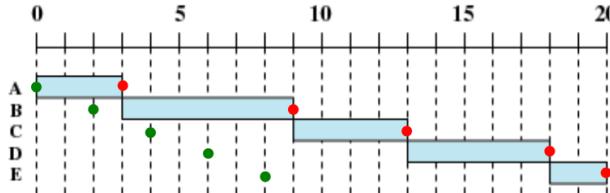


L'evento che viene ripetutamente spinto indietro nella coda dei processi pronti soffre di **starvation**. Non tutti gli algoritmi di scheduling soffrono di starvation, generalmente solo quelli in cui viene data maggiore priorità a specifici processi; infatti, si tende a scegliere sempre i processi a priorità più “alta” per l'esecuzione, lasciando gli altri in attesa. Ovviamente, non esiste l'algoritmo perfetto anzi, spesso i vari criteri sono interdipendenti o in contrasto tra di loro, rendendo necessario un tradeoff accuratamente pensato sull'utilizzo che dovrà essere fatto del sistema.

Prima di analizzare nel dettaglio gli algoritmi di scheduling, **si vogliono introdurre le seguenti definizioni**:

- **Tempo di servizio (T_s)**, è il tempo in cui un processo è in esecuzione;

- **Tempo di Turnaround (Tr)**, è il tempo che intercorre tra la creazione di un processo e la sua terminazione;
- **Indicatore di ritardo relativo (Tr/Ts)**, descrive il tempo di attesa tra la creazione e l'esecuzione di un processo (se Ts è alto, ad esempio, l'utente può tollerare un ritardo più alto) e idealmente si cerca di portarlo quanto più vicino possibile all'unità.



Con:

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

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

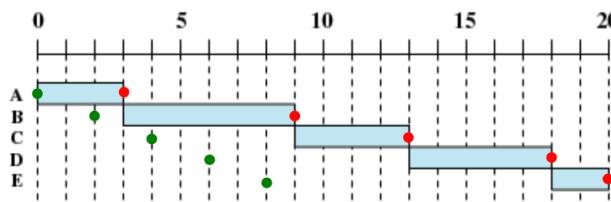
$TrD/TsD=2.40, TrE/TsE=6.00$

Chiaramente, gli algoritmi di scheduling sono volti ad evitare situazioni come quella del processo E, per il quale il tempo di turnaround è nettamente maggiore di quello di servizio, indicando che il processo spende nel sistema un tempo pari a 6 volte quello di esecuzione (quindi, come se per scrivere un carattere su un editor di testo ci si mettessero 6 secondi).

Gli algoritmi di scheduling a breve termine sono i seguenti:

- **Non – preemptive**, sono quelli per cui un processo in esecuzione rimane in tale stato finché non si sospenderà automaticamente:
 - First – Come First – Serve

Questo algoritmo segue una regola molto simile a quella delle code: **quando un processo termina, viene eseguito il primo processo in ordine di arrivo**. Graficamente, questo algoritmo può essere rappresentato come segue:



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

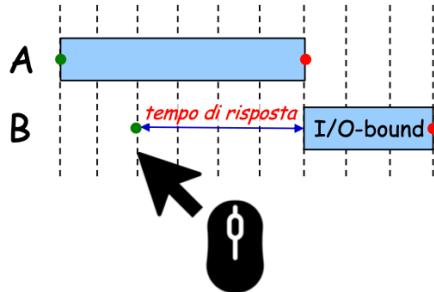
Tr/Ts $TrA/TsA=1, TrB/TsB=1.17, TrC/TsC=2.25$

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

Nonostante ci sia un **alto ritardo relativo** (innescando il cosiddetto effetto convoglio), il sistema non risente di starvation, quindi prima o poi un processo creato sarà sempre eseguito.

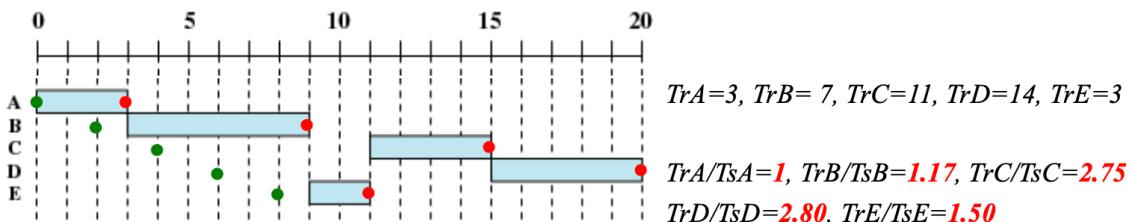
L'**effetto convoglio** è definito come la condizione per cui un processo CPU – bound occupa la CPU per un tempo eccessivo, a discapito dei processi I/O – bound (come il processo E). Tale effetto causa una riduzione dell'utilizzo delle risorse, che esse siano CPU o I/O, ritardando e fermando i processi I/O – bound e, come conseguenza, lasciando le periferiche inutilizzate; nel momento in cui i processi in questione vengono, poi, eseguiti, la CPU risulta sotto – utilizzata e le risorse sprecate. L'attenzione nei confronti dei processi I/O – bound è necessaria perché sono quelli che scandiscono all'utente la velocità del sistema: se un processo I/O – bound (come la scrittura di un

testo) ha un tempo di risposta lungo, l'utente percepisce un rallentamento del sistema (la scrittura non è rapida e fluida).



- **Shortest Process Next**

L'algoritmo prevede l'esecuzione dei processi sulla base del minor tempo di esecuzione (stimato) e, sebbene riduca i tempi di risposta per i processi brevi, per i processi lunghi c'è un alto rischio di starvation.



Per adottare questo algoritmo è necessario poter stimare il tempo di esecuzione di ciascun processo, tendenzialmente la media dei periodi di esecuzione passati:

$$S_{n+1} = \frac{1}{n} \sum_{i=1}^n T_i$$

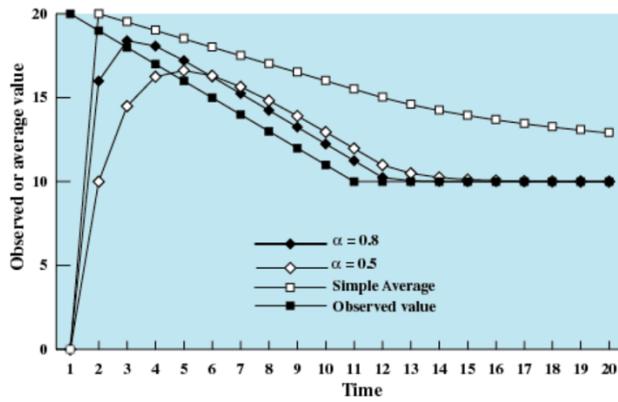
Ma ciò non significa che vanno registrati in memoria i tempi di esecuzione di ogni singolo processo per ogni singola volta che vengono eseguiti (la memoria sarebbe ben presto saturata); infatti, per un processo è sufficiente conoscere il tempo di esecuzione della n – esima esecuzione e la media fino all'iterazione n – esima:

$$S_{n+1} = \frac{1}{n} \sum_{i=1}^n T_i = \frac{1}{n} T_n + \frac{n-1}{n} S_n$$

Un altro approccio potrebbe preferire una media pesata sulla base delle esecuzioni più recenti, dando più peso a queste in quanto potenzialmente più rappresentative del comportamento da prevedere; un esempio di stima media pesata è quella esponenziale:

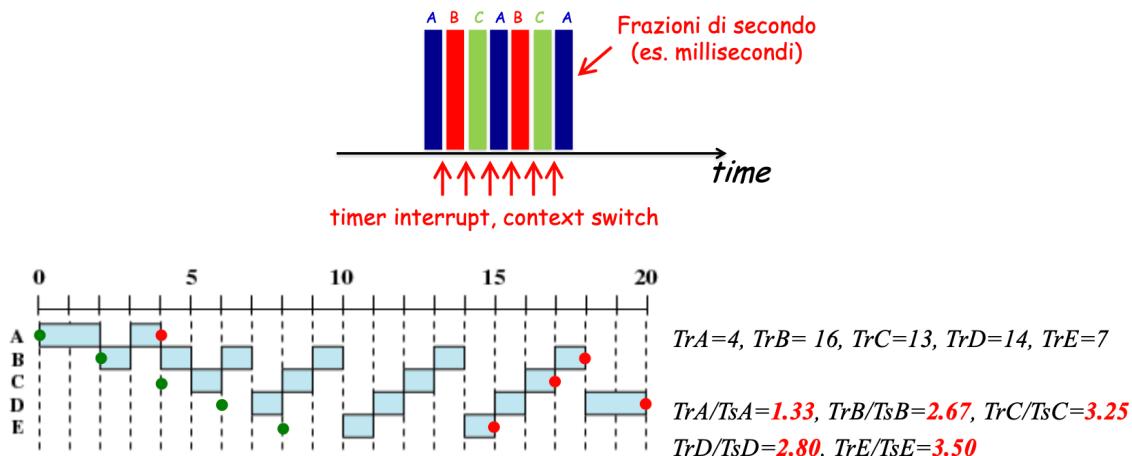
$$S_{n+1} = \alpha T_n + (1 - \alpha) S_n, \forall \alpha \in (0,1)$$

Nel tempo la differenza tra la stima e l'effettivo tempo di esecuzione di un processo può essere visualizzata dal seguente grafico:



- Preemptive, sono quelli per cui un processo in esecuzione può essere interrotto anche se non effettua una system call o se viene forzato nello stato pronto:
 - Round – Robin

L'algoritmo Round – Robin si configura come la **versione preemptive del First – Come First – Serve**, mediante l'utilizzo di un timer: viene assegnato ad un processo un tempo massimo di esecuzione, detto "time slice" o "quanto di tempo" (spesso frazioni di millisecondi) superato il quale il processo viene interrotto e viene eseguito quello successivo (secondo la stessa regola del FCFS).

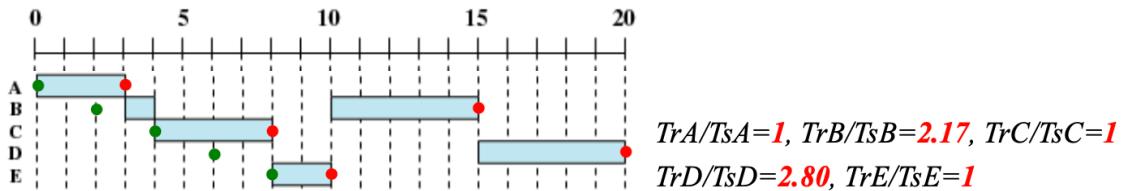


Si noti che, a differenza del First – Come First – Serve, il processo E ha (quasi) dimezzato il suo ritardo relativo.

Si può ben intuire come sia molto importante la scelta del time slice, da cui dipendono inevitabilmente le prestazioni del sistema: aumentando il time slice, l'algoritmo tende a trasformarsi nel First – Come First – Serve, mentre, diminuendolo, si aumenta la frequenza dei context switch e, quindi, l'overhead del SO. Ad esempio, se un quanto di tempo dura 10ms e un context switch 1ms (caso estremo, nelle moderne CPU dura 5μs), viene sprecato circa il 10% del tempo di CPU per la gestione del SO, tempo che potrebbe essere impiegato per del lavoro utile. Idealmente, il time slice dovrebbe essere appena superiore alla durata dei processi I/O – bound (quindi dai 10ms ai 100ms), in modo da garantire che non vengano interrotti (e che l'utente percepisca il rallentamento), che siano attivati i dispositivi e che la CPU sia liberata.

- Shortest Remaining Time

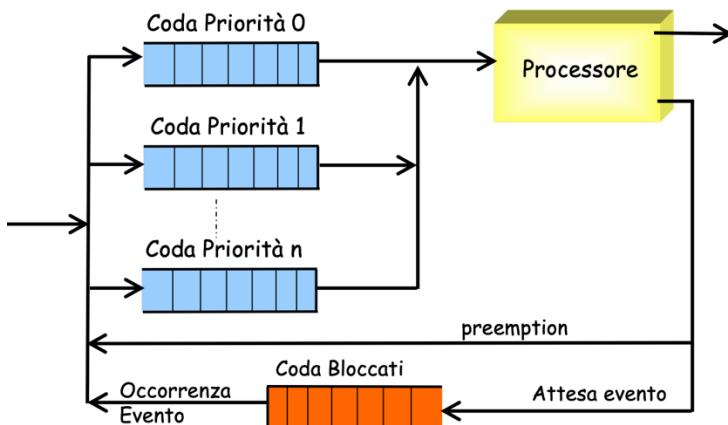
È la versione preemptiva dell'algoritmo Shortest Process Next e prevede che la prelazione (interruzione dovuta al fatto che il processo sta usando troppe risorse per troppo tempo) possa avvenire subito quando entra un nuovo processo; in altri termini, i processi brevi non aspettano mai:



Rispetto al caso non – preemptive, il processo C prelaziona il processo B.

- Multilevel Feedback

Nella pratica, sia il Shortest Process Next che il Shortest Remaining Time sono algoritmi di difficile utilizzo, dal momento in cui la stima precisa è abbastanza complessa da fare. Uno scheduler a priorità (multilevel) semplifica il tutto raggruppando i processi in livelli di priorità e gestendo i processi allo stesso livello con un algoritmo Round – Robin. Ogni gruppo di processi è gestito con una coda separata, in modo che la coda $n -$ esima abbia precedenza sulla coda $n + 1 -$ esima:



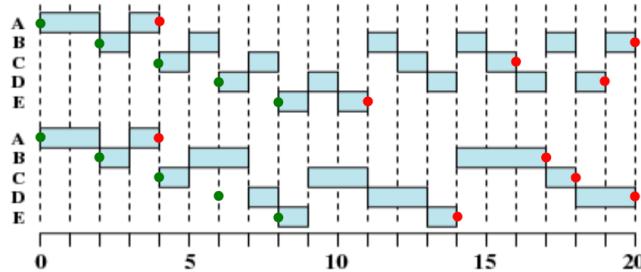
I processi a bassa priorità possono essere eseguiti quando si esauriscono i processi nella coda a massima priorità, seguendo uno scorrimento automatico di priorità. Per non penalizzare gli ultimi livelli di priorità, si utilizza un meccanismo di “feedback”: si assegna ai processi una priorità dinamica, penalizzando i processi CPU – bound a favore di quelli I/O – bound; nella pratica, i processi che esauriscono il loro quanto di tempo sono sospesi e degradati di priorità. Infine, l’ultima coda (quella a priorità più bassa), invece che essere gestita con un algoritmo Round – Robin, è gestita da uno scheduler First – Come First – Serve (essendo composta tendenzialmente da processi di background).

L’approccio Multilevel Feedback favorisce i processi più corti, sebbene occorra minimizzare la starvation per i processi più lunghi; per poter compensare, i processi a minor priorità sono caratterizzati da un quanto di tempo più lungo, sulla base del livello di priorità ad essi assegnato:

$$\text{coda } i - \text{esima} \Rightarrow 2^i \text{ unità di tempo}$$

Dopo un tempo massimo, si fa ritornare un processo dalla coda inferiore alla coda ad alta priorità (si evita, così, il fenomeno di aging per cui un processo è per troppo tempo nella stessa coda, precursore della starvation).

Confrontando uno scheduler a Multilevel Feedback con quanto di tempo costante (1) e uno variabile (2^i):



Riassumendo:

	Funzione di selezione	Modalità di decisione	Throughput	Tempo di risposta	Overhead	Impatto sui processi	Attesa indefinita
FCFS	Max[w]	Non interrompente	Non enfatizzato	Potrebbe essere alto, nel caso in cui il tempo di esecuzione abbia una varianza elevata	Minimo	Penalizza i processi brevi ed I/O bound	No
Round Robin	Costante	Interrompente (quanto di tempo)	Può essere basso se il quanto di tempo è eccessivamente piccolo	Tempi di risposta buoni per processi brevi	Minimo	Politica fair	No
SPN	Min [s]	Non interrompente	Elevato	Tempi di risposta buoni per processi brevi	Può essere elevato	Penalizza i processi lunghi	Possibile
SRT	Min [s-e]	Interrompente (istante di arrivo)	Elevato	Buono	Può essere elevato	Penalizza i processi lunghi	Possibile
Feedback	Con retroazione	Interrompente (quanto di tempo)	Non enfatizzato	Non enfatizzato	Può essere elevato	Potrebbe favorire i processi I/O bound	Possibile
	Processo	A	B	C	D	E	Media
	Tempo di Arrivo	0	2	4	6	8	
	Tempo di Servizio (Ts)	3	6	4	5	2	
FCFS	T. di Completamento T.di Turnaround (Tr) Tr/Ts	3 3 1	9 7 1.17	13 9 2.25	18 12 2.40	20 12 6	8.60 2.56
RR q=1	T. di Completamento T.di Turnaround (Tr) Tr/Ts	4 4 1.33	18 16 2.67	17 13 3.25	20 14 2.80	15 7 3.50	10.80 2.71
RR q=4	T. di Completamento T. di Turnaround (Tr) Tr/Ts	3 3 1	17 15 2.5	11 7 1.75	20 14 2.80	19 11 5.50	10 2.71
SPN	T. di Completamento T. di Turnaround (Tr) Tr/Ts	3 3 1	9 7 1.17	15 11 2.75	20 14 2.80	11 3 1.50	7.60 1.84
SRT	T. di Completamento T. di Turnaround (Tr) Tr/Ts	3 3 1	15 13 2.17	8 4 1	20 14 2.80	10 2 1	7.20 1.59
FB q=1	T. di Completamento T. di Turnaround (Tr) Tr/Ts	4 4 1.33	20 18 3	16 12 3	19 13 2.60	11 3 1.5	10 2.29
FB q=2 ⁱ	T. di Completamento T. di Turnaround (Tr) Tr/Ts	4 4 1.33	17 15 2.50	18 14 3.50	20 14 2.80	14 6 3	10.60 2.63

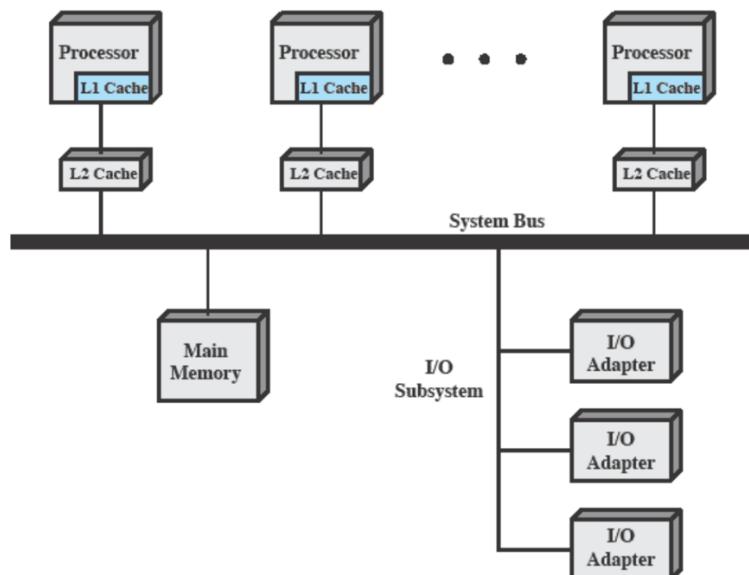
Con w tempo di attesa, e tempo di esecuzione e s tempo di servizio (incluso e).

TECNOLOGIA MULTI – PROCESSORE

Fino a questo momento, il “calcolatore” su cui è stata sviluppata la trattazione è sempre stato composto da una sola unità di elaborazione, una sola CPU. Nella pratica, i moderni dispositivi sono realizzati adottando diverse tecnologie multi – processore, con lo scopo di massimizzare la capacità calcolo computazionale; tra queste tecnologie si distinguono:

- **Symmetric Multi – Processing (SMP)**, è realizzato collegando più CPU (una per chip) ad una stessa memoria centrale condivisa e rendendo ognuna di esse accessibile agli stessi dispositivi di I/O;
- **Multi – core CPU**, realizzato costruendo più CPU (“core”) sono sullo stesso chip;
- **Hyperthreading**, realizzato in modo che uno stesso core abbia risorse multiple (ALU, registri, ...) e possa eseguire più operazioni contemporaneamente.

Un’architettura SMP si struttura come segue:



Mentre un’architettura multicore:

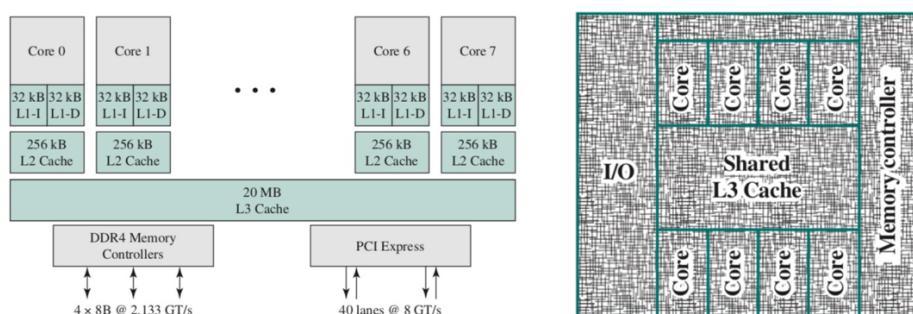
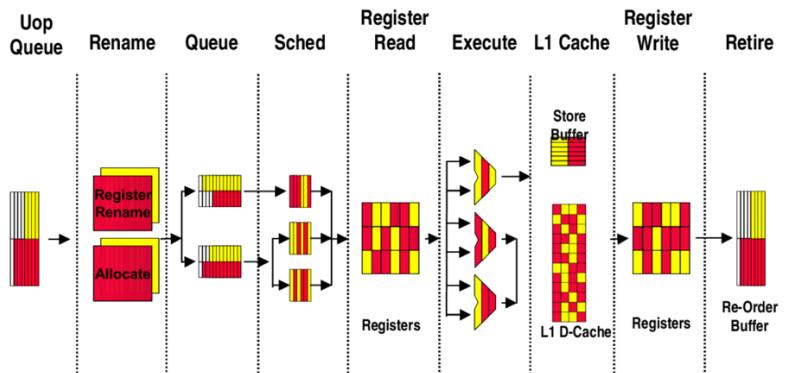


Diagramma a blocchi

Layout sul chip fisico

Ed un’architettura hyperthreading:

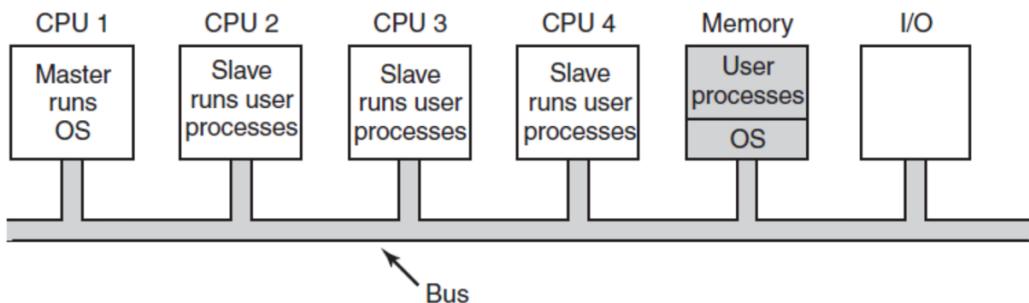


Per quanto riguarda queste ultime, **il SO “vede” diversi core “virtuali”, ognuno dei quali esegue un programma diverso** (come una vera e propria CPU) ma in realtà appartiene fisicamente allo stesso core, il quale condivide le componenti hardware (come l’ALU) tra i vari core virtuali.

La progettazione di un SO ha due margini di libertà dal punto di vista del design del sistema:

- Esecuzione del meccanismo di assegnazione:
 - Approccio Master – Slave

Secondo questo approccio, **il kernel esegue su un solo processore (master) responsabile dello scheduling, mentre gli altri (slave) eseguono solo processi utente**; quando uno slave incontra una system call, inoltra l’intero processo al master.



Il vantaggio di questo approccio è la sua semplice implementazione, ma si porta dietro due principali problemi: **il master rappresenta sia un enorme collo di bottiglia per le prestazioni che un single point of failure**.

- Approccio Peer

Il kernel può eseguire su tutti i processori, anche contemporaneamente, ed ogni processore gestisce lo scheduling autonomamente. L’approccio in questione è di più difficile implementazione, dal momento in cui necessita della sincronizzazione dei processori nell’accesso alle risorse comuni (come la coda dei processi pronti) ma risolve le principali falle dell’approccio precedente.

- Assegnazione dei processi ai processori:
 - Assegnazione statica

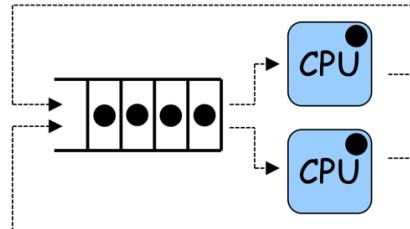
Ogni processo o thread è assegnato permanentemente ad uno dei processori, in modo che ognuno di questi abbia una propria coda dei processi pronti. Chiaramente, questo è un approccio

volto alla semplicità, l'assegnazione è fatta una volta e per sempre, ma il carico potrebbe non essere uniforme tra i processori (dipende dal comportamento dei processi).

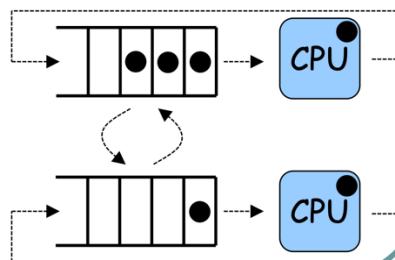
- **Assegnazione dinamica**

Durante la sua vita, un processo può eseguire su processori differenti; si distinguono poi, **due implementazioni della runqueue**:

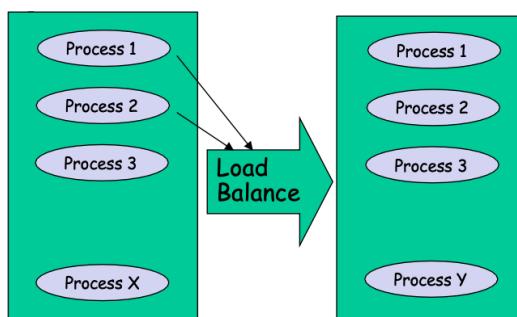
- **Load sharing**, una sola coda dei processi pronti condivisa;



- **Dynamic Load Balancing**, più code (una per processore) con la possibilità che un processo sia spostato tra di esse.



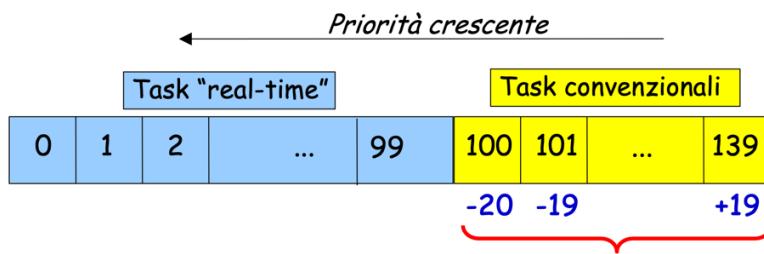
Idealemente, avere una sola coda garantisce l'utilizzo massimale della CPU; con code separate, infatti, c'è il rischio (anche se piccolo) che una di esse si svuoti, rendendo una CPU inutilizzata. Tuttavia, avere una sola coda può creare due problemi: in primis la coda occupa una regione di memoria condivisa, che deve essere protetta da accessi concorrenti (mutua esclusione), ma non garantisce neanche che un processo riprenda l'esecuzione sullo stesso processore (uso poco efficiente delle cache dei processori); in SMP, invece, le cache/TLB sono locali alle CPU/core. Con code multiple ogni CPU accede ad una coda separata e i processi riutilizzano la stessa CPU (a meno di load balancing); a questo proposito, si definisce la CPU affinity come la tendenza di un processo ad eseguire più velocemente se viene riutilizzato lo stesso CPU/core. Spostare un processo/thread da un processore all'altro, inoltre, danneggia il principio di località, mentre il load balancing comporta un impatto negativo sui processi migrati; quindi, è necessario migrare i processi con minore affinità, in modo da ottenere un minor tempo speso in esecuzione sulla CPU e un minor consumo di cache.



Linux adotta il **dynamic load balancing**, che viene attivato periodicamente o quando una runqueue è vuota, e vengono estratti dalla lista i task che non stanno eseguendo e che non sono cache – hot e l'algoritmo termina quando la runqueue con il maggior numero di task non eccede del 25% le altre (nell'immagine $X = 20$ e $Y = 15$).

SCEDULING IN SISTEMI OPERATIVI UNIX/LINUX E WINDOWS

In Linux l'**unità fondamentale dello scheduler** è il **task**, inteso come un **flusso di esecuzione**, e viene utilizzato per rappresentare sia i **processi** che i **threads**, mentre la sua identificazione avviene tramite **Process ID** (PID). Ad ogni task è attribuita una **priorità**, che ne determina l'ordine di scheduling e il quanto di tempo assegnato; poi, sulla base della priorità, si distinguono due tipologie di task: “**real time**” e **convenzionali**.



I programmi utente sono solitamente **task convenzionali**, la cui priorità è rappresentata da un **numero intero compreso tra 0 e 39** a cui l'utente può sommare un **valore di correzione** (compreso tra -20 e $+19$) chiamato **nice value**.

In un SO Linux sono adottati principalmente **due algoritmi di scheduling**:

- **Scheduler $O(1)$**

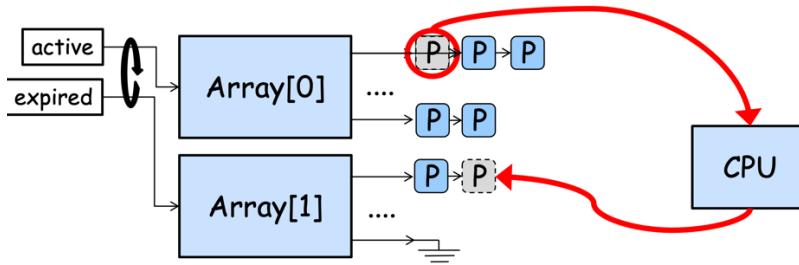
È stato introdotto con il kernel Linux 2.6 ed è basato sul **modello classico del multilevel feedback** usato anche in Unix. Gli obiettivi che questo algoritmo si pone sono:

- Ottenere uno **scheduling con overhead costante all'aumentare dei processi** nel sistema;
- Ottenere un **buon compromesso tra tempo di risposta ed equità**;
- Utilizzare **efficientemente le architetture SMP**.

Il nome dell'algoritmo è stato dato sulla base di una delle sue proprietà; infatti, **il tempo impiegato per scegliere quale processo eseguire è costante ed indipendente dal numero di task nel sistema**. Ad ogni processore è associata **una runqueue** (la struttura dati contenente le code di processi in attesa) e per attenuare il fenomeno della **starvation** sono introdotti **due gruppi di code**:

- **Active**, contenente i task che non hanno ancora esaurito il quanto di tempo assegnato;
- **Expired**, contenente i task che hanno esaurito il quanto di tempo assegnato.

Un task in una delle code active che ha esaurito il proprio quanto di tempo viene sempre spostato nella omologa coda del gruppo expired, per non alterare i livelli dei processi. Nel momento in cui tutte i task sono passate dalla coda active a quella expired, le due si invertono (**mantenendo l'ordine dei livelli**) e inizia un nuovo round di esecuzione.



Si noti che **tutte le operazioni** (come selezione di un processo, ri – accodamento, ...), essendo basate su linked lists, **hanno una durata fissa**.

Internamente, per un task, lo scheduler distingue tra priorità statica e priorità dinamica, la prima coincide con il nice value e determina la durata del quanto di tempo assegnato ad un task, mentre la seconda, inizialmente pari al nice value, varia durante l'esecuzione e determina la coda (runqueue) in cui è inserito il task associato (quindi, l'ordine di esecuzione).

Si tenga in considerazione che **il quanto di tempo** (o timeslice) è **proporzionale alla priorità statica del task**:

Tipo di task	Nice value	Timeslice
Creazione	processo padre	metà del padre
Priorità minima	+19	5 ms
Priorità di default	0	100 ms
Priorità massima	-20	800 ms

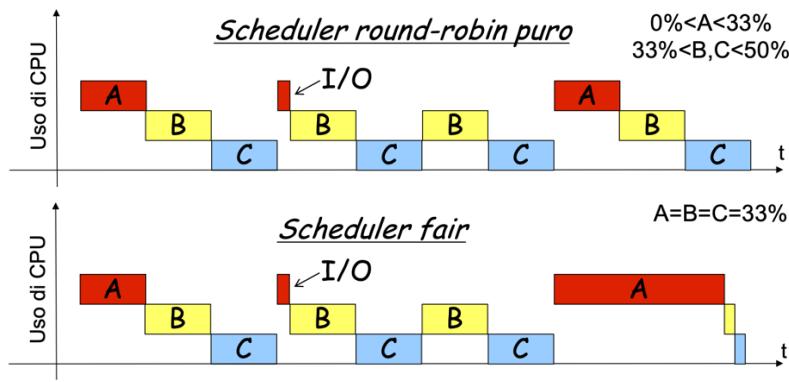
Mentre la **priorità dinamica determina la runqueue in cui è inserito il task**, aggiungendo/sottraendo un bonus (fino a ± 5) alla priorità statica. Questo tipo di approccio **identifica e premia i task I/O – bound**; infatti, quando un task passa da BLOCKED a RUNNING, il suo tempo di attesa viene aggiunto ad un contatore (“tempo di sleep”), mentre quando un task passa da RUNNING a BLOCKED, il suo tempo di esecuzione viene sottratto al tempo di sleep: maggiore il tempo di sleep, maggiore il bonus.

- Completely Fair Scheduler (CFS)

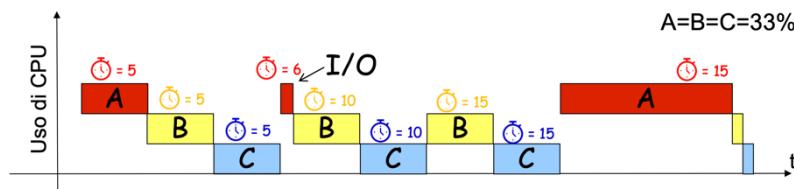
Questo tipo di scheduler, introdotto nella versione 2.6.23, **differisce dallo Scheduler O(1) per:**

- Fairness garantita;
- Assenza di euristiche per la stima dei task interattivi;
- Maggiore controllabilità dello scheduling;
- Supporto al group scheduling;
- Maggiore onere computazionale ($O(\log n)$).

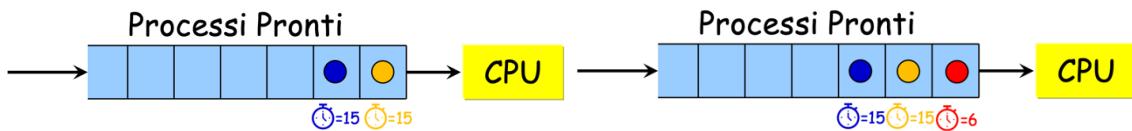
Con gli algoritmi classici (ad esempio, quelli con priorità) è **difficile controllare la percentuale di tempo di CPU concessa ad ogni task e non è garantita l'assenza di starvation**, mentre nel fair scheduling a ciascun processo è dedicata una frazione di tempo proporzionale alla sua priorità. Ad esempio, siano considerati tre processi con la stessa priorità, 2 processi CPU – bound (B e C) e uno misto I/O – bound/CPU – bound, con un solo processore:



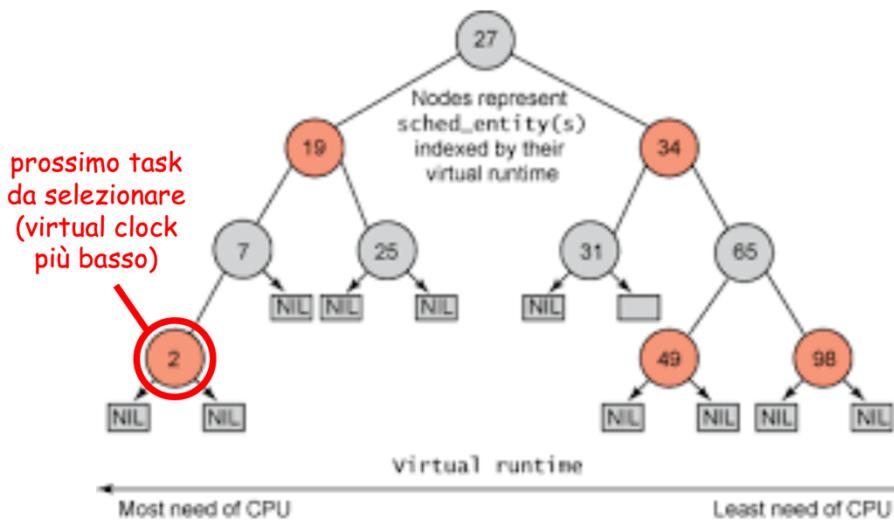
A ciascun task è associato un “clock virtuale” (detto **vruntime**) e, durante l’esecuzione del task, è periodicamente incrementato (tick del timer). L’algoritmo CFS fa in modo che i clock virtuali dei vari task non differiscano troppo tra loro e seleziona sempre il task con il minor valore di clock virtuale. Riprendendo l’esempio precedente:



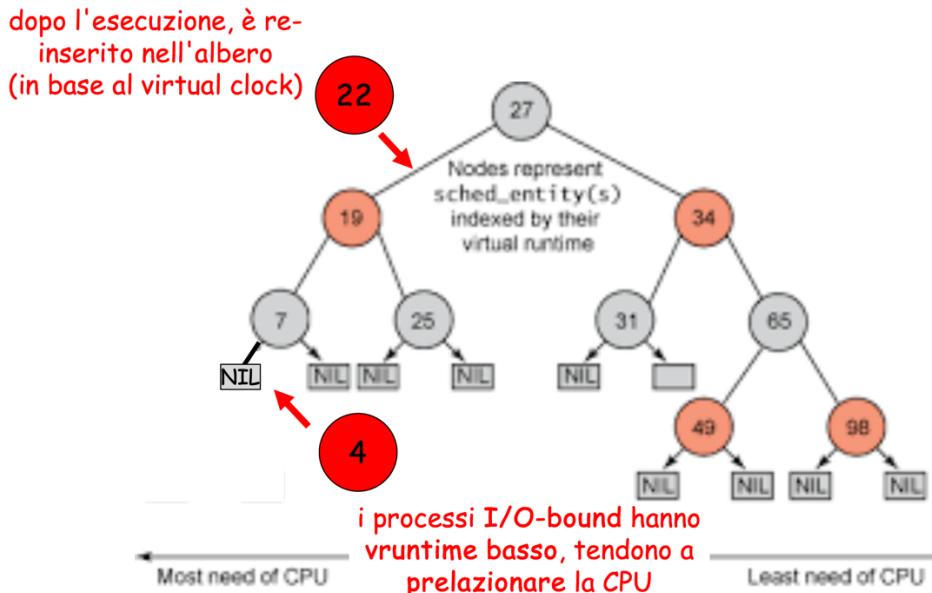
Nella runqueue:



I task sono raccolti in una struttura dati, detta **red – black tree**, che facilita la ricerca del valore più piccolo ad una complessità $O(\log n)$; tendenzialmente, i processi I/O – bound tendono ad avere un clock virtuale più basso.



Dopo l’esecuzione, il **virtual clock** del task è aumentato e dovrà, quindi, essere riposizionato più in alto nella struttura:



Per come l'algoritmo è costruito, il quanto di tempo non è un valore fissato a priori, viene determinato di volta in volta sulla base della targeted latency, una finestra temporale (alcuni ms) da dividere in modo equo tra i task in modo che ognuno esegua almeno una volta nella finestra:

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

Ovviamente, una grande quantità di task per una fissata targeted latency può portare un elevato overhead dovuto ai context switch; ad esempio, una targeted latency di 20ms per 200 task implica un timeslice di 0.1ms, rendendo necessari fin troppi context switch.

Una formula alternativa per determinare il timeslice di un task viene impiegata quando questo ha un “peso” (in termini di priorità) diverso rispetto agli altri:

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

La targeted latency è necessaria perché garantisce un limite superiore al tempo di risposta ed è un parametro configurabile del kernel:

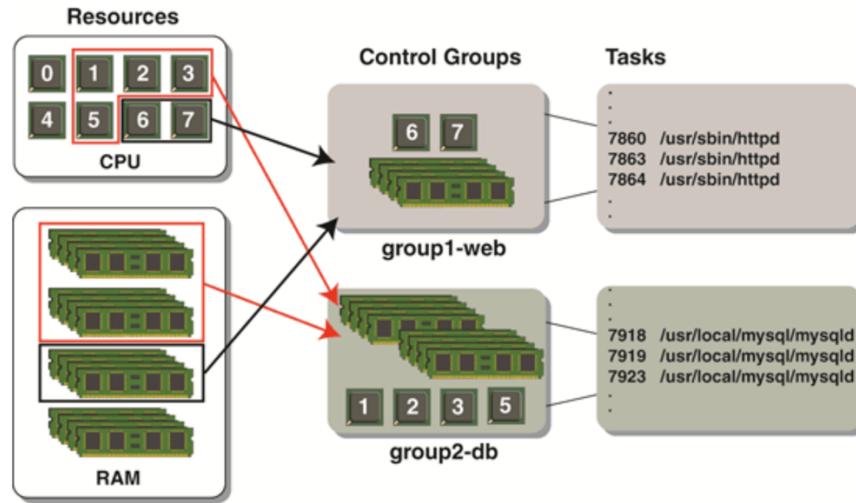
/sys/kernel/debug/sched/latency_ns

Per avere un **limite inferiore al timeslice** viene impiegato il parametro **minimum granularity** (ad esempio, se è *1ms* garantisce che, anche con 200 task, ognuna riceva almeno *1ms* di timeslice) e può essere **anch'esso manualmente configurato**:

/sys/kernel/debug/sched/min_granularity_ns

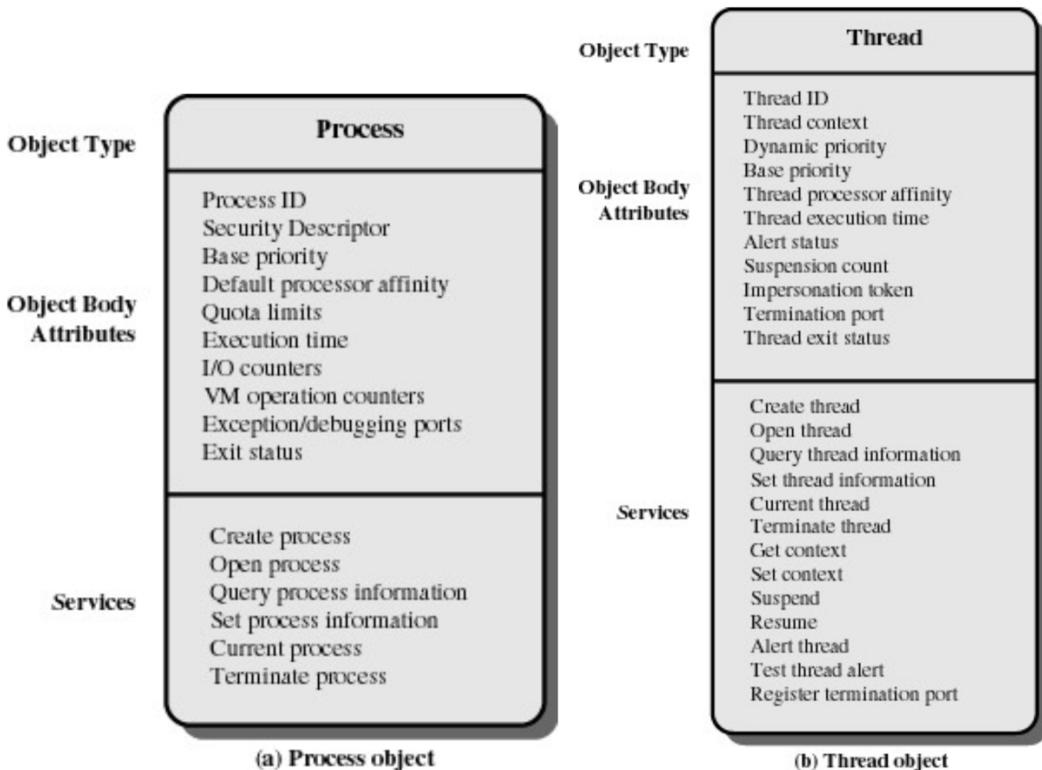
Il kernel Linux permette di raggruppare più processi in un control group (detto cgroup) per limitare l'uso di risorse (come CPU, memoria, disk I/O, network, ...) di un gruppo, prioritizzare l'accesso dei gruppi alle risorse, tracciare (accountability) l'uso di risorse e controllare lo stato (frozen, stopped, restarted, ...) di tutti i processi in un gruppo con un solo comando.

Graficamente, l'impiego di cgroups è visualizzato come segue:

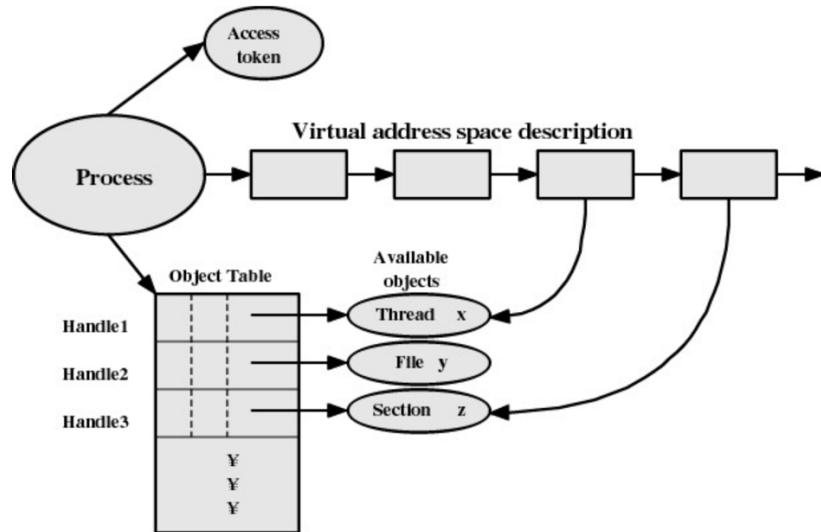


CFS può gestire i task in gruppi in modo tale che ognuno di essi sia gestito come un'unica entità schedulabile in maniera fair, ovvero facendo in modo che il timeslice sia assegnato all'intero gruppo e che all'interno di un gruppo i task si spartiscano il tempo disponibile.

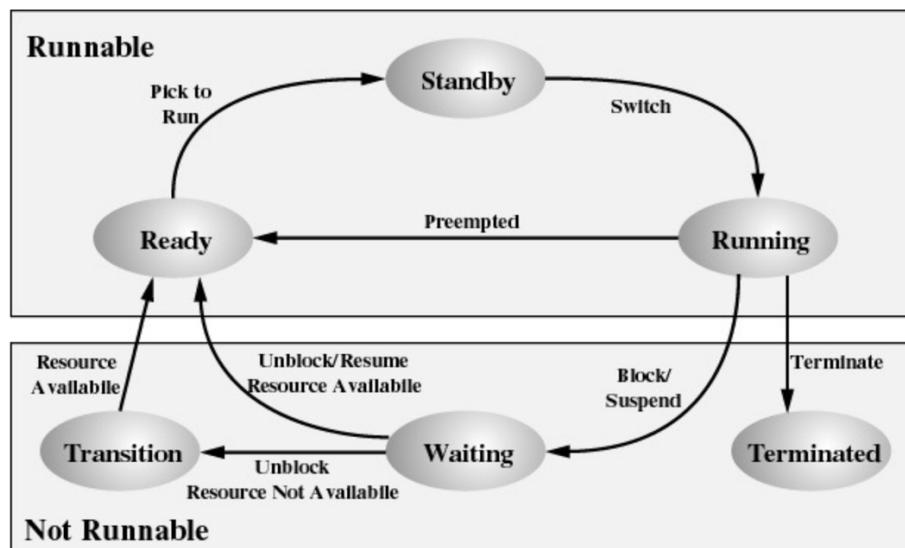
Windows 2000 (abbreviato in W2K) è un SO multithread e time – sharing tale che i processi/threads sono implementati secondo il paradigma Object Oriented: ogni processo è un oggetto (kernel object), che offre stato e funzionalità agli utenti attraverso dei riferimenti (handle), e può contenere uno o più threads, che a loro volta sono kernel objects. La concorrenza è realizzata dai thread, per cui un processo W2K deve contenere almeno un thread per eseguire.



Per le risorse di un processo W2K ci si riferisce al seguente schema:



Mentre per quanto riguarda gli stati di un thread in W2K:



La **politica di scheduling** dei thread in W2K è una **via di mezzo tra la politica prioritaria e quella dei “Round – Robin”**: ad **ogni thread viene associata una priorità assoluta che varia tra 0 e 31**, calcolata come somma di due componenti:

1. Una classe di priorità associata al processo a cui il thread appartiene;
2. Una priorità relativa del thread.

Ad ogni classe viene assegnato un valore numerico nominale; in ordine decrescente:

- Classi di priorità di un processo:
 - Real – time (24), precedenza su ogni thread, da usare con cautela;
 - High (13), precedenza sui thread delle classi inferiori, per applicazioni critiche (come il Task Manager);
 - Above Normal (10), per i thread leggermente più prioritari del normale;
 - Normal (8), maggiormente usato;
 - Below Normal (6), per i thread leggermente meno prioritari del normale;
 - Idle (4), i thread eseguono quando il sistema non ha altro da fare;
- Priorità relative di un thread:

- Time critical, priorità uguale a 15 (31 se la classe è Real – time);
- Highest, +2 rispetto alla nominale (valore della classe);
- Above Normal, +1 rispetto alla nominale;
- Normal, stessa della nominale;
- Below Normal, -1 rispetto alla nominale;
- Lowest, -2 rispetto alla nominale;
- Idle, priorità uguale a 1 (16 se la classe è Real – time).

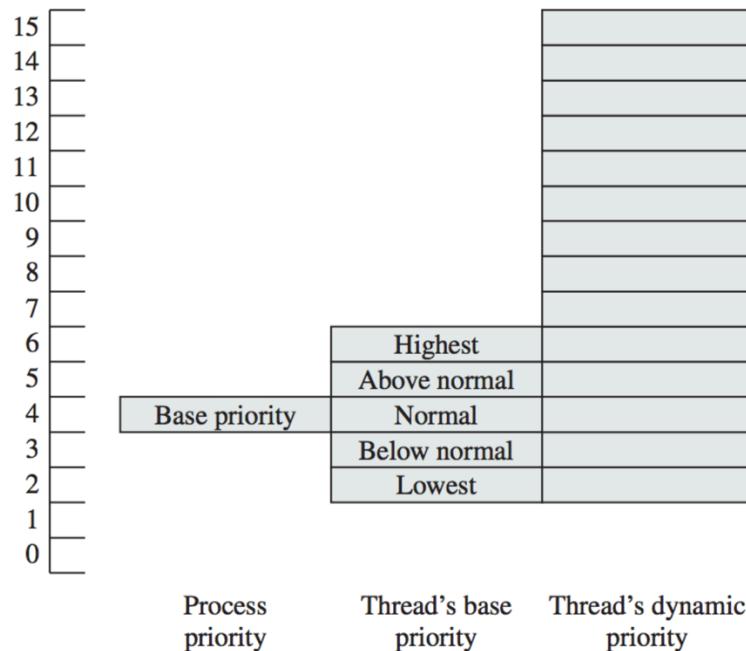
Per ogni livello di priorità, lo scheduler mantiene una coda gestita con Round – Robin con quanto di tempo 10ms, per poi eseguire il primo thread della coda a priorità più alta che contiene almeno un thread pronto. Il thread può eseguire fino a che:

- Il quanto temporale termina;
- Il thread si sospende in attesa di un evento esterno;
- Un thread a priorità più alta viene attivato.

In tutti questi casi, **lo scheduler viene invocato nuovamente per scegliere il successivo thread.**

Il meccanismo del **Dynamic Priority Boost** permette di **ridurre il tempo di risposta dei thread interattivi in accordo con le seguenti regole:**

- All'attivazione di un thread, la sua priorità base viene aumentata o diminuita fino a 2 punti;
- Al consumo completo di un quanto di tempo, la sua priorità viene decrementata di un punto;
- Se il thread si sospende in attesa di un evento I/O, la sua priorità viene incrementata;
- In ogni caso la priorità del thread non può essere inferiore alla sua priorità base.

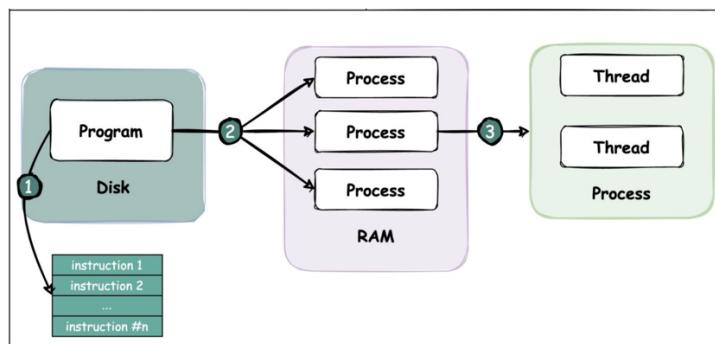


I THREADS

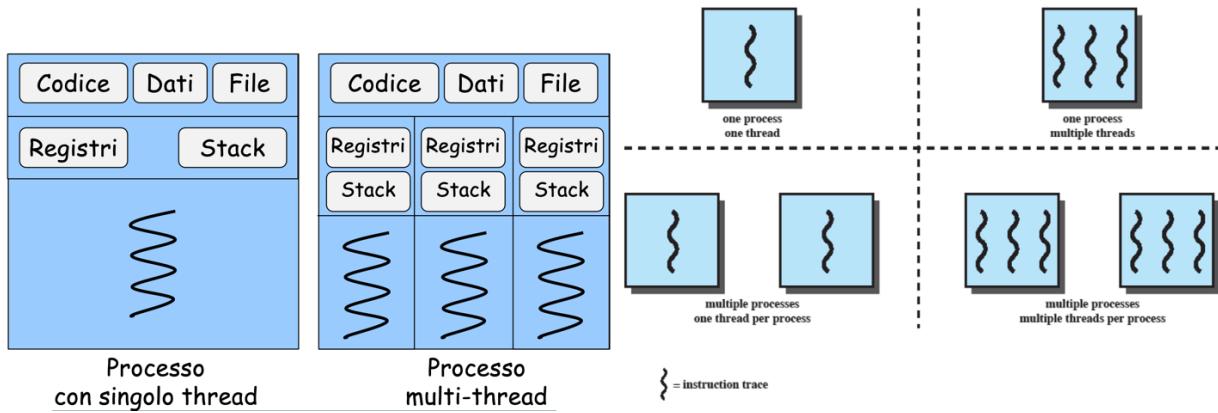
Finora è stato mostrato come **un processo è costituito da un codice di uno o più programmi, un insieme di locazioni per i dati, le variabili globali e locali, lo stack e il suo descrittore**, ovvero è fornito di immagine; aggiungendo poi le **risorse** necessarie all'esecuzione del processo, si costituisce lo spazio di indirizzamento del processo stesso. Tuttavia, **in un sistema di processi concorrenti, le operazioni di scambio tra due processi possono risultare onerose in termini di tempo di esecuzione (overhead), richiedendo il salvataggio e il ripristino del loro spazio di indirizzamento.**

La problematica appena enunciata, nei Sistemi Operativi moderni, è **risolta introducendo il concetto di thread**, un **flusso di esecuzione interno ad un processo che condivide con gli altri threads dello stesso processo le risorse e lo spazio di indirizzamento**; a questo punto, **non essendo dotato di risorse, la creazione, la distruzione e la modifica del contesto di un thread risultano essere operazioni eseguibili con maggiore facilità e con un overhead minore**. L'introduzione del concetto di thread **non sarebbe possibile se alla base dell'astrazione di un processo non ci fosse la dualità tra esecuzione e possesso di risorse**, due aspetti indipendenti che permettono di separare le responsabilità di un processo e alleggerire il carico della CPU:

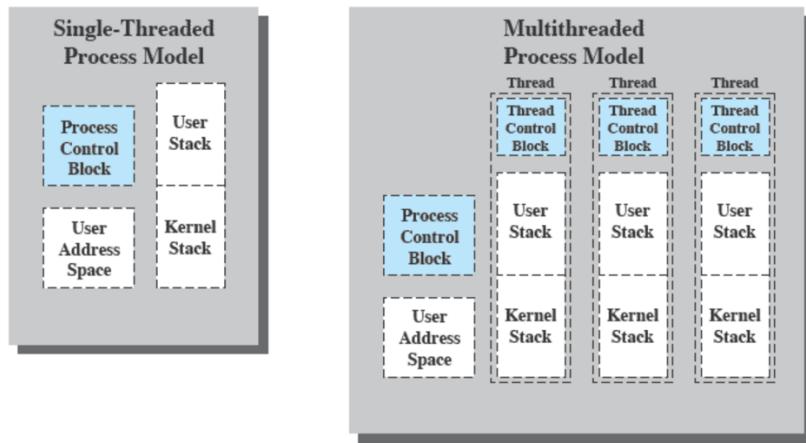
- **Esecuzione**, definito come l'insieme del flusso di controllo (ovvero, l'esecuzione di una sequenza di istruzioni) e del contesto di esecuzione (registri, stack, stato di scheduling) del processo;
- **Possesso di risorse (e protezione)**, definito come l'insieme dello spazio di indirizzamento (che identifica tutti gli indirizzi di memoria a cui un processo può accedere) e delle risorse di memoria e di I/O possedute dal processo.



Essendo **due aspetti indipendenti** ed essendo **il secondo la causa dell'overhead menzionato prima**, il **Sistema Operativo** può gestirli separatamente, ridefinendo il primo come **thread** e il secondo come **processo**, composto da uno o più **thread** (flussi di esecuzione) che condividono le stesse risorse. In realtà, **un Sistema Operativo che non include il threading può comunque essere interpretato secondo il paradigma appena descritto**, con la sola restrizione di **un thread a processo**, mentre **i Sistemi Operativi che implementano la possibilità di associare più thread ad uno stesso processo sono detti multithreaded**. Il **multithreading** è la capacità di un **Sistema Operativo** di consentire l'esecuzione di più **threads all'interno di un singolo processo**:



Più nello specifico, la differenza tra un processo ed un thread può essere visualizzata come segue:



Il **multithreading** implica, come già è stato evidenziato, che i **thread** che eseguono in uno stesso processo **condividono le risorse del processo stesso**; per questo motivo, è necessaria una sincronizzazione esplicita tra i threads, attraverso l'uso di **semafori o monitor**.

A questo punto possono già essere un po' più chiari i motivi per cui i threads sono utili:

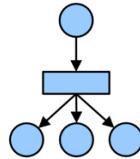
- **Suddividono il lavoro in più parti**, ciascuna assegnata ad un thread diverso;
- **Ogni thread può essere eseguito su CPU distinte**;
- Aumentano notevolmente la **velocità di esecuzione**;
- Conferiscono **maggior modularità** ai programmi.

In linea di principio, i vantaggi appena enunciati possono essere ottenuti anche con applicazioni **multi – processo**; tuttavia, rispetto a questi ultimi, i **programmi multi – thread** migliorano ulteriormente:

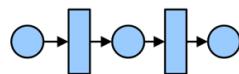
- **La creazione/terminazione di un thread**;
- **La comunicazione tra threads** (non coinvolge il kernel);
- L'overhead del context switch, più leggero tra threads che tra processi.

I tipici modelli di programmazione multithread sono tre:

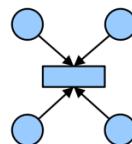
- **Manager/Workers**, per il quale **un thread, il manager, riceve in input i comandi e assegna i lavori ad altri threads, i workers**:



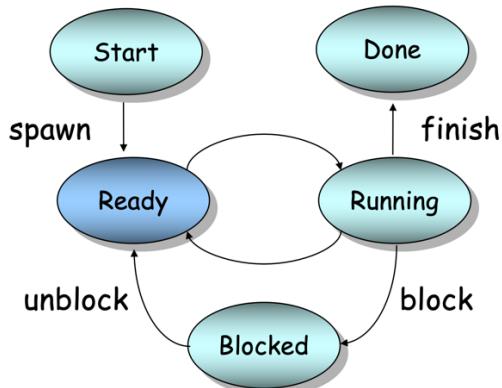
- **Pipeline**, per cui un task è suddiviso in una serie di operazioni più semplici, che possono essere eseguite in serie, e concorrentemente, da diversi threads:



- **Peer**, simile al modello Manager/Workers, ma per il quale una volta che il thread principale assegna il lavoro agli altri threads, partecipa attivamente anch'esso nel lavoro:



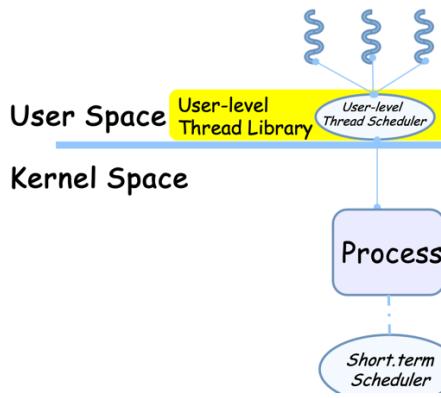
Così come un processo classico, anche i threads sono caratterizzati da uno stato che ne descrive la condizione di esecuzione:



Per implementare il **meccanismo dei threads in un Sistema Operativo**, possono essere seguite due strade:

- **User – Level Threads (ULT)**

La gestione dei threads avviene a livello applicativo senza che il kernel abbia visibilità o coscienza dell'esistenza dei threads:

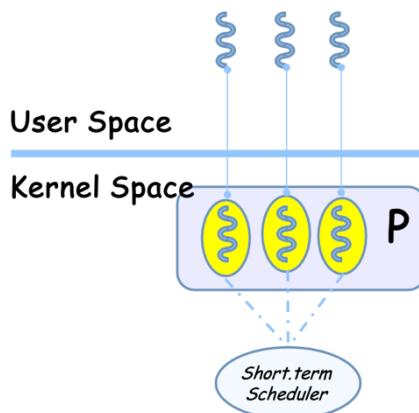


Dal punto di vista del Sistema Operativo, ad un programma sarà sempre associato un thread; è poi compito del programma stesso servirsi di una libreria esterna, una thread library, che gestisca il suo contesto di esecuzione e simuli il context switch del Sistema Operativo.

L'approccio illustrato prende il nome di **scheduling cooperativo**: ci si avvale dell'aiuto del programma per **preemption** e **context switch**. L'implementazione a livello utente dei thread garantisce un minore **overhead sul sistema** (perché il context switch non richiede il passaggio in kernel mode), l'**indipendenza dello scheduler dei thread da quello dei processi** (ogni applicazione può usare uno scheduler personalizzato) e la **portabilità delle applicazioni**; tuttavia, comporta anche il **bloccaggio di tutti i thread di un processo quando anche solo uno invoca una system call** e l'**assenza di vantaggi nelle architetture multiprocessore**, dal momento in cui il kernel assegna un processo per ogni processore e tutti i suoi thread devono essere necessariamente eseguiti sulla stessa CPU.

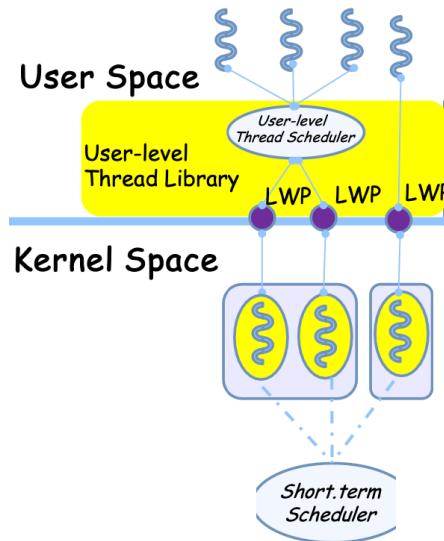
- Kernel – Level Threads (KLT)

È il Sistema Operativo che si fa carico di tutte le funzioni per la realizzazione dei threads e a ciascuna di esse corrisponde una chiamata di sistema, richiedendo che il nucleo contenga non solo i descrittori dei singoli processi ma anche di tutti i threads, così che in caso di sospensione di un thread sia in grado scegliere di mettere in esecuzione un thread dello stesso processo. In questo caso, a differenza del precedente, lo scheduling non è delegato al programma utente ma viene eseguito direttamente dal Sistema Operativo sui threads:

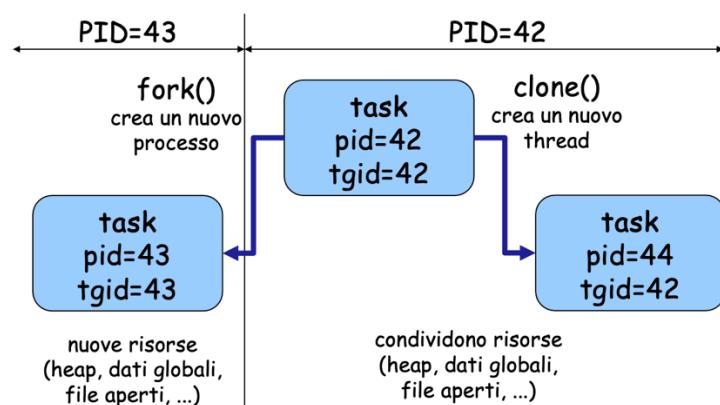


Questo approccio risulta essere decisamente meno efficiente del precedente ma, in ambienti multiprocessore, garantisce che threads diversi appartenenti allo stesso processo possano essere eseguiti su diverse unità di elaborazione. Si noti, infine, che il kernel stesso può essere sviluppato adottando un approccio multithreaded.

In realtà, è possibile anche un approccio combinato, per il quale più threads a livello utente corrispondono ad un numero inferiore o uguale di threads a livello del nucleo; la creazione di threads avviene nello spazio utente, dove avviene anche il grosso dello scheduling e della sincronizzazione.



In Linux, il task (un flusso di esecuzione) è l'unità fondamentale di scheduling ed **un thread stesso è un task che condivide delle strutture con altri task** (come codice, heap, ecc...); ogni task, poi, ha un Process ID (PID) univoco, mentre un “processo multithreaded” (ovvero, un gruppo di task) è identificato da un Thread Group ID (TID). La creazione di un thread avviene attraverso la chiamata del sistema `clone()` ma, anziché creare una copia del task chiamante come sarebbe intuitivo pensare, **crea un nuovo task che condivide lo spazio d'indirizzamento del task chiamante**, mentre la chiamata di sistema `fork()` dello standard POSIX, implementata attraverso `clone()`, **crea effettivamente un nuovo processo**.



In Windows 2000, invece, i threads sono mappati uno ad uno tra ULT e KLT, mentre ciascun thread contiene:

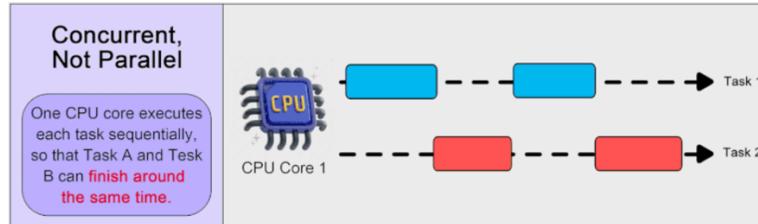
- Un **identificatore** di thread (ID);
- Un **insieme di registri**;
- Una **pila d'utente** e una **pila del nucleo**;
- Un'area di memoria **privata**.

In Java, i thread possono essere creati in due modi: creando una nuova classe derivata dalla classe **Thread** o ridefinendo il metodo **run()** di quella classe; in entrambi i casi, sono gestiti dalla macchina virtuale (JVM).

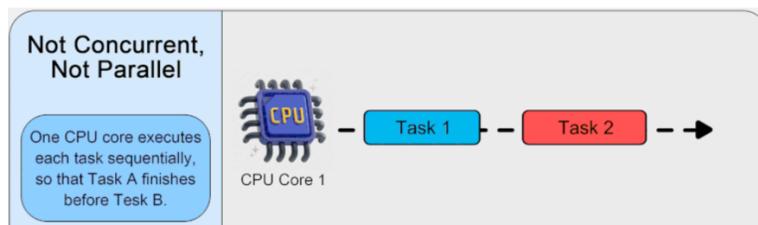
PROGRAMMAZIONE CONCORRENTE

LA PROGRAMMAZIONE CONCORRENTE

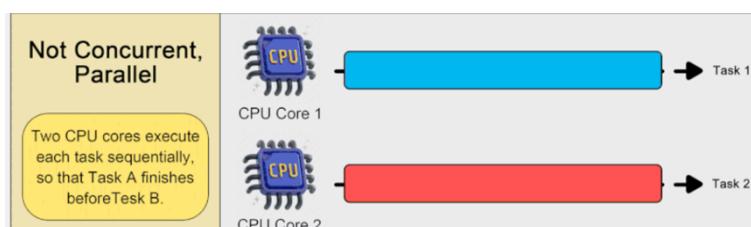
Per **programmazione concorrente** si intende l'insieme delle tecniche, delle metodologie e degli strumenti per lo sviluppo di software in quanto insieme di attività svolte simultaneamente. La **multiprogrammazione** è la forma più elementare di programmazione concorrente e prevede l'esecuzione intercalata di più processi (o threads) sulla stessa CPU; il sistema diventa, così, una macchina astratta che dispone di più processi virtuali, uno per ogni processo:



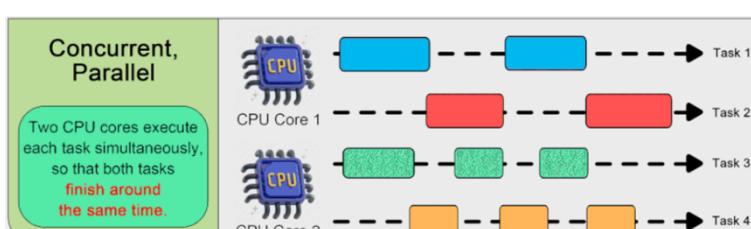
Confrontando con la classica programmazione imperativa:



La multiprogrammazione migliora l'efficienza complessiva del sistema, pur non migliorando la velocità di esecuzione del singolo programma; infatti, è possibile facilmente intuire che sia in programmazione imperativa che concorrente, il tempo totale di esecuzione delle due task è lo stesso. Il parallelismo, invece, utilizza più CPU fisiche su cui eseguire più processi contemporaneamente, migliorando il tempo di esecuzione dei task ma non velocizzando l'esecuzione del singolo programma. L'idea alla base del parallelismo è del tutto diversa da quella che muove la multiprogrammazione: non si parla più di concorrenza o alternanza ma effettivo parallelismo.



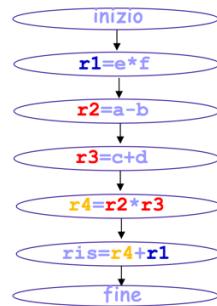
Concorrenza e parallelismo possono essere applicati insieme: per accelerare il singolo programma, è necessario suddividerlo in più elaborazioni concorrenti ed eseguirle su più processori; graficamente:



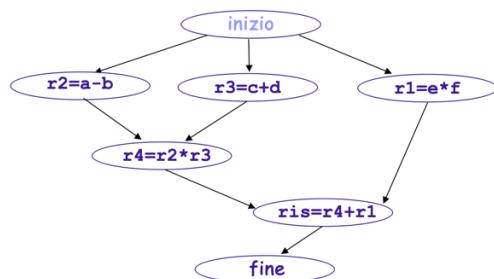
Nella programmazione concorrente, è il programmatore a definire le elaborazioni concorrenti, ognuna delle quali è eseguita in un processo/thread; in questo modo, rispetto alla programmazione imperativa, è migliorata la velocità di esecuzione di un programma. La potenza della programmazione concorrente si mostra ancora di più quando ci si rende consapevoli che l'ordinamento delle istruzioni può non essere sequenziale; ad esempio, si voglia scrivere un programma per il calcolo della seguente espressione aritmetica:

$$(a - b)(c + d)(ef)$$

Adottando un approccio imperativo, il grafo di precedenza è il seguente:



Evidenziando una **struttura sequenziale**; adottando, invece, un approccio concorrente:

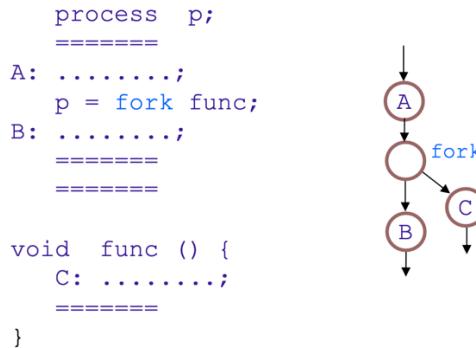


Evidenziando una **struttura non più sequenziale** (o almeno, **non completamente sequenziale**, la relazione di precedenza temporale è ancora evidente in alcune elaborazioni) ed un **albero con profondità minore dell'omologo imperativo**.

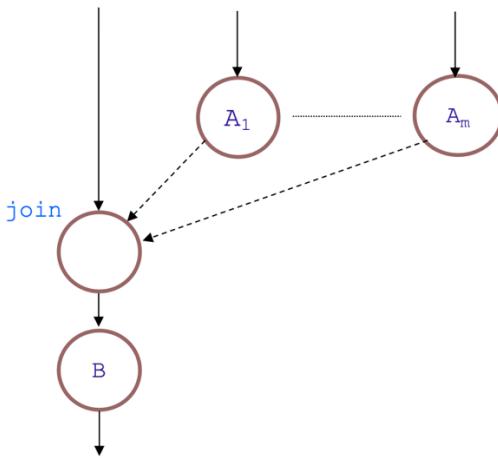
Alla base della programmazione concorrente ci sono **due tipologie di primitive**:

- **Primitive per definire attività indipendenti** (come processi o threads);
- **Primitive per la comunicazione e la sincronizzazione** tra attività eseguite in maniera concorrente.

Le due operazioni principali che possono essere eseguite a partire da un processo sono due: **fork** e **join**. L'esecuzione di un'operazione di **fork** crea ed attiva un nuovo processo figlio, eseguito in maniera concorrente con il processo padre:



L'operazione di **join**, invece, consente al processo di sincronizzarsi con la terminazione di un altro processo, quest'ultimo creato tramite la **fork**:



I problemi principali di questo approccio sono la **necessità di far comunicare i processi**, la **condivisione delle risorse**, la **sincronizzazione manuale** e l'**assegnazione mandatoria del tempo del processore**. Si propone, di seguito, un esempio per comprendere come possa avvenire una **mancata sincronizzazione tra processi**; si consideri il seguente codice, in cui **ad ogni iterazione corrispondono tre istruzioni macchina**:

```

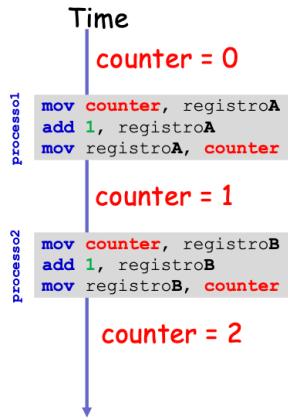
int counter = 0;

process1() {
    for(1 ... 1,000,000)
        mov counter, registroA
        add 1, registroA
        mov registroA, counter
}

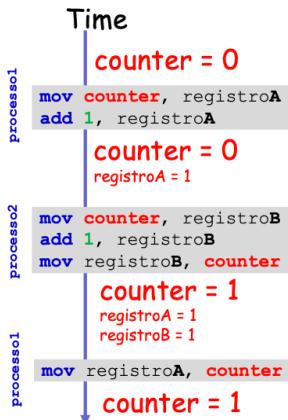
process2() {
    for(1 ... 1,000,000)
        mov counter, registroB
        add 1, registroB
        mov registroB, counter
}

```

Intuitivamente, si potrebbe pensare che il flusso di esecuzione sia il seguente:



In realtà, per via del time sharing, il **processo1** può essere prelazionato alla seconda istruzione, lasciando il contenuto dell'elaborazione nel **registro** piuttosto che nel **counter**; a questo punto, il **processo2** legge il valore 0 e incrementa il **counter**, mentre l'ultima istruzione annulla completamente quanto fatto dal secondo processo e registra solo il risultato del primo. L'elaborazione è errata:



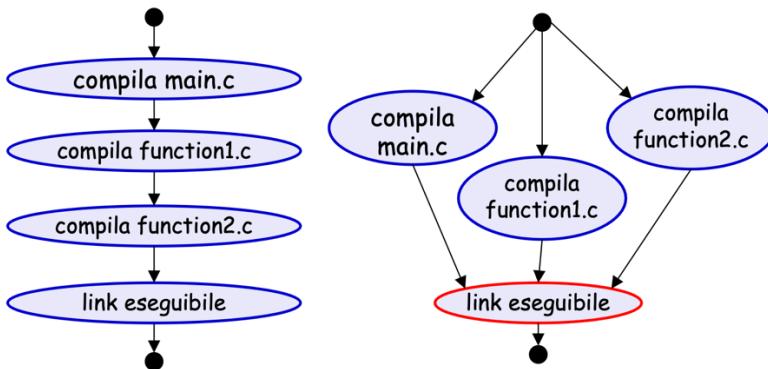
Per l'esempio appena mostrato, si può dire che si è verificata una “Race condition”, ovvero una condizione per la quale più processi leggono o scrivono dati condivisi, facendo dipendere il risultato dell'elaborazione totale dall'ordine (e dalla velocità) con cui le istruzioni dei vari processi vengono eseguite.

Si vuole anche menzionare il concetto di **processi interagenti**, facendo riferimento a quel tipo di esecuzione in cui il **processo P1** è influenzato da **P2** e viceversa, producendo un **comportamento imprevedibile** (nel senso di **non riproducibile**), ovvero un comportamento che **dipende dall'ordine** in cui i processi sono eseguiti nel sistema. I tipi di interazione sono tre: **competizione**, per la quale le risorse sono comuni ma non utilizzabili contemporaneamente (mutua esclusione), **cooperazione**, per la quale c'è uno scambio di informazioni per portare a termine un'elaborazione comune (comunicazione), e **interferenza**, per la quale c'è l'erronea soluzione di problemi di competizione e di cooperazione o l'uso non autorizzato di risorse comuni.

Per un corretto funzionamento, è necessario inserire nel programma dei vincoli circa l'esecuzione per le elaborazioni concorrenti (come locks, semafori, barriere, guardie, monitor, ...) che, seppur riducano la velocità di esecuzione dei programmi concorrenti, garantiscono la correttezza dei risultati. Si consideri il seguente comando:

```
gcc main.c function1.c function2.c -o eseguibile
```

Che può essere schematizzato come segue, sia in una struttura sequenziale che in una non sequenziale:



Chiaramente, per quest'ultimo caso, l'operazione di linking deve attendere direttamente non solo la compilazione di `funzione2.c` ma anche quella di `funzione1.c`. Supponendo ci vogliano:

- 3 secondi per compilare `main.c`;
- 2 secondi per compilare `function1.c`;
- 1 secondo per compilare `function2.c`;
- 1 secondo per il linking dei file oggetto.

Il tempo totale di esecuzione, nel caso sequenziale, è di 7 secondi; avendo, però, a disposizione **tre CPU**, ci si chiede in quanto tempo sia possibile completare le stesse operazioni. Osservando la struttura dell'albero non sequenziale, si noti che **la CPU che opera dall'entry point all'operazione di linking**, per la quale è necessario un solo secondo, **può operare per un tempo minimo di 1 secondo** (nel caso in cui si passi per la compilazione di `function2.c`) **ad uno massimo di 3 secondi** (nel caso in cui si passi per la compilazione di `main.c`); di conseguenza, **si può ragionevolmente pensare che il tempo di esecuzione totale nel caso non sequenziale sia di 4 secondi**.

Per quantificare il (mancato) “guadagno” nel passare da un programma sequenziale ad uno parallelo è possibile utilizzare il **parametro di speed – up**:

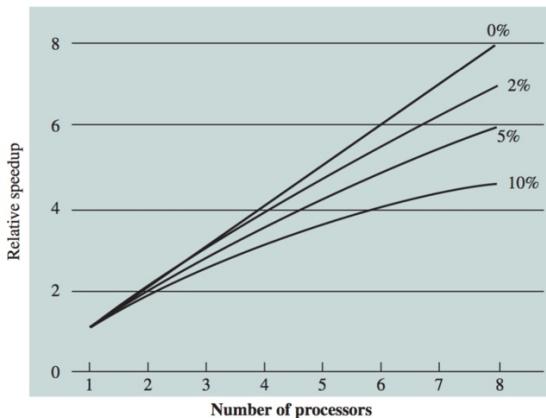
$$S = \frac{T_s}{T_p}$$

Con T_s il **tempo di esecuzione sequenziale del processo** e T_p il **relativo tempo di esecuzione parallela**. Idealmente, si ha $S = N = \text{numero di processi/CPU nel sistema}$; tuttavia, questa **condizione non sempre si verifica** (come nell'esempio precedente, in cui lo speed – up è 1.75 nonostante teoricamente avrebbe dovuto essere 3, in questo caso causato dal vincolo di sincronizzazione). Come si è potuto intuire, lo speed – up è **limitato dalla quantità di operazioni sequenziali (non – parallelizzabili) che ci sono nel programma**, ovvero dipende dalle **caratteristiche del problema da risolvere**; per evidenziare quanto appena detto, è possibile ricorrere ad una **riformulazione della formula di speed – up** (più pratica, dal momento in cui il tempo di esecuzione non sempre è una quantità determinabile a priori), detta **legge di Amdahl**:

$$S = \frac{N}{1 + (N - 1)f}$$

Con f percentuale di codice sequenziale (non – parallelizzabile) e N = numero di processi/CPU nel sistema. La formula in questione permette di evidenziare come **una maggior quantità di codice sequenziale corrisponda ad un minor speed – up**. Esempi notevoli sono:

- $f = 0 \Rightarrow S = N$, il codice è tutto parallelizzabile e lo speed – up è massimizzato;
- $f = 1 \Rightarrow S = 1$, il codice è tutto sequenziale e lo speed – up è minimizzato.

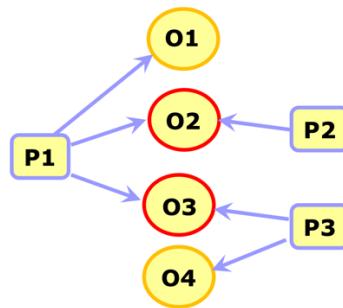


(a) Speedup with 0%, 2%, 5%, and 10% sequential portions

I vincoli di sincronizzazione precedentemente menzionati si possono mostrare in una **duplice veste**, in quanto **problemi di competizione** (imponendo che un solo processo alla volta acceda alla risorsa comune) o in quanto **problemi di cooperazione** (imponendo un ordinamento, parziale, tra le operazioni dei processi).

Prima di definire i modelli di programmazione concorrente è necessario chiarire delle definizioni: **una risorsa è definita come un oggetto**, fisico o logico, **di cui un processo necessita per la propria elaborazione**; una risorsa può essere **privata** (o locale), se è vista da un solo processo, o **comune** (o globale), se è condivisibile tra più processi. Con queste ipotesi, è possibile definire i due modelli di programmazione concorrente: **modello ad ambiente locale** e **modello ad ambiente globale**.

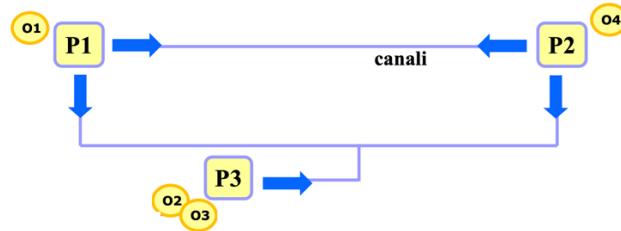
Nell'esempio che segue, le **risorse O1 e O4 sono private**, mentre **O2 e O3 sono globali**:



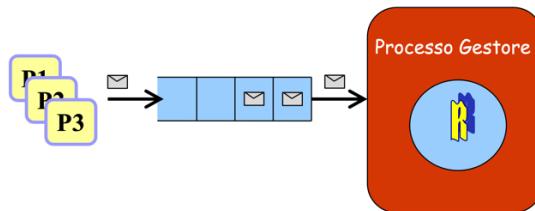
Si può facilmente intuire come **l'interazione tra più processi non possa avvenire se non utilizzando risorse comuni**, non private.

Nel **modello ad ambiente globale**, per garantire i vincoli di sincronizzazione, ci si avvale dell'**intermediazione di un gestore**, un insieme di procedure (e delle relative strutture dati) che operano sulla risorsa; in realtà, **il gestore stesso è condiviso tra tutti i processi**, i quali sono **trasparenti tra di loro**. Seguendo questa struttura, **le procedure del gestore disciplinano l'ordine di accesso alle risorse**, in modo tale da **garantire un accesso ordinato quando più processi fanno una chiamata a procedura**.

Nel **modello ad ambiente locale**, invece, le risorse sono private ai processi e gli altri vi accedono interagendo solo con il processo proprietario, in una relazione binaria senza intermediazione esterna (come accade per il modello ad ambiente globale), e tramite l'impiego di messaggi:



In questo modello, la risorsa è acceduta dai processi per via di un processo gestore, al quale è necessario inviare un messaggio di richiesta che, infine, verrà eseguita dal processo gestore stesso:



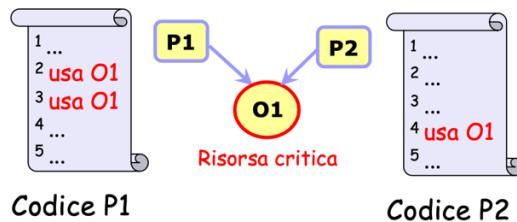
Gli strumenti di sincronizzazione a disposizione per i due diversi modelli sono i seguenti:

- Semafori e primitive **wait** e **signal**, per il modello ad ambiente globale;
- Messaggi e primitive **send** e **receive**, per il modello ad ambiente locale.

SINCRONIZZAZIONE NEL MODELLO AD AMBIENTE GLOBALE

La necessità di sincronizzare i processi nel modello ad ambiente globale sorge dal problema di **competizione**, che occorre quando più processi competono nell'uso della stessa **risorsa**; per risolvere il problema e per garantire l'uso corretto della **risorsa**, essa deve essere raggiunta da al più un processo alla volta (ad esempio, una stampante deve stampare un file alla volta, non può prendere in consegna ed eseguire la stampa su due documenti contemporaneamente).

Si supponga che due o più processi vogliono utilizzare una risorsa ad uso esclusivo, detta **risorsa critica**, allora la porzione di codice che usa la risorsa in questione è detta **sezione critica**:

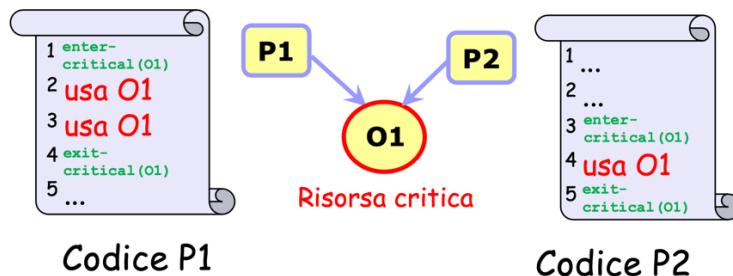


La **sezione critica**, come si può immaginare, può essere composta da più parti di codice, eseguite da processi differenti; nell'esempio riportato sopra, la sezione critica è costituita dalle righe 2 e 3 di P1 e dalla riga 4 di P2. La gestione della sincronizzazione, sotto le ipotesi di cui sopra, può avvenire seguendo due approcci:

- **Mutua esclusione**, per la quale l'ordine con cui devono avvenire due eventi non è fissato e non è importante, finché i processi non utilizzano contemporaneamente la risorsa;
- **Comunicazione**, per la quale è posto un ordinamento tra gli eventi.

Riprendendo l'esempio della stampante, **non è importante quale documento è stampato prima, purché non si cerchi di stampare i due (o più) documenti contemporaneamente**; è adottato l'approccio della **mutua esclusione**.

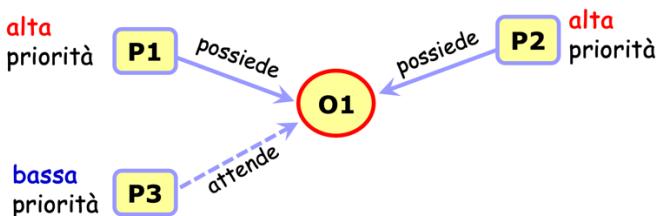
Il Sistema Operativo fornisce ai programmi i meccanismi per richiedere accesso in mutua esclusione alle risorse, adottando diverse soluzioni (hardware e software):



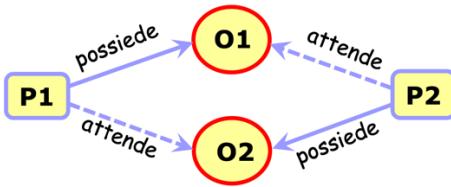
Quando P1 entra nella sezione critica, esso “acquisisce” il possesso della risorsa; se P2 tenta di entrare nella sezione critica prima che P1 ne sia uscito, viene posto in attesa dal Sistema Operativo. Da questo semplice esempio sono chiari i requisiti per la mutua esclusione:

- Un solo processo alla volta può accedere alla sezione critica;
- Il corretto funzionamento non dipende dal numero o dalla velocità di esecuzione dei processi (quindi, no Race condition);
- Devono essere evitati deadlock e starvation dei processi;
- Se non vi sono altri processi nella sezione critica, un processo deve poter accedere immediatamente alla risorsa;
- Quando un processo non è nella sezione critica, esso non interferisce con l'uso della risorsa da altri processi;
- Un processo può rimanere in una sezione critica solo per un tempo finito.

La **starvation**, come anticipato in precedenza, denota una condizione di attesa indefinita da parte di un processo a bassa priorità, causata potenzialmente dal continuo alternarsi di processi ad alta priorità nel possesso delle risorse comuni:



Il **deadlock**, invece, indica la presenza di una condizione di blocco permanente (stallo) di un gruppo di processi in competizione, in modo che ogni processo possiede una risorsa in mutua esclusione, bloccando il progresso dell'altro:



Il supporto hardware alla mutua esclusione può avvenire **disabilitando gli interrupt o implementando l'istruzione Test – and – Set – Lock**.

In un sistema monoprocesso, per garantire la mutua esclusione a livello hardware, è sufficiente **disabilitare le interruzioni** (no timer, no I/O), evitando così che un processo in una sezione critica venga prelazionato:



Questo approccio **non è privo di problemi**:

- La disabilitazione delle interruzioni **in un sistema multiprocessore non garantisce la mutua esclusione**, limitando questo metodo risolutivo a sistemi che nel tempo stanno diventando sempre più rari;
- **Sono violati i principi di protezione**, non è sicuro consentire ai processi utente di disabilitare le interruzioni;

Nonostante ciò, è **conveniente per uso interno nel kernel**, quando deve aggiornare delle variabili condivise interne.

In alternativa, è possibile approcciare al problema utilizzando una variabile condivisa (Lock), **inizialmente 0 ed associata ad una risorsa**; quando la variabile in questione viene modificata ad 1, si codifica la **condizione per la quale la risorsa in gioco è occupata ed il processo che la vede deve attendere**. Nel momento in cui **il processo che sta impiegando la risorsa termina il possesso** (terminando il ciclo di esecuzione della sezione critica), **la variabile Lock torna ad essere 0** ed il processo al quale è stato detto di attendere può prendere lui il possesso della risorsa.

Così descritto, l'**approccio in questione non è sufficiente**: **in caso di context switch prima di aver posto la variabile di Lock ad 1, un altro processo può entrare nella sezione critica**, violando così la mutua esclusione; **la causa del problema è da ricercare nel fatto che la lettura e la scrittura del Lock sono in due momenti diversi, sono due operazioni divisibili**. Per risolvere via hardware il problema, **molti processori forniscono l'istruzione TSL RX, LOCK** (o versioni analoghe), che equivale a:

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

In modo da rendere indivisibili le operazioni di lettura e scrittura di Lock (eseguono entrambe in un solo ciclo di CPU); questa istruzione, inoltre, può essere utilizzata anche nei sistemi

multiprocessore, in quanto **inibisce l'accesso alla memoria alle altre CPU**. Infine, si vuole menzionare che **questa soluzione è caratterizzata da busy wait**, ovvero dal fenomeno per cui **il processo permane in attesa senza fare nulla mentre l'altro esegue**:

```

LOOP: TSL RX, LOCK          //Copia il Lock in RX e lo setta a 1
      CMP RX, 0            //Verifica se il Lock è 0
      JNE LOOP              //Lock != 0, busy wait
      ...
      MOVE LOCK, 1          //Lock == 0, entra nella regione critica
      ...
      MOVE LOCK, 0          //Esegue il blocco
      MOVE LOCK, 0          //Esce dalla sezione critica
  
```

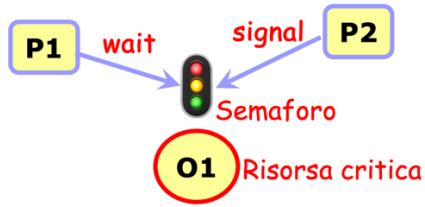
<u>Process P1</u>	<u>Process P2</u>	<u>lock</u>
.	.	0
tsl x, lock	.	1
if x == 0, vai avanti	.	1
.	tsl x, lock	1
ATTESA	if x == 0, vai avanti	1
ATTIVA	tsl x, lock	1
.	if x == 0, vai avanti	1
.	...si ripete...	1
...sezione critica...	.	1
lock = 0	.	0
.	tsl x, lock	1
.	if x == 0, vai avanti	1
.	...sezione critica...	1

Oltre al busy waiting, non ottimale quando si vuole massimizzare l'utilizzo delle risorse, **l'approccio con le variabili Lock presenta anche un altro problema: il priority inversion**. Si supponga di disporre di **due processi che accedono alla stessa sezione critica**, il processo **H** con priorità alta e il processo **L** con priorità bassa; si supponga, poi, che vi sia inizialmente in esecuzione solo **L** e che esso entri nella sezione critica, impostando il **Lock** ad **1**. Il processo **H** diviene “pronto” e, avendo priorità maggiore, prelaziona **L** facendo assegnare la CPU a sé stesso; tuttavia, essendo il **Lock** lasciato ad **1**, **H** inizia il busy waiting, rimanendo così bloccato nel ciclo di attesa attiva per sempre, dal momento in cui **L** non avrà mai la possibilità di eseguire (e quindi di modificare il **Lock**) quando **H** è in esecuzione.

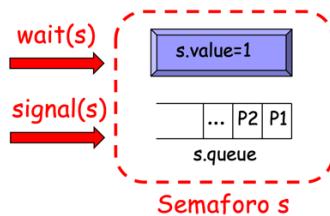
I problemi di busy waiting e di priority inversion vengono risolti “forzando” il processo che trova la variabile Lock al valore 1 a sospendersi (ovvero a transitare nello stato bloccato) in attesa che la risorsa divenga disponibile. In questo caso, è necessario l’uso di due system call:

- **suspend(P)**, il processo P che la chiama si auto – sospende in attesa di un segnale di risveglio;
- **wakeUp(P)**, un processo invia un segnale di risveglio al processo P, che è stato precedentemente sospeso con l’istruzione suspend).

I semafori sono variabili speciali utilizzate per la competizione e la cooperazione tra processi. I processi in un sistema condividono una stessa istanza di semaforo **s** per coordinarsi ed ognuno di essi può eseguire la primitiva **wait(s)** sul semaforo in argomento per sospendersi, in attesa di ricevere un segnale, o di proseguire, se il segnale è stato già ricevuto; poi, un altro processo può inviare sullo stesso semaforo un segnale tramite la procedura **signal(s)**:



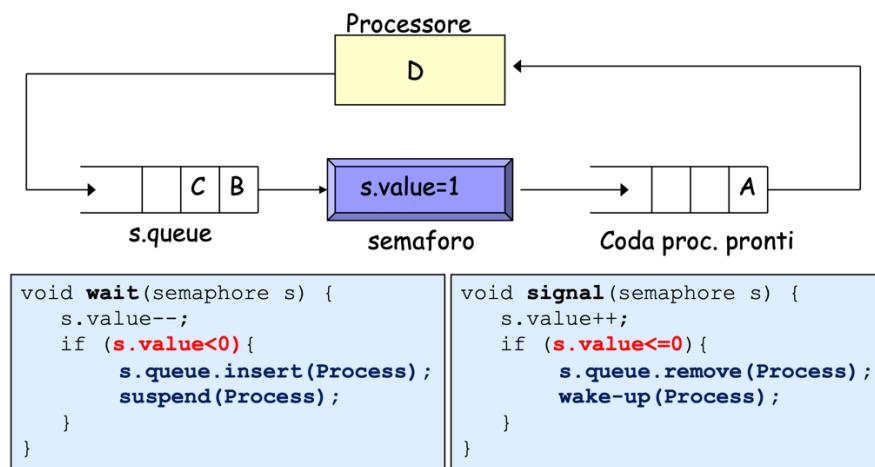
Nel dettaglio, un semaforo **s** è un tipo di dato astratto che incapsula una variabile di tipo intero (**s.value**) e una coda (**s.queue**), quest'ultima utilizzata per tenere traccia dei processi che si sono sospesi con **wait(s)** nell'attesa di una **signal(s)**. Graficamente, un semaforo è composto come segue:



Sui semafori sono definite le seguenti operazioni:

- **Inizializzazione della variabile ad un valore non negativo**, scelto dal programmatore in base al tipo di interazione che si vuole utilizzare (mutua esclusione, comunicazione, ...);
- L'operazione di **wait** ha l'effetto di **decrementare il valore del semaforo**, il quale se **negativo rende bloccato il processo chiamante**;
- L'operazione di **signal** ha l'effetto di **incrementare il valore del semaforo**, il quale se **minore o uguale a zero rende sbloccato uno dei processi sospesi**.

Il modello concettuale più completo con cui si presenta un semaforo è il seguente:



Si noti la necessità che l'accesso alla variabile **s** sia a sua volta mutuamente esclusivo: ad esempio, tramite l'uso di una TSL, ovvero di una variabile di Lock; la sospensione, in questo caso, allevia comunque i problemi di busy waiting e di priority inversion.

Una particolare variante di semaforo sono i Mutex, o semafori binari, in cui il valore intero può assumere solo i valori 0 e 1, ed è un tipo di semaforo chiaramente più semplice da utilizzare (ad esempio, per soli problemi di mutua esclusione).

<pre>void waitB(Binary_sem s) { if (s.value==1) { s.value=0; } else { s.queue.insert(Process); suspend(Process); } }</pre>	<pre>void signalB(Binary_sem s) { if (s.queue.is_empty()) { s.value=1; } else { s.queue.remove(Process); wake-up(Process); } }</pre>
--	--

In conclusione, l'utilizzo dei semafori per la mutua esclusione può essere così riassunto:

```
semaphore s; /* condiviso tra i processi */

int main() {
    s.value = 1; /* inizializzazione */

    /* avvia l'esecuzione concorrente di
       P(1) ... P(n) */
}

void P(int i) {

    ...
    wait(s);
    /* sezione critica */
    signal(s);
    ...
}
```

Questa soluzione richiede di **inizializzare il valore del semaforo** (operazione estremamente importante) **ad 1**, con **s.value = 1**. P1 trova il valore del semaforo alto, lo abbassa a 0 ed entra nella **sezione critica**; successivamente, quando P2 vuole eseguire trova il valore a 0 e rimane in attesa di una **signal**, che occorre da P1 quando la sezione critica è stata eseguita.

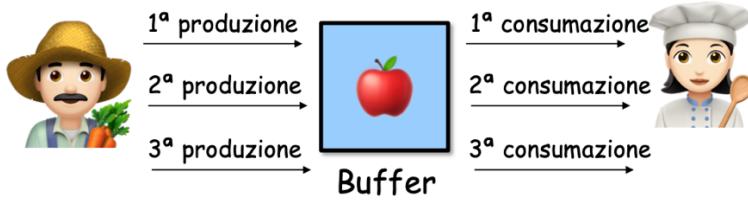
PROBLEMI NEL MODELLO AD AMBIENTE GLOBALE

In questa sede, per **problemi nel modello ad ambiente globale** si intendono **problemi di cooperazione** (produttore/consumatore e lettori/scrittori) perché, pur esistendo un problema potenziale di **mutua esclusione nell'utilizzo di buffer comuni**, la soluzione impone un **ordinamento nelle operazioni dei processi**, rendendo necessario che questi si coordinino, indicando agli altri quando le loro operazioni sono avvenute e risolvendo implicitamente il problema di mutua esclusione stesso.

Il primo dei problemi di cooperazione è il **problema produttore/consumatore**, che parte dal presupposto per il quale esistono due categorie di processi: i processi produttori, che **depositano un messaggio su una risorsa condivisa**, e i processi consumatori, che **prelevano il messaggio dalla stessa risorsa**. I vincoli che questa categorizzazione induce sono due:

- Il produttore non può produrre un messaggio prima che qualche consumatore abbia prelevato il messaggio precedente;
- Il consumatore non può prelevare alcun messaggio fino a che qualche produttore non ne abbia depositato uno.

Il problema, così formato, viene denotato con **problema produttore/consumatore ad unico buffer** e **necessita la coordinazione dei due soggetti**, per indicare rispettivamente l'avvenuto deposito e prelievo.



Il produttore non può produrre un messaggio se il consumatore non ha consumato il messaggio precedente.

Per la sincronizzazione dei due processi, si utilizzano **due semafori**:

- **SPAZIO_DISP**, rimane bloccato da un produttore prima di una produzione e viene sbloccato da un consumatore in seguito ad un consumo (valore iniziale 1);
- **MSG_DISP**, rimane bloccato da un produttore in seguito ad una produzione e viene sbloccato da un consumatore prima del consumo (valore iniziale 0).

La **produzione** ed il **consumo** avvengono, rispettivamente, all'interno delle procedure **produttore(msg*, int)** e **consumatore(msg*, int)**, all'interno delle quali si effettuano anche le operazioni di **Wait_Sem** e di **Signal_Sem** necessarie per la sincronizzazione.

```

void Produttore(msg * ptr_sh, int sem){
    msg mess;
    Wait_Sem(sem, SPAZIO_DISP);
    // Produzione di valore_prodotto ...
    printf ("Produzione in corso...");
    mess = valore_prodotto;
    *ptr_sh = mess;
    Signal_Sem(sem, MSG_DISP);
}
  
```

Con **Wait_Sem()** che sospende il chiamante se **SPAZIO_DISP == 0** e non sospende il chiamante se **SPAZIO_DISP == 1**, ponendo anche **SPAZIO_DISP = 0**.

Si supponga che il buffer sia inizialmente vuoto (quindi, **SPAZIO_DISP = 1** e **MSG_DISP = 0**). Alla prima produzione, il processo entra nella sezione critica e pone **SPAZIO_DISP = 0** (**s.value--**), mentre in uscita pone **MSG_DISP = 1**; alla seconda produzione, poi, il processo viene sospeso finché non si ha **SPAZIO_DISP = 1**, per effetto di una signal dal consumatore.

```

void Consumatore(msg * ptr_sh, int sem){

    msg mess;
    Wait_Sem(sem, MSG_DISP);
    // Prelievo del messaggio
    mess = *ptr_sh;
    printf("Messaggio letto: <%d> \n", mess);
    Signal_Sem(sem, SPAZIO_DISP);
}
  
```

Si supponga che il produttore abbia già depositato il valore (quindi, `MSG_DISP = 1`); **il consumatore procede a consumare**, ponendo `MSG_DISP = 0` (`s.value--`), mentre **in uscita pone `SPAZIO_DISP = 1`** (`s.value++`), attivando un produttore in attesa.

Un’alternativa alla gestione del problema produttore/consumatore ad unico buffer può essere l’**implementazione di un vettore di buffer** (problema produttore/consumatore a vettore di buffer), che consente ai messaggi di essere gestiti tramite una coda circolare per la quale il produttore si sospende se i buffer sono tutti pieni e il consumatore se sono tutti vuoti ed è sufficiente che anche solo un buffer sia disponibile per far riprendere le produzioni; inoltre, essendo il vettore una coda circolare, tutte le operazioni sono gestite in maniera circolare:



La coda in questione è implementata mediante i seguenti campi:

- **buffer [DIM]**, array di elementi di tipo `msg` (tipo del messaggio depositato dai produttori) contenente i valori prodotti;
- **testa**, tipo intero che indica la posizione del primo buffer libero in testa, `buffer[testa]`, ossia il primo buffer disponibile per la memorizzazione di un messaggio (quindi, l’elemento prodotto più recentemente sarà in posizione `buffer[testa-1]`);
- **coda**, tipo intero che indica la posizione dell’elemento prodotto meno recentemente, `buffer[coda]`, da accedere alla prossima consumazione.

Si consideri che, **così come il buffer in sé e per sé, anche le variabili di testa e coda devono essere condivise**.

Per la sincronizzazione dei processi sono stati utilizzati due semafori: **SPAZIO_DISP**, che indica la presenza di spazio disponibile in coda per la produzione di un messaggio, e **NUM_MESS**, che indica il numero di messaggi presenti in coda; **SPAZIO_DISP** ha valore iniziale **DIM** e **NUM_MESS** ha valore iniziale **0**.

```
void Produttore(int* testa, msg* buffer, int sem) {
    Wait_Sem(sem, SPAZIO_DISP);
    // Produzione di valore_prodotto . . .
    buffer[*testa] = valore_prodotto;
    *testa = (*testa + 1) % DIM;      //gestione circolare
    Signal_Sem(sem, NUM_MESS);      //nelem=nelem+1
}
```

```

void Consumatore(int* coda, msg* buffer, int sem){

    msg mess;

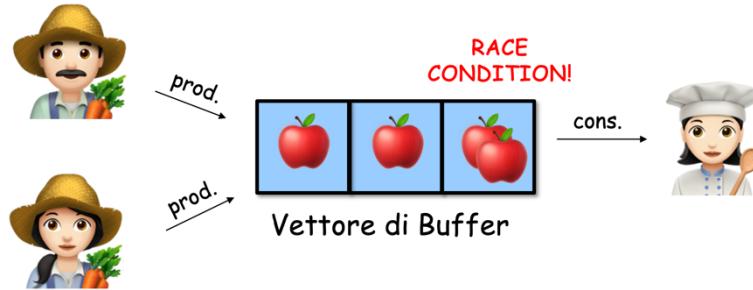
    Wait_Sem(sem, NUM_MESS);

    // Consumo
    mess = buffer[coda];
    coda = (coda + 1) % DIM; // gestione circolare
    printf("Messaggio letto: <%d> \n", mess);

    Signal_Sem(sem, SPAZIO_DISP); // nelem=nelem-1
}

```

Con questa implementazione, il produttore e il consumatore non accedono mai contemporaneamente alla stessa posizione della coda dei messaggi. Tuttavia, è una soluzione adatta solo nel caso in cui si ha un solo processo produttore e un solo processo consumatore; infatti, se vi sono due buffer liberi e due produttori iniziano contemporaneamente a produrre, si verifica una race condition:



Nell'ipotesi in cui vi siano più produttori e più consumatori che accedono allo stesso buffer, le operazioni di **deposito** e **prelievo** devono essere eseguite in mutua esclusione ed essere, quindi, **programmate come sezioni critiche**. A tal fine, è necessario introdurre due nuovi semafori: **MUTEX_P**, per le operazioni di **produzione**, e **MUTEX_C**, per le operazioni di **consumo**, entrambi inizializzati a 1.

```

void Produttore(int* testa, msg* buffer, int sem) {

    Wait_Sem(sem, SPAZIO_DISP);
    Wait_Sem(sem, MUTEX_P); // inizio sez. critica

    // Produzione di valore_prodotto . . .
    buffer[testa] = valore_prodotto;
    testa = (testa + 1) % DIM; // gestione circolare

    Signal_Sem(sem, MUTEX_P); // fine sez. critica
    Signal_Sem(sem, NUM_MESS); // nelem=nelem+1
}

```

```

void Consumatore(int* coda, msg* buffer, int sem) {
    msg mess;

    Wait_Sem(sem, NUM_MESS);
    Wait_Sem(sem, MUTEX_C); // inizio sez. critica

    // Consumo
    mess = buffer[coda];
    coda = (coda + 1) % DIM; // gestione circolare
    printf("Messaggio letto: <%d> \n", mess);

    Signal_Sem(sem, MUTEX_C); // fine sez. critica
    Signal_Sem(sem, SPAZIO_DISP); // nelem=nelem-1
}

```

Inoltre, per una corretta sincronizzazione, è necessario che gli indici di testa e di coda siano condivisi tra i processi (ad esempio, quando un processo incrementa testa, un altro processo scriverà alla posizione successiva, corretta); pertanto, è possibile collocare sia il vettore di buffer sia gli indici di testa e coda nella stessa memoria condivisa:

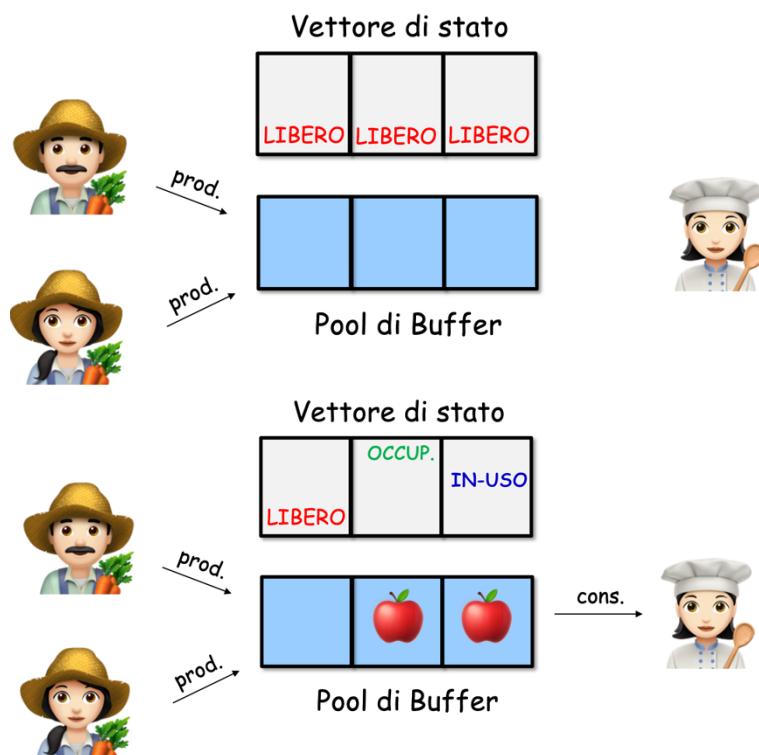
```

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

int id_shm = shmget(..., sizeof(prod_cons), ...);
prod_cons * p = shmat(...);

```

La soluzione appena presentata penalizza produttori o consumatori “veloci” in presenza di corrispettivi “lenti”; ciò può accadere, ad esempio, quando i messaggi prodotti hanno una dimensione variabile. Una possibile soluzione a questo problema consiste nell’utilizzo di un pool di buffer gestito mediante un vettore ausiliario di stato; l’accesso al vettore di stato per acquisire un buffer è in mutua esclusione ma, dopo averlo fatto, produttori e consumatori procedono in concorrenza.



La **mutua esclusione**, tuttavia, è **limitata solo al vettore di stato** (è un'operazione veloce), mentre il pool di buffer non è acceduto in mutua esclusione, i processi lavorano su buffer diversi. Con questo accorgimento, un processo “lento” non penalizza uno “veloce”, l'ordine non è più circolare ma dipeso dalla velocità dei processi; ad esempio, se il secondo produttore finisce prima, il consumatore preleva dal secondo buffer invece che dal primo.

La gestione del pool di buffer avviene mediante due vettori:

- **buffer [DIM]**, array di elementi di tipo msg (tipo del messaggio depositato dai produttori) contenente i valori prodotti;
- **stato [DIM]**, array di elementi di tipo intero il cui i – esimo elemento, **stato[i]**, può assumere uno tra i seguenti valori:
 - **VUOTO**, indica che la cella **buffer[i]** non contiene alcun valore prodotto;
 - **PIENO**, indica che la cella **buffer[i]** contiene un valore prodotto e non ancora consumato;
 - **IN_USO**, indica che la cella **buffer[i]** contiene un valore in uso da un processo attivo, consumatore o produttore.

Per la mutua esclusione al buffer di stato, si utilizzano due mutex, **MUTEXP** e **MUTEXC**, rispettivamente per i produttori e per i consumatori, entrambi **inizializzati a 1**. I soliti due semafori, **SPAZIO_DISP** e **MSG_DISP**, usati per la sincronizzazione, sono **inizializzati**, rispettivamente, a **DIM** (o al numero di produttori, se minore o uguale a **DIM**) e a **0**.

```
void Produttore(int* stato, msg* buffer, int sem) {
    int indice = 0;
    Wait_Sem(sem, SPAZIO_DISP); // attende spazio

    Wait_Sem(sem, MUTEXP); // sez. critica
    // trova il primo elemento VUOTO in "indice"
    while (indice<DIM && stato[indice]!=VUOTO)
        indice++;
    stato[indice]=IN_USO;
    Signal_Sem(sem, MUTEXP); // sez. critica

    // Produzione di valore_prodotto . . .
    buffer[indice]=valore_prodotto;
    stato[indice]=PIENO;

    Signal_Sem(sem, MSG_DISP); // segnala i cons

}

void Consumatore(int* stato, msg* buffer, int sem) {
    int indice = 0;
    Wait_Sem(sem, MSG_DISP); // attende messaggio

    Wait_Sem(sem, MUTEXC); // sez. critica
    // determina l'indice del primo elemento PIENO
    while (indice<DIM && stato[indice]!=PIENO)
        indice++;
    stato[indice]=IN_USO;
    Signal_Sem(sem, MUTEXC); // sez. critica

    // consumo del messaggio
    msg valore_consumato = buffer[indice];
    stato[indice]=VUOTO;

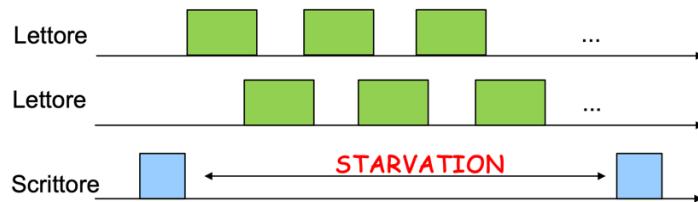
    Signal_Sem(sem, SPAZIO_DISP); // segnala i prod
}
```

Con la soluzione a pool di buffer, i produttori (o i consumatori) accedono alla sezione critica gestita da **MUTEXP** (o **MUTEXC**) solo per acquisire una cella tramite ricerca nel vettore di stato; una volta acquisita la cella, possono procedere concorrentemente nella produzione (o nel consumo), in modo che i produttori “veloci” possano terminare prima e segnalare prima i consumatori (e viceversa).

Il secondo dei problemi di cooperazione è il problema lettore/scrittore, che parte dal presupposto per il quale esistono due categorie di processi: i processi lettori, che leggono un messaggio su di una risorsa condivisa, e i processi scrittori, che scrivono il messaggio sulla risorsa condivisa. I vincoli che questa categorizzazione induce sono tre:

- I processi lettori possono accedere contemporaneamente alla risorsa;
- I processi scrittori accedono solo esclusivamente alla risorsa (si sospendono se c’è un altro processo);
- I lettori e gli scrittori si escludono mutuamente dall’uso della risorsa.

L’operazione di lettura è protetta dalle procedure **Inizio_Lettura()** e **Fine_Lettura()**, mentre l’operazione di scrittura da **Inizio_Scrittura()** e **Fine_Scrittura()**. Questa configurazione fa sì che un processo lettore attenda solo se la risorsa è occupata da un processo scrittore, il quale può accedere alla risorsa solo se questa è libera; tuttavia, la strategia di sincronizzazione appena mostrata può indurre i processi scrittori in starvation (gli scrittori si sospendono sia se c’è attivo uno scrittore sia se c’è attivo un lettore, mentre quest’ultimi non si sospendono quando ci sono più omologhi attivi). La presenza di un lettore permette solo ad altri lettori di entrare in attività ma non lo permette agli scrittori; pertanto, finché c’è un lettore attivo, gli scrittori non possono entrare in gioco:



Una variabile condivisa, **NUM_LETT**, va ad indicare il numero di lettori che contemporaneamente accedono ad una risorsa, alla quale gli scrittori possono accedere (uno alla volta) solo quando tale variabile è 0. Per garantire questo comportamento, si utilizzano due semafori: **MUTEXL**, per gestire l’accesso alla variabile **NUM_LETT** in mutua esclusione da parte dei lettori, e **SYNCH**, per garantire la mutua esclusione tra i processi lettori e scrittori e tra i soli processi scrittori, entrambi inizializzati a 1.

L’algoritmo per la descrizione di un processo lettore viene diviso in due funzioni, **Inizio_Lettura()** e **Fine_Lettura()**, come anticipato in precedenza; queste due funzioni verranno chiamate, rispettivamente, subito prima e subito dopo l’operazione di lettura stessa:

```
void Processo_Lettore(int sem) {
    Inizio_Lettura(sem);
    // la lettura
    ...
    Fine_Lettura(sem);
}
```

Mentre le singole funzioni:

```
void Inizio_Lettura(int sem) {
    Wait_Sem(sem, MUTEXL);
    Num_Lettori++;
    if (Num_Lettori==1) Wait_Sem (sem, SYNCH);
    Signal_Sem(sem, MUTEXL);
}

void Fine_Lettura(int sem) {
    Wait_Sem(sem, MUTEXL);
    Num_Lettori--;
    if (Num_Lettori==0) Signal_Sem (sem, SYNCH);
    Signal_Sem(sem, MUTEXL);
}
```

Se non ci sono scrittori attivi, tutti i processi lettori eseguono senza bloccarsi; tuttavia, se il buffer è già occupato da un processo scrittore, questo pone NUM_LETT a 1 e si pone in attesa su SYNCH, riattivando e permettendo agli altri la lettura. Un eventuale scrittore, però, si riattiva solo quando l'ultimo lettore termina di operare, introducendo nel sistema la possibilità di starvation.

La starvation, però, non è tipica degli scrittori; infatti, introducendo la variabile NUM_SCRITT è possibile avere un comportamento analogo. In questo caso, concorrono quattro semafori, tutti inizializzati a 1:

- **MUTEXL**, per gestire l'accesso alla variabile NUM_LETT in mutua esclusione da parte dei lettori;
- **MUTEXS**, per gestire l'accesso alla variabile NUM_SCRITT in mutua esclusione da parte degli scrittori;
- **MUTEX**, per gestire l'accesso in mutua esclusione alla risorsa condivisa da parte degli scrittori;
- **SYNCH**, per garantire la mutua esclusione tra i processi lettori e scrittori.

```
void Inizio_Scrittura(int sem) {
    Wait_Sem(sem, MUTEXS);
    Num_Scrittori++;
    if (Num_Scrittori==1) Wait_Sem(sem, SYNCH);
    Signal_Sem(sem, MUTEXS);

    Wait_Sem(sem, MUTEX);
}

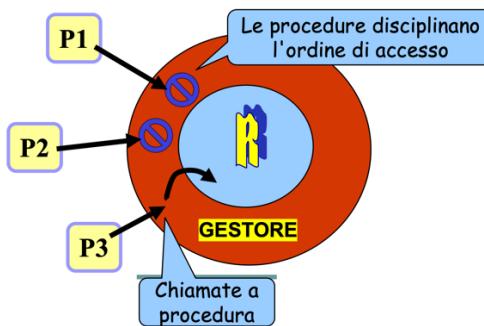
void Fine_Scrittura(int sem) {
    Signal_Sem(sem, MUTEX);
    Wait_Sem(sem, MUTEXS);
    Num_Scrittori--;
    if (Num_Scrittori==0) Signal_Sem (sem, SYNCH);
    Signal_Sem(sem, MUTEXS);
}
```

L'algoritmo di scrittura è uguale a quello di lettura, con l'aggiunta di un'ulteriore sezione critica (Wait_Sem in coda all'inizio e Signal_Sem in testa alla fine), necessaria affinché gli scrittori si escludano tra di loro.

I MONITOR

I **semafori** forniscono strumenti primitivi ma potenti e flessibili per la realizzazione di politiche di mutua esclusione e di algoritmi di sincronizzazione tra processi; tuttavia, potrebbe essere difficile produrre un programma correttamente funzionante usando i semafori, dal momento in cui `Wait_Sem()` e `Signal_Sem()` possono essere sparpagliate nel programma, rendendolo difficile da leggere ed interpretare. In aiuto intervengono i **monitor**, entità funzionalmente equivalenti ai semafori ma più facili da utilizzare.

Il monitor è un costrutto sintattico, cioè un **insieme di operazioni abbinato ad una struttura dati** (la risorsa) **condivisa tra processi**; dal punto di vista sintattico, il monitor è molto simile alle classi ma viene utilizzato per la gestione delle risorse condivise. L'impiego dei monitor facilita la programmazione concorrente, permettendo di creare politiche di accesso alle risorse condivise in maniera più agevole dei semafori.



Il monitor, così descritto, è un **modulo software composto di una o più procedure, una sequenza di inizializzazione e dei dati locali, caratterizzato dalle seguenti proprietà**:

- Le variabili locali e le procedure locali sono accessibili solo dalle procedure del monitor e non da altri agenti esterni (**principio di incapsulazione**);
- **Un processo entra in un monitor invocando una delle sue procedure;**
- Solo un processo alla volta può star eseguendo nel monitor, ogni altro processo che vuole entrarvi è bloccato in attesa della disponibilità del monitor stesso (**principio di mutua esclusione**, a priori implementato a differenza dei semafori).

Si può notare che **le prime due proprietà sono una reminiscenza della programmazione orientata agli oggetti**, nei confronti della quale i monitor si rivolgono molto pur senza cadervi esplicitamente; infatti, un Sistema Operativo implementato ad oggetti può facilmente essere realizzato con l'impiego di monitor in quanto oggetti con determinate caratteristiche.

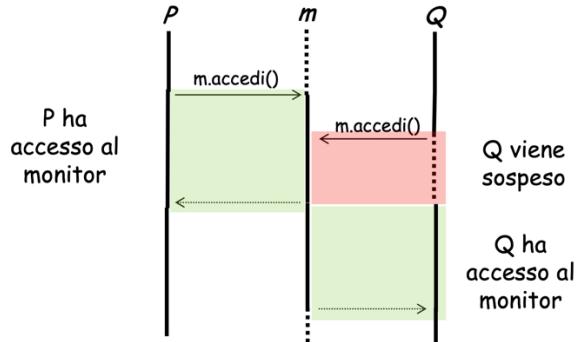
Quindi, in sé e per sé il **monitor si configura come un tipo di dato astratto**, caratterizzato da:

- **Risorsa**, variabile membro, privata;
- **Stato della risorsa**, variabile membro, privata;
- **Funzioni membro, per uso interno**, private;
- **Funzioni membro, per l'accesso alla risorsa**, pubbliche.

La **politica di accesso** ad una risorsa impone che:

- **Un solo processo alla volta può avere accesso alla risorsa condivisa**, per competizione;
- **I processi seguano un determinato ordine di accesso alla risorsa**, per cooperazione.

Le **funzioni pubbliche** del monitor sono eseguite in modo mutuamente esclusivo a priori, senza che, come per i semafori, si debba implementare questa funzionalità.



I metodi pubblici di un monitor si definiscono con librerie di funzioni e con parole chiave del linguaggio di programmazione, nonostante alcuni di essi hanno già dei monitor implementati (come Java o Competitive Pascal).

Per quanto riguarda le **politiche di cooperazione**, si deve considerare un **cambio di paradigma rispetto ai semafori**; infatti, i processi si sospongono se non è verificata una “condizione logica” di accesso. La condizione logica in questione è definita come **variabile condition** (tipo astratto), interna al monitor e dotata di due metodi:

- **wait_cond()**, che sospende il processo chiamante;
- **signal_cond()**, riprende l'esecuzione del processo che è stato sospeso con il metodo precedente.

```

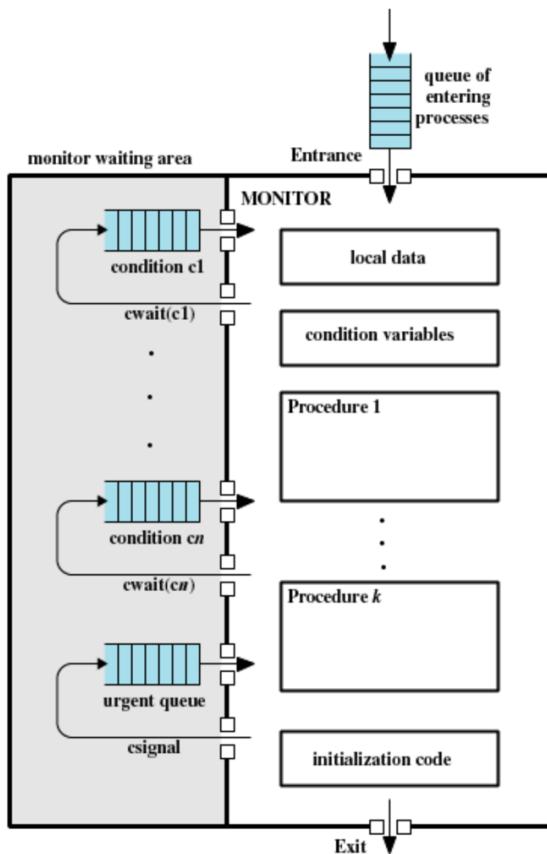
Monitor M {
    var_cond x;
    var_cond y;

    void metodo1() {
        enter_monitor();
        if/while (! condizione_logica) {
            x.wait_cond();
        }
        // operazioni su risorsa ...
        y.signal_cond();
        leave_monitor();
    }
    ...
}
  
```

Per ottenere un **ordine di accesso**, e quindi implementare **logiche di sincronizzazione** (non definite a priori), vanno osservate le seguenti disposizioni:

- In ogni metodo del monitor, il processo chiamante controlla se è soddisfatta una **condizione logica**;
- Se la condizione non è verificata, il processo chiamante viene sospeso;
- Si consente l'accesso ad un altro processo, che può eventualmente risvegliare il processo sospeso.

Una panoramica grafica del costrutto monitor è la seguente:



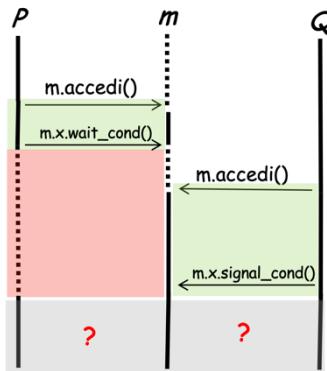
Le procedure sono eseguite in mutua esclusione, cioè da al più un processo, e controllano, tramite le variabili locali, se la condizione di sincronizzazione è valida (ad esempio, la condizione di buffer vuoto); se tale condizione è valida, il processo completa l'esecuzione e libera il monitor.

Finora ci si è limitati a dire che semafori e monitor differiscono perché questi ultimi implementano la mutua esclusione a priori; tuttavia le **differenze non si limitano a questa** ed è importante notare che le **funzioni di `wait_condition()` e `signal_condition()` sono diverse da quelle di `wait_sem()` e `signal_sem()`**, così come le **variabili `condition` sono diverse dai semafori stessi**. I motivi sono i seguenti:

- La `wait_cond()` sospende sempre il processo chiamante, mentre nei semafori con la `wait_sem()` la sospensione era condizionata dalla variabile interna del semaforo;
- La `signal_cond()` non ha alcun effetto se non vi è alcun processo in attesa sulla variabile `condition`.

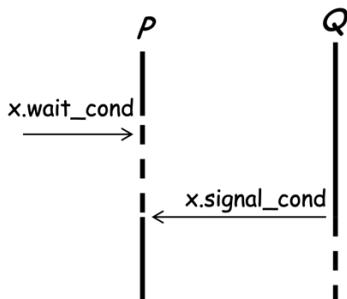
Inoltre, essendo le funzioni di `signal` and `wait` confinate al monitor stesso, è più agevole verificare che la sincronizzazione sia stata effettuata correttamente e trovare eventuali bug; con i semafori, l'accesso alla risorsa è corretto solo se tutti i processi che accedono a tale risorsa sono programmati correttamente, mentre per i monitor i processi sono trasparenti alla correttezza dell'accesso.

Si consideri il seguente grafico:



Inizialmente, **P** ha accesso per primo al monitor ma la risorsa non è ancora pronta e, quindi, si sospende; **Q** ha accesso al monitor e può aggiornare lo stato della risorsa. Quando la risorsa diventa pronta, il processo **Q** riattiva **P**, il quale ritorna ad essere pronto. Problema: **P** e **Q** non possono eseguire entrambi, si violerebbe la mutua esclusione tipica dei monitor. A questo problema non esiste una soluzione univoca; diversi sistemi attribuiscono comportamenti (agendo quindi sulla semantica) diversi alle primitive `wait_cond()` e `signal_cond()`.

Una prima soluzione, detta **signal and wait**, prevede che **il processo segnalato P riprenda immediatamente l'esecuzione e che il processo segnalante Q venga sospeso** (al fine di evitare che possa modificare nuovamente la condizione di sincronizzazione):



Secondo questo approccio, **il processo segnalato è il primo ad eseguire e, al risveglio, ha la certezza di trovare verificata la condizione che attendeva**. Lo schema tipico dell'invocazione di una `wait_cond()` è all'interno di un `if`:

```

if (!B) {           // B = condizione di sincronizzazione
    cv.wait_cond();   // cv = var. condition, abbinata a B
}
<...accesso alla risorsa...>
  
```

Quindi, **il processo P entra per primo, prematuramente** (la condizione di sincronizzazione non è ancora valida), e (quindi) si sospende; il processo **Q attende l'uscita di P e compete con altri processi per rientrare nel monitor**.

Nella **soluzione di Hoare**, il processo **Q ha la precedenza sugli altri processi**, che attendono di entrare nel monitor; questa precedenza è rappresentata da un'apposita coda, detta **coda dei processi urgenti** (`urgent_queue`) separata dal mutex. Questa soluzione può essere anche riferita con il nome di **signal and urgent wait** e si configura come una **particularizzazione della signal and wait**.

Una seconda soluzione prende il nome di **wait and notify** (detta anche **soluzione di Lampson/Redell**) e privilegia il processo segnalante rispetto al segnalato; pertanto, il processo Q segnalante prosegue la sua esecuzione, mantenendo l'accesso esclusivo al monitor. Il processo segnalato P non viene attivato subito, ma è trasferito alla coda di ingresso nel monitor, mentre Q continua ad eseguire; l'accesso di P al monitor avviene solo dopo l'uscita di Q e la contesa con altri processi (come accadeva a Q nella signal and wait). Durante il tempo in cui P è in attesa di entrare, il processo Q (o un terzo K appena entrato) possono modificare la risorsa, invalidando la condizione di sincronizzazione e costringendo il processo P ad una nuova attesa; pertanto, anche se risvegliato, il processo P non ha la certezza che la condizione sia verificata e deve effettuare un controllo prima di poter proseguire. Lo schema tipico dell'invocazione di una `wait_cond()` è all'interno di un `while`:

```
while (!B) {  
    cv.wait_cond();  
}  
<...accesso alla risorsa...>
```

*// È possibile che wait_cond() venga
// chiamata più volte prima di accedere*

`signal_cond()` riattiva al più un processo ma è possibile risvegliare tutti i processi sospesi sulla stessa variabile `condition` utilizzando la variante `signal_all()`. Tutti i processi risvegliati vengono messi nella `entry_queue`, dalla quale uno alla volta possono rientrare nel monitor ricontrollando la condizione di sincronizzazione e sospendendosi con `wait_cond()` se necessario.

Volendo confrontare le due soluzioni proposte, **signal and wait** e **wait and notify**, la semantica della prima richiede che venga chiamata precisamente quando il processo segnalato deve essere svegliato, mentre la semantica della seconda (e `signal_all()`) è più robusta, dal momento in cui il processo segnalante può chiamarla anche quando non è sicuro di se/quali processi risvegliare (sono i processi risvegliati a controllare se possono eseguire o se devono sospendersi).

I sistemi UNIX non forniscono di serie delle chiamate di sistema per realizzare i monitor ma rimandano ai linguaggi di programmazione; si consideri che le ultime versioni del C++ forniscono apposite classi, `std::mutex` e `std::condition_variable`, per la realizzazione dei monitor. In questa sede, i monitor saranno realizzati in due modi: nei programmi multi – processo, verranno implementati utilizzando una libreria custom basata internamente su semafori e su memoria condivisa UNIX, mentre nei programmi multi – thread sarà utilizzata la libreria PThreads.

Si procede con un **esempio di problema produttori – consumatori su buffer singolo tramite monitor** implementati con semantica `signal and wait`:

```

struct buff {
    msg buffer;
    int buffer_pieno=0;
    int buffer_vuoto=1;
    Monitor m;
    // 2 cond: CV_PROD, CV_CONS
}

void Produzione (buff* b, msg mx) {
    enter_monitor (&b->m);
    if (b->buffer_pieno==1)
        wait_cond(&b->m, CV_PROD);

    b->buffer = mx;
    b->buffer_pieno = 1;
    b->buffer_vuoto = 0;

    signal_cond(&b->m, CV_CONS);
    leave_monitor (&b->m);
}
...
msg Consumo (buff* b) {
    enter_monitor (&b->m);
    msg mx;
    if (b->buffer_vuoto==1)
        wait_cond(&b->m, CV_CONS);

    mx = b->buffer;
    b->buffer_vuoto = 1;
    b->buffer_pieno = 0;

    signal_cond(&b->m, CV_PROD);
    leave_monitor (&b->m);
    return mx;
}

```

Si noti che:

- La scelta delle variabili di stato (buffer_pieno e buffer_vuoto) e il numero di variabili condition (CV_PROD e CV_CONS) sono a discrezione del programmatore;
- A differenza dei semafori, la condizione di sincronizzazione appare chiaramente nel programma, mentre wait_cond() nella funzione Produzione() è un semplice meccanismo di sospensione:
 - Con i semafori, wait_sem() includeva al suo interno sia il meccanismo di sospensione sia una politica basata sulla variabile intera interna;
- È importante aggiornare la variabile contatore insieme a testa/coda (e, in generale, ogni volta che si modifica lo stato della risorsa);
- b->contatore == 0 sarebbe potuto essere anche b->coda == b->testa;
- In alternativa a b->contatore == N, il controllo sullo stato di “ pieno ” sarebbe potuto essere anche b->coda == (b->testa - 1) %DIM;

Con pool di buffer con stato, invece:

```
struct buff {
    msg buffer[N];
    enum stato_buf { LIBERO, OCCUPATO, INUSO };
    stato_buf stato[N];
    int numero_occupati=0;
    int numero_liberi=N;
    Monitor m;
    // 2 cond: CV_PROD, CV_CONS
}

void Produzione (buff* b, msg mx) {
    int i = IniziaProduzione(b);
    // ...operazione lenta...
    buffer[i] = mx;
    FineProduzione(b,i);
}
...
msg Consumo (buff* b) {
    msg mx;
    int i = IniziaConsumo(b);
    // ...operazione lenta...
    mx = buffer[i];
    FineConsumo(b,i);
    return mx;
}
...
```

Si noti che:

- I processi producono e consumano senza tenere occupato l'oggetto – monitor, permettendo operazioni in parallelo su buffer distinti;
- Le funzioni IniziaProduzione, IniziaConsumo, FineProduzione e FineConsumo entrano temporaneamente nel monitor, modificano le variabili di stato ed escono (presto) dal monitor.

```

...
int IniziaProduzione (buff* b){
    enter_monitor(&b->m);
    int i = 0; //indice
    if (b->numero_liberi==0)
        wait_cond(&b->m, CV_PROD);

    while(i<N && b->stato[i]!=LIBERO)
        i++;

    b->stato[i] = INUSO;
    b->numero_liberi--;

    leave_monitor(&b->m);
    return i;
}

void FineProduzione(buff*b,int i){
    enter_monitor(&b->m);
    b->stato[i] = OCCUPATO;
    b->numero_occupati++;
    signal_cond(&b->m, CV_CONS);
    leave_monitor(&b->m);
}

int IniziaConsumo (buff* b){
    enter_monitor(&b->m);
    int i = 0;
    if (b->numero_occupati==0)
        wait_cond(&b->m, CV_CONS);

    while(i<N && b->stato[i]!=OCCUPATO)
        i++;

    b->stato[i] = INUSO;
    b->numero_occupati--;

    leave_monitor(&b->m);
    return i;
}

void FineConsumo (buff* b, int i){
    enter_monitor(&b->m);
    b->stato[i] = LIBERO;
    b->numero_liberi++;
    signal_cond(&b->m, CV_PROD);
    leave_monitor(&b->m);
}

```

Si noti che:

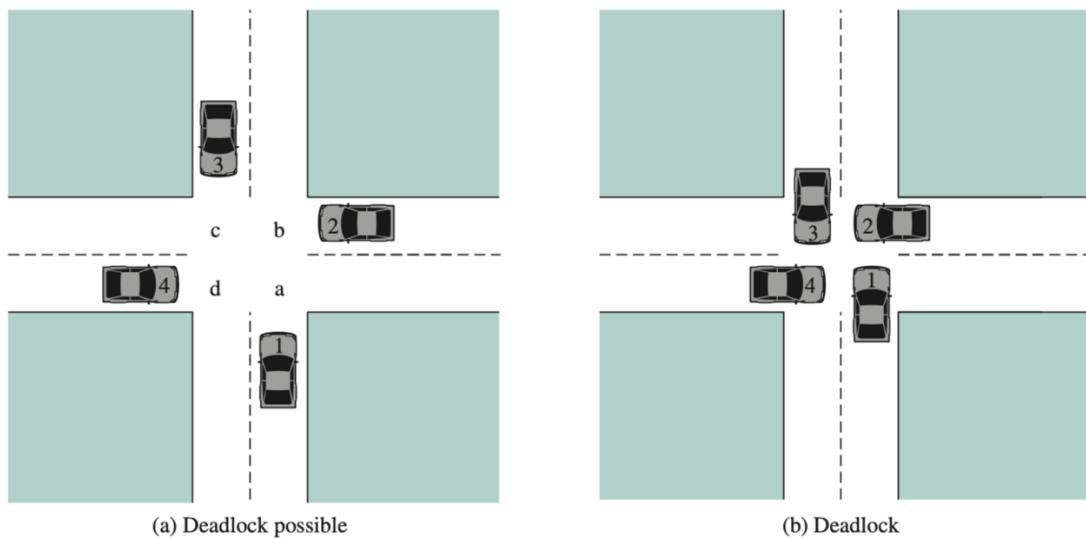
- Sia la lettura che la scrittura sono svolte al di fuori del monitor:
 - Lettura consente accesso in parallelo a più lettori;
 - Scrittura fa sospendere altri processi scrittori su una variabile condition se c’è già uno scrittore attivo, in modo da poter gestire la starvation;
- Ogni lettore sveglia un altro lettore;
- Lo scrittore tenta dapprima di riattivare eventuali altri scrittori in attesa (se presenti) per bilanciare la starvation, altrimenti riattiva i lettori;

- In InizioLettura, non c'è più bisogno della signal_cond(CV_LETT) grazie alla signal_all() in FineScrittura;
 - Lo scrittore, così, riattiva tutti i lettori in attesa (ovvero se non ci sono altri scrittori in attesa)

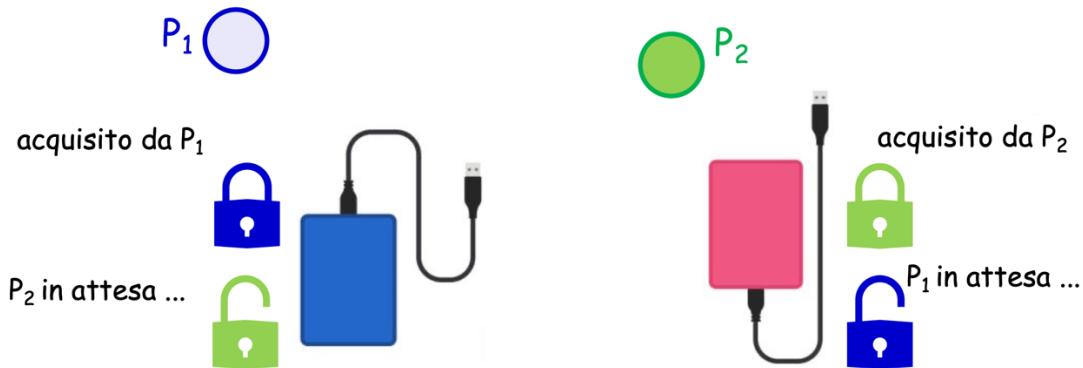
IL PROBLEMA DEL DEADLOCK

Per **deadlock** si intende una **condizione di blocco permanente di un gruppo di processi in competizione per le risorse di sistema** e descrive un **problema complesso e di rilievo**, che può provocare **gravi malfunzionamenti**.

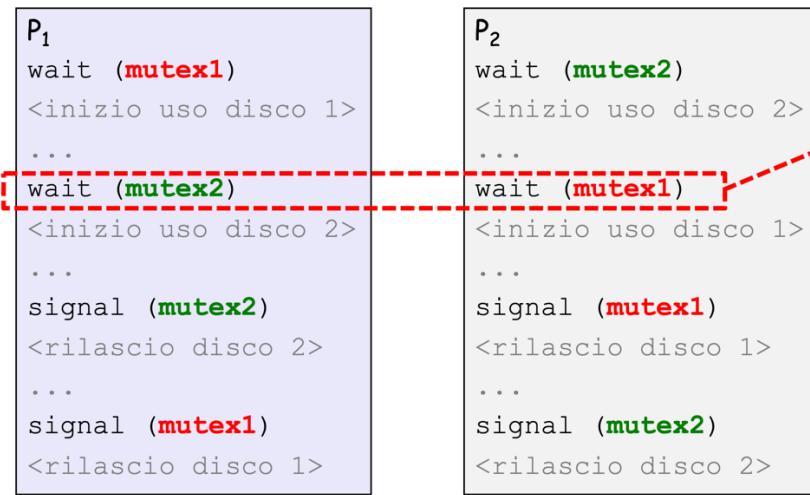
Un **incrocio stradale** è un buon esempio per comprendere come, **in assenza di regole di precedenza**, si può generare la possibilità di deadlock; **ogni auto**, che può essere vista come un **processo**, ha da attraversare due **quadranti**, ognuno dei quali è possibile vederlo come una **risorsa**:



Un esempio più contestuale di deadlock è quello della **copia di un file**; si consideri un **sistema dotato di due dischi esterni e due processi P_1 e P_2** che copiano un grosso file da un disco all'altro, supponendo sia necessaria la mutua esclusione. P_1 e P_2 “acquisiscono” l'esclusiva ad uno dei due dischi ma ciascuno ha bisogno di accedere anche all'altro acquisito dall'altro processo, generando così un **blocco permanente**.



Per la sincronizzazione, si usano due semafori: `mutex1` e `mutex2`, entrambi inizializzati ad 1. Osservando gli pseudocodici di entrambi i processi, si noti che sia P_1 che P_2 possono sospendersi sulla `wait`:



In particolare, i due processi rimangono bloccati se accade la sequenza di azioni seguente:

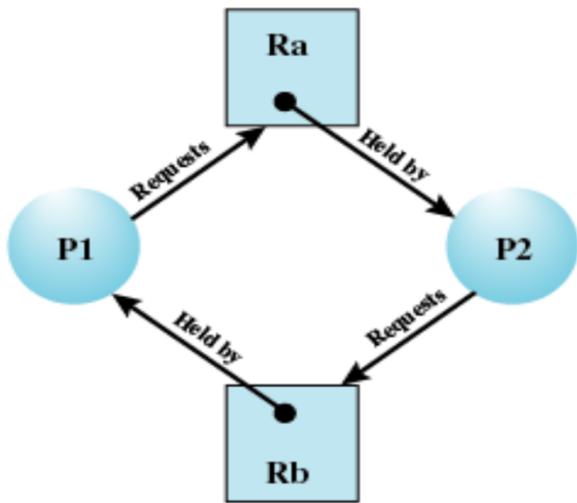
- P_1 esegue la `wait (mutex1)` ed acquisisce il disco 1;
- P_2 esegue la `wait (mutex2)` ed acquisisce il disco 2;
- P_1 esegue la `wait (mutex2)` e si blocca in attesa di P_2 ;
- P_2 esegue la `wait (mutex1)` e si blocca in attesa di P_1 .

Il deadlock può verificarsi saltuariamente, in base alla **velocità relativa di esecuzione dei processi** e, quindi, **non in maniera deterministicamente predeterminata** (il codice sopra descritto, ad una singola esecuzione, può produrre o meno deadlock). Prima di proseguire, si chiarisca che **il deadlock non è né la starvation né una sua variante**, sebbene le definizioni appaiano simili; la **starvation** denota una condizione di **attesa indefinita**, cioè **la cui durata non è nota a priori ma che prevede comunque la possibilità di esecuzione del processo che la subisce**, mentre il **deadlock** denota una condizione di **attesa infinita**, cioè **la cui durata non è finita e che non prevede in alcun modo la possibilità di esecuzione del processo che lo subisce**.

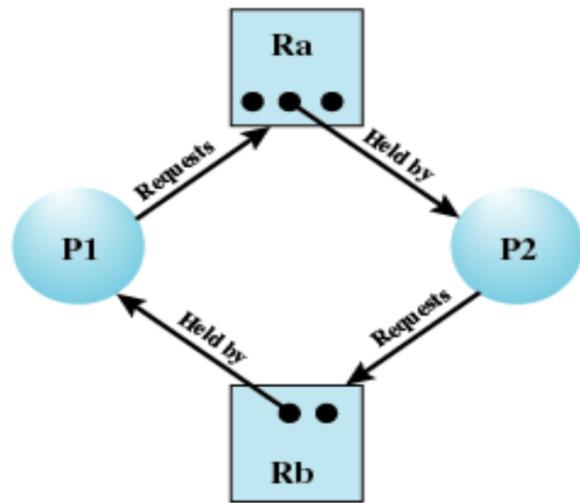
Per verificare se un sistema soffre o può soffrire di deadlock è utile l'impiego del **grafo di assegnazione delle risorse**; si tratta di **un insieme di nodi V e archi E assegnati secondo i seguenti criteri**:

- L'insieme di nodi V è partizionato in:
 - $P = \{P_0, P_1, \dots, P_n\}$, l'insieme di tutti i processi del sistema;
 - $R = \{R_0, R_1, \dots, R_m\}$, l'insieme di tutti i tipi di risorsa nel sistema;
 - Per una questione grafica, i nodi risorsa sono descritti anche con il numero di istanze di cui sono dotati
- Gli archi sono categorizzati in:
 - Arco (orientato) di richiesta $P_i \rightarrow R_j$;
 - Arco (orientato) di assegnazione $R_j \rightarrow P_i$;

Un esempio di grafo di assegnazione delle risorse è il seguente:



(c) Circular wait



(d) No deadlock

A partire dal grafo, **il deadlock non si verifica se non sono presenti dei cicli**, ovvero se non sono possibili situazioni di stallo; al contrario, **se il grafo contiene un ciclo e:**

- Se c'è solo un'istanza per tipo di risorsa, si verifica una **situazione di stallo**;
- Se vi sono più istanze per tipo di risorsa, c'è (solo) la **possibilità che si verifichi una situazione di stallo**.

Si consideri che, quando si parla di deadlock, **le risorse coinvolte sono risorse riutilizzabili**, ovvero risorse che **devono sempre essere utilizzate da un processo alla volta ma che non vengono distrutte quando il processo che le sta usando termina la propria esecuzione**, rendendole **possedibili da un nuovo processo** per un numero illimitato di volte. Questa distinzione va fatta perché **in un sistema possono essere presenti anche risorse non riutilizzabili** (che vengono distrutte alla terminazione dell'esecuzione del processo che le possiede) o **consumabili** (cioè riusabili un numero limitato di volte). Anche queste risorse seguono il paradigma **Richiesta – Utilizzo** quando un processo desidera assumerne il controllo ma, a differenza del caso generale, **è necessario anche il Rilascio della risorsa dopo l'Utilizzo**, in modo da **permettere un nuovo ciclo Richiesta – Utilizzo – Rilascio da parte di un altro processo**.

La gestione dei **deadlock** in un sistema può avvenire secondo **tre modalità**:

- **Prevenzione del deadlock**, rendendo impossibile il verificarsi delle condizioni per un deadlock al costo di un basso utilizzo delle risorse;
- **Evitamento del deadlock**, limitando la possibilità di entrare in uno stato di deadlock nonostante ne siano consentite le condizioni;
- **Rilevamento del deadlock**, risolvendo il problema una volta che il sistema è entrato in stato di deadlock (ripristino del sistema).

La maggioranza dei **Sistemi Operativi general – purpose**, inclusi UNIX e Windows, **non dispone di una soluzione generale ed efficiente al problema del deadlock**.

All'atto pratico, **ci sono possibilità di deadlock se sussistono le seguenti condizioni**:

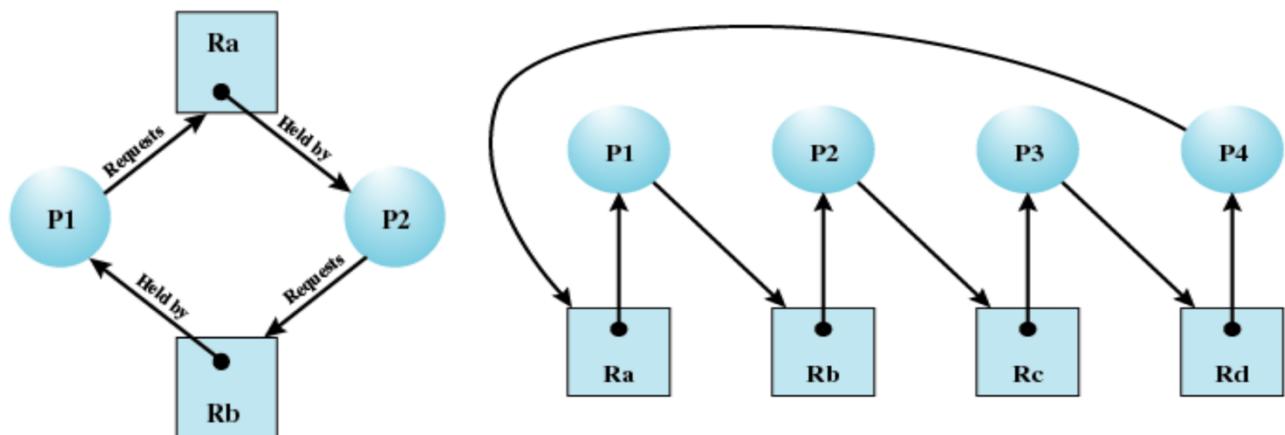
- **Mutua esclusione** (un solo processo alla volta può usare una risorsa);

- **Impossibilità di prelazione** (una risorsa può essere rilasciata solo volontariamente dal processo che la possiede al termine del suo compito);
- **Possesso ed attesa** (un processo che possiede almeno una risorsa attende di acquisire ulteriori risorse già possedute da altri processi).

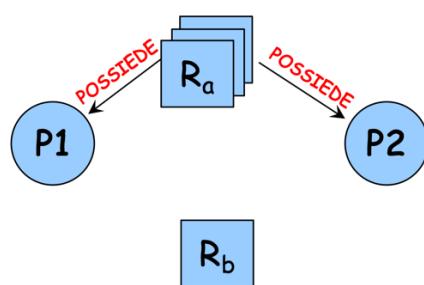
Mentre il **deadlock** esiste se sussistono le seguenti condizioni:

- **Mutua esclusione;**
- **Impossibilità di prelazione;**
- **Possesso ed attesa;**
- **Attesa circolare** (esiste un insieme di processi in attesa $\{P_0, P_1, \dots, P_n\}$: P_i è in attesa di P_{i+1} , con P_n in attesa di P_0).

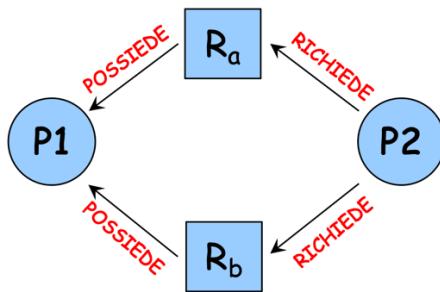
Pertanto, è possibile affermare che le prime tre sono condizioni necessarie ma non sufficienti, mentre l'ultima è condizione necessaria e sufficiente.



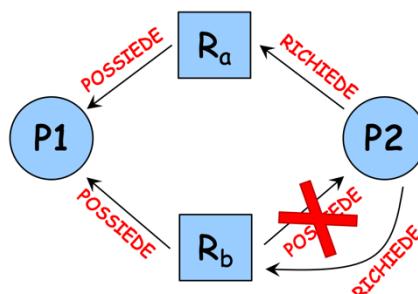
Nella **deadlock prevention**, si evita il deadlock **invalidando una delle quattro condizioni**, causando inefficienza nell'uso delle risorse per mancato loro utilizzo quando disponibili e per esecuzione rallentata dei processi. La **mutua esclusione** è imposta dalle caratteristiche della risorsa e spesso non è una condizione evitabile, sebbene rilassabile in alcuni casi di risorse condivise, ma impone costi maggiori e una minore efficienza.



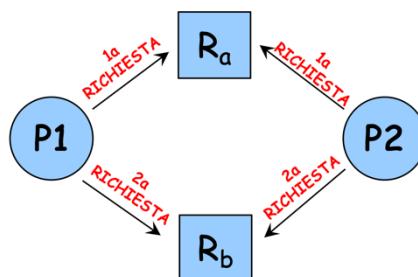
Per quanto riguarda il possesso e l'attesa, **si forza un processo a richiedere una risorsa solo quando non ne possiede altre** (ad esempio, all'avvio) ma comporta una bassa efficienza nell'uso delle risorse e il rischio di starvation.



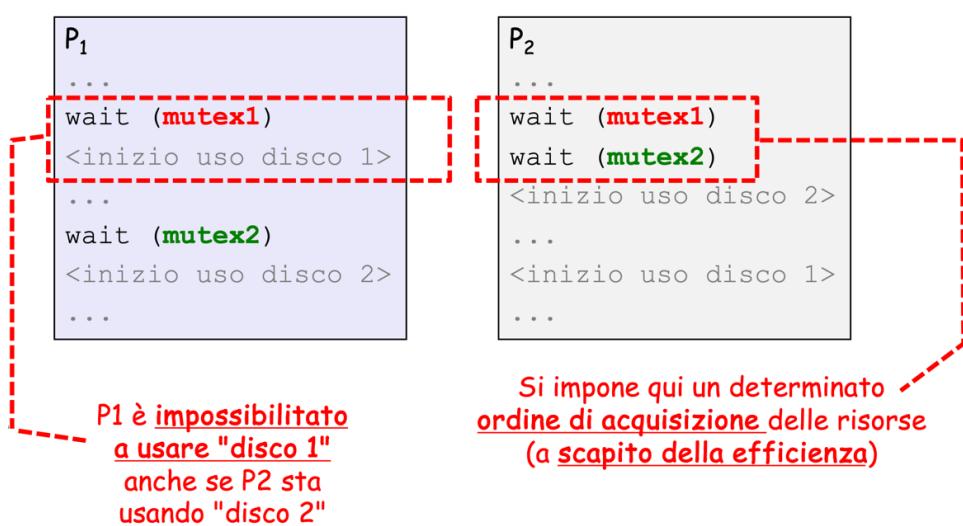
Per quanto riguarda l'impossibilità di prelazione, se un processo già possiede alcune risorse e ne richiede un'altra che non gli può essere allocata immediatamente, gli si fa rilasciare tutte le risorse possedute; allora, il processo viene eseguito nuovamente solo quando può riottenere sia le vecchie che le nuove risorse.



Infine, per quanto riguarda l'attesa circolare, si stabilisce a priori un ordinamento totale tra tutte le risorse e si impone che ciascun processo richieda le risorse seguendo l'ordine prestabilito.



Riprendendo l'esempio della copia di un file, si può imporre un ordine disco 1 – disco 2, cambiando così lo pseudocodice dei due processi:



Nella **deadlock avoidance**, il sistema decide a tempo di esecuzione se una richiesta di una risorsa può portare ad un deadlock e, pertanto, si può interpretare come una **deadlock prevention dinamica**. Questa tecnica di gestione non impone nessun vincolo a priori alle richieste, ma semplicemente rifiuta la richiesta di allocazione se lo stato attuale delle risorse è “rischioso”. Ad esempio:



Quando P_1 avrà terminato, le sue allocazioni saranno rilasciate e P_2 potrà essere ripreso.

Il **deadlock avoidance** richiede, tuttavia, un presupposto: la conoscenza anticipata di tutte le richieste che un processo può fare nell’arco della sua esecuzione. Ad esempio, il caso più semplice consiste in processi che dichiarano il numero massimo di risorse di cui possono avere bisogno (tipo, un consumo massimo di 150 kb di memoria che non sono usati necessariamente tutti subito). In quest’ottica, possono essere usati due approcci:

- All’avvio di un nuovo processo, si impedisce il suo avvio se le sue richieste potrebbero portare ad un deadlock (**Process Initiation Denial**);
- Al momento di una richiesta di allocare una risorsa, si consente l’avvio al processo ma le richieste di allocazione possono essere rifiutate se portano ad un eventuale deadlock (**Resource Allocation Denial**).

Volendo approfondire le due tecniche:

- **Process Initiation Denial**

Si consideri un **numero n di processi** e un **numero m di tipi di risorse**, con $R = \{R_1, \dots, R_m\}$ l’insieme costituito da tutti i tipi di risorse tali che R_i sia il numero di istanze della risorsa i -esima; si consideri, inoltre, $V = \{V_1, \dots, V_m\}$ l’insieme di istanze per ogni risorsa non allocate ad alcun processo (V_i è il numero di istanze della risorsa i -esima non ancora allocate), C una matrice $n \times m$ tale che C_{ij} sia la richiesta del processo P_i per la risorsa R_j (è, infatti, detta matrice delle richieste) e A una matrice $n \times m$ tale che A_{ij} sia l’allocazione corrente al processo P_i della risorsa R_j (è, infatti, detta matrice delle allocazioni).

Di necessità, C indica il numero massimo di richieste per ciascun processo (righe) di una certa risorsa (colonne) e deve essere fornita prima dell’avvio dei processi. L’algoritmo di Process Initiation Denial prevede che un processo P_{n+1} venga eseguito solo se:

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^n C_{ij}, \forall j$$

Cioè se il numero massimo di richieste di tutti i processi più quelle del nuovo possono essere soddisfatte. Ad esempio, sia considerato un sistema dotato di tre tipi di risorse:

- 100MB di memoria;
- 1 file di log;
- 1 porta seriale.

$$R = (100 \quad 1 \quad 1)$$

Con le seguenti richieste:

- P_1 : 70MB di memoria, 1 porta seriale;
- P_2 : 70MB di memoria, 1 porta seriale;
- P_3 : 50MB di memoria, 1 file di log.

$$C = \begin{pmatrix} 70 & 1 & 0 \\ 70 & 1 & 0 \\ 50 & 0 & 1 \end{pmatrix}$$

Inizialmente, con nessun processo attivo, le matrici ed i vettori individuati si configurano come segue:

$$V = (100 \quad 1 \quad 1) \wedge A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

All'avvio di P_2 , approvato perché primo processo:

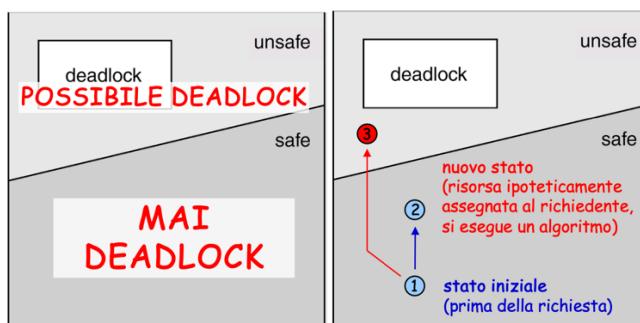
$$V = (30 \quad 0 \quad 1) \wedge A = \begin{pmatrix} 0 & 0 & 0 \\ 70 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Volendo avviare P_1 , il sistema blocca il processo per potenziale insorgenza di deadlock; infatti:

$$V = (30 \quad 0 \quad 1) \wedge A = \begin{pmatrix} 70 & 1 & 0 \\ 70 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

• Resource Allocation Denial

Chiamato anche **algoritmo del banchiere**, viene eseguito ad ogni tentativo di **allocazione** (piuttosto che all'inizializzazione del processo), il quale viene rifiutato se può condurre ad uno stato “non sicuro”. L'algoritmo in questione **fa in modo che lo stato del sistema** (ovvero, processi e risorse) **non sia mai uno stato non sicuro**; si può, così, **dividere il sistema in due porzioni**, una **safe** che non puo condurre in deadlock ed una **unsafe** soggetta, invece, ad un possibile deadlock:



Si consideri un processo che fa richiesta di assegnazione di una risorsa, in **stato 1**. Lo stato in questione **per esistere si deve già trovare nella regione sicura del sistema**; tuttavia, la richiesta

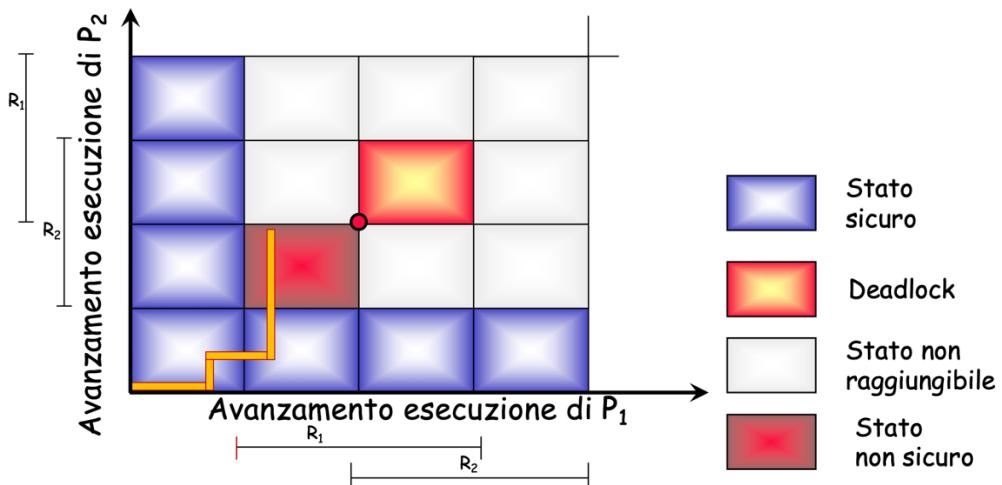
sposta il processo in questione, facendolo **rimanere nella zona sicura** (stato 2, l'algoritmo riconosce il successo dell'operazione e assegna la risorsa) o **mandandolo nella zona unsafe** (stato 3, l'algoritmo riconosce un potenziale rischio e non concede la risorsa). A questo punto è lecito chiedersi **cosa renda uno stato sicuro o meno**: il tutto dipende dalle risorse disponibili e dalle richieste di tutti i processi nel sistema.

Si considerino due processi, P_1 e P_2 , la cui sequenza di possesso e rilascio di **due risorse**, R_1 e R_2 , è la seguente:

P_1 : acquisisce $R_1 \rightarrow$ acquisisce $R_2 \rightarrow$ rilascia $R_1 \rightarrow$ rilascia R_2

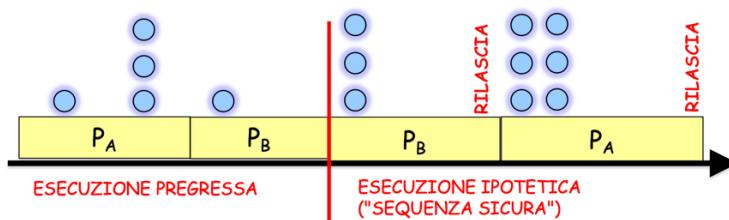
P_2 : acquisisce $R_2 \rightarrow$ acquisisce $R_1 \rightarrow$ rilascia $R_2 \rightarrow$ rilascia R_1

Ponendo su un **sistema di assi cartesiani l'avanzamento dell'esecuzione di P_1 sulle ascisse e di P_2 sulle ordinate**, si ottiene il seguente grafico:



Con questo grafico, è possibile individuare un criterio per discriminare uno stato sicuro da uno non sicuro: **il sistema è in uno stato sicuro se, partendo da tale stato, esiste una sequenza sicura di esecuzione di tutti i processi nel sistema**. Si parla di **sequenza di esecuzione ipotetica** dei processi nel sistema ed è sufficiente che tale sequenza esista.

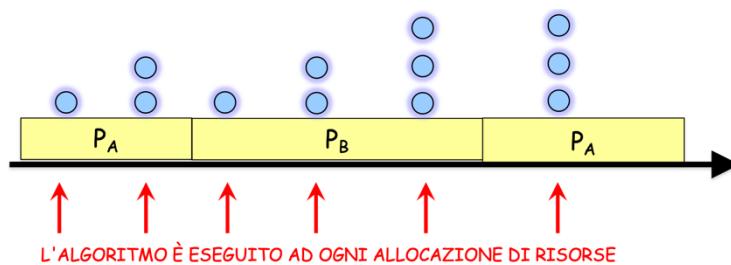
Per individuare una sequenza sicura si sceglie un processo, tra quelli del sistema, da aggiungere alla sequenza e si assegnano (ipoteticamente) tutte le risorse del suo claim, se disponibili. Per definizione, **il processo in questione ha tutte le risorse per completare la sua esecuzione**, che avviene con successo; infine, **sono rilasciate tutte le risorse del processo**. Se questa serie di operazioni è **teoricamente possibile**, si può aggiungere il processo alla sequenza e procedere per **il prossimo processo**. Ad esempio, si considerino due processi P_A e P_B , il primo dei quali ha un claim di 6 unità di risorsa ed il secondo 3. Considerando l'esecuzione pregressa in figura, si individua in $\langle P_B, P_A \rangle$ una sequenza valida per la quale lo stato è sicuro:



In conclusione, la sequenza sicura $\langle P_1, P_2, \dots, P_n \rangle$ è un ordine di esecuzione dei processi tale che:

- Sono inclusi tutti i processi attualmente attivi nel sistema;
- Ogni processo P_i esegue nella sua interezza, dopo che tutti i processi precedenti P_j (con $j < i$) abbiano a loro volta eseguito per intero e nell'ordine della sequenza (procedimento ricorsivo);
- Ogni processo P_i ottiene tutte le risorse del suo claim e le rilascia tutte al termine della propria esecuzione;
- Ogni processo P_i usa una quantità di risorse non superiore alla somma di:
 - Risorse disponibili nello stato S;
 - Risorse possedute e rilasciate dai processi precedenti P_j (con $j < i$) nella sequenza.

L'esempio precedente applica l'**algoritmo del banchiere** in un momento specifico dell'esecuzione, cioè dopo la sequenza $\langle P_A, P_B \rangle$, ma si deve considerare la necessità di **applicarlo ad ogni tentativo di allocazione**; quindi, **ogni volta che lo stato cambia, va verificata l'esistenza di una sequenza sicura**.



Intuitivamente, l'**algoritmo garantisce che esista sempre almeno una “exit strategy” che evita il deadlock**, ovvero la sequenza sicura stessa, dal momento in cui essa non è necessariamente l'ordine con cui eseguiranno i processi.

```
struct state
{
    int resource[m];
    int available[m];
    int claim[n][m];
    int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])           /* total request > claim*/
    < error >;
else if (request [*] > available [*])
    < suspend process >;
else
    /* simulate alloc */
{
    < define newstate by:
    alloc [i,*] = alloc [i,*] + request [*];
    available [*] = available [*] - request [*] >;
}
if (safe (newstate))
    < carry out allocation >;
else
{
    < restore original state >;
    < suspend process >;
}
```

(b) resource alloc algorithm

```

boolean safe (state S)
{
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible)
    {
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;>
        if (found)                                /* simulate execution of Pk */
        {
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else
            possible = false;
    }
    return (rest == null);
}

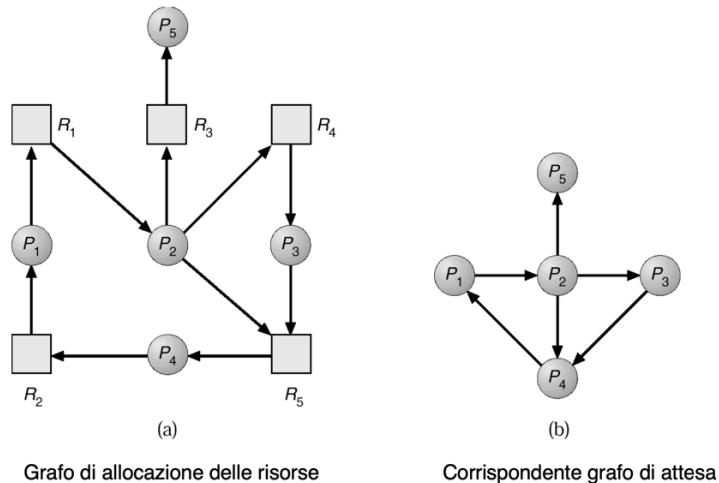
```

(c) test for safety algorithm (banker's algorithm)

La strategia del **deadlock avoidance** non è priva di problemi:

- È richiesto che sia preventivamente noto il numero massimo di risorse che verranno utilizzate;
- I processi che vengono analizzati dall'algoritmo devono essere indipendenti (non è prevista la sincronizzazione);
- Ci deve essere un numero predeterminato e costante di risorse da allocare;
- Nessun processo può terminare mentre è in possesso di una risorsa.

La **deadlock detection**, invece, non vincola le richieste alle risorse, consente il verificarsi del deadlock ma il sistema esegue un algoritmo per il rilevamento dell'attesa circolare (periodicamente o ad ogni richiesta o quando il grado di utilizzo della CPU è basso) e, in caso affermativo, applica un algoritmo di ripristino (recovery).



Una strategia di detection impiega un **grafo di attesa** i cui nodi sono processi e impone $P_i \rightarrow P_j$ se P_i è in attesa che P_j rilasci una risorsa che gli occorre; periodicamente viene richiamato un algoritmo ($O(n^2)$) che ricerca un **ciclo nel grafo** e richiede un **numero di operazioni nell'ordine di n^2** con n il numero di vertici del grafo. Per quanto riguarda le **strategie di ripristino**, si “uccidono” tutti i processi in uno stato di deadlock e si esegue un **checkpoint** di uno stato precedente al deadlock per far ripartire da quel punto i processi; queste due operazioni vengono ripetute un processo alla volta, e si prelazionano le risorse ai processi bloccati, finché non esiste più deadlock nel sistema. Il criterio di selezione dei processi da abortire o delle risorse da prelazionare prende in considerazione:

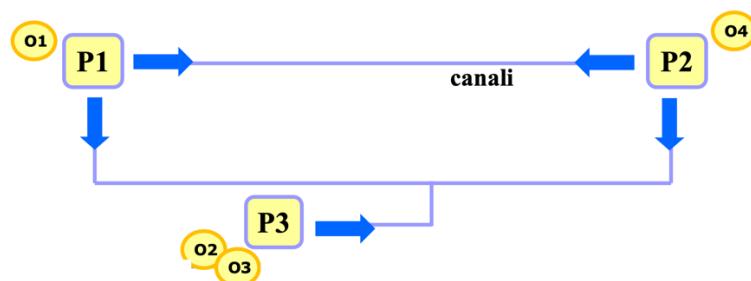
- **Minor tempo di CPU consumato** fino a quel punto;
- **Minor numero di linee di output prodotte** fino a quel punto;
- **Maggior tempo stimato per la terminazione;**
- **Minor numero di risorse allocate** fino a quel punto;
- **Minore priorità.**

In conclusione, si riassumono e si comparano le diverse strategie per la gestione del deadlock nella seguente tabella:

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	<ul style="list-style-type: none"> • Works well for processes that perform a single burst of activity • No preemption necessary 	<ul style="list-style-type: none"> • Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	<ul style="list-style-type: none"> • Convenient when applied to resources whose state can be saved and restored easily 	<ul style="list-style-type: none"> • Preempts more often than necessary
		Resource ordering	<ul style="list-style-type: none"> • Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design 	<ul style="list-style-type: none"> • Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	<ul style="list-style-type: none"> • No preemption necessary 	<ul style="list-style-type: none"> • Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	<ul style="list-style-type: none"> • Never delays process initiation • Facilitates on-line handling 	<ul style="list-style-type: none"> • Inherent preemption losses

SINCRONIZZAZIONE NEL MODELLO AD AMBIENTE LOCALE

Adottando il **modello ad ambiente locale**, va tenuto in considerazione che **non esiste memoria condivisa** e le **risorse sono tutte private**, quindi **non modificabili direttamente da altri processi**. Il **naturale supporto fisico** a questo tipo di modello sono i **sistemi con architettura distribuita**.



In ambiente locale, la **cooperazione avviene tramite scambio diretto di messaggi** (visto che non c'è uno spazio di indirizzamento comune ai processi) per mezzo di primitive fornite dal **Sistema Operativo (IPC, Inter Process Communication)**; in questo modo, la **mutua esclusione è garantita a priori**, dal momento in cui tutte le risorse sono private. Le primitive menzionate sono di due tipi: **send(destination, message)** e **receive(source, message)**; ovviamente, fra i sistemi variano in base al tipo di sincronizzazione dei processi comunicanti e all'indirizzamento, ovvero la modalità con cui si designano provenienza e destinazione. Un **messaggio**, invece, viene strutturato come segue:

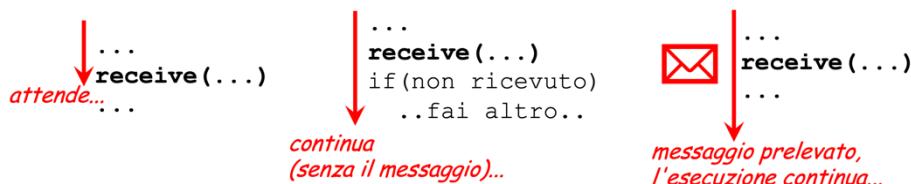


All'invio di un messaggio tramite una **send**, sono possibili due scenari:



Il **primo approccio** prende il nome di **send sincrona** e il processo che la esegue rimane in attesa finché il messaggio è stato ricevuto (richiedendo una conferma di ricezione), mentre il **secondo** prende il nome di **send asincrona** e il processo che la esegue continua la sua esecuzione senza attendere l'avvenuta consegna del messaggio.

Alla ricezione di un messaggio tramite una **receive**, sono possibili due scenari:



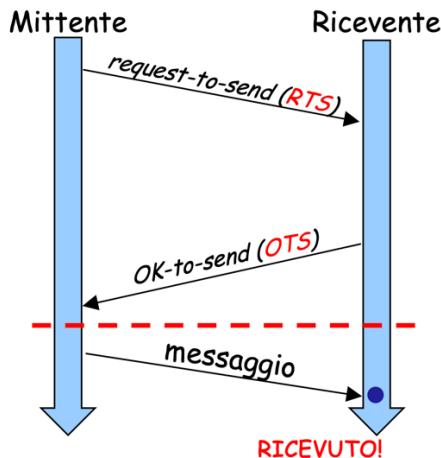
Il **primo approccio** prende il nome di **receive bloccante** e impone di rimanere in attesa fino alla ricezione del messaggio, se questo non è stato ancora inviato, mentre il **secondo** prende il nome di **receive non bloccante** e permette di continuare l'esecuzione senza dover attendere l'avvenuta consegna del messaggio; infine, se il sender ha già mandato il messaggio, il processo lo riceve e continua l'esecuzione.

Il sender e il receiver possono porre in attesa il processo a seconda del tipo di primitive utilizzate, individuando generalmente tre combinazioni:

- **Send sincrona e receive bloccante** (o rendezvous), una stretta sincronizzazione tra sender e receiver dal momento in cui entrambi rimangono in attesa della consegna del messaggio;
- **Send asincrona e receive bloccante**, il sender continua ad eseguire dopo l'invio ma il receiver attende;
- **Send asincrona e receive non bloccante**, nessuna delle due parti rimane in attesa.

La combinazione più comune nei Sistemi Operativi e nei linguaggi di programmazione è la seconda, con **send asincrona e receive bloccante**.

In realtà, sebbene possa essere considerata una primitiva a sé stante, la send sincrona può essere realizzata a partire da una send asincrona e una receive bloccante, facendo in modo che il programma determini se il messaggio è stato ricevuto e inviando eventuali messaggi aggiuntivi:



```

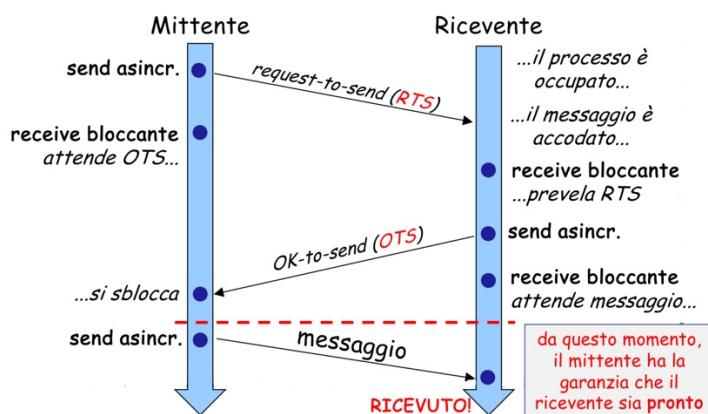
procedure sendSincrona(dest, mess)
{
    sendAsincrona (dest, messRTS)
    /* messRTS è un messaggio di "pronto
       ad inviare" (Request To Send) */
    receiveBloccante (dest, messOTS)
    /* messOTS è un messaggio di "pronto
       a ricevere" (OK To Send) */
    sendAsincrona (dest, mess)
}

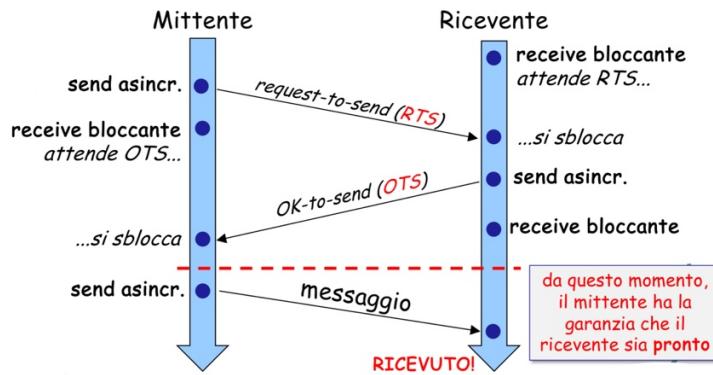
procedure receive(source, mess)
{
    receiveBloccante (source, messRTS)

    sendAsincrona (source, messOTS)

    receiveBloccante (source, mess)
}
  
```

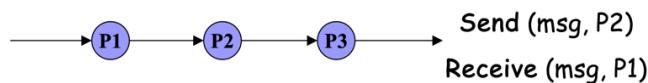
Avendo sviluppato la send sincrona, è possibile implementare anche un meccanismo di rendezvous; si considerano due casi, nel primo il ricevente non è ancora pronto a ricevere, mentre nel secondo lo è:



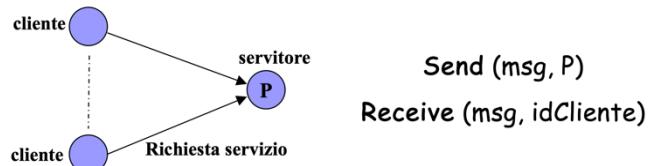


La destinazione e la provenienza possono essere gestite in più modi:

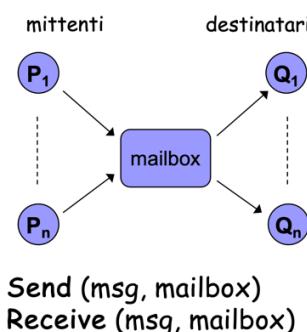
- **Comunicazione diretta simmetrica** (schema a pipeline), per la quale, il mittente nella `send()` e il destinatario nella `receive()`, esplicitano il PID dell'altro;



- **Comunicazione diretta asimmetrica** (schema client – server), per la quale il mittente esplicita il PID del destinatario nella `send()` ma il destinatario non indica alcun PID (viene a conoscenza del PID del mittente alla ricezione del messaggio tramite parametro di uscita);



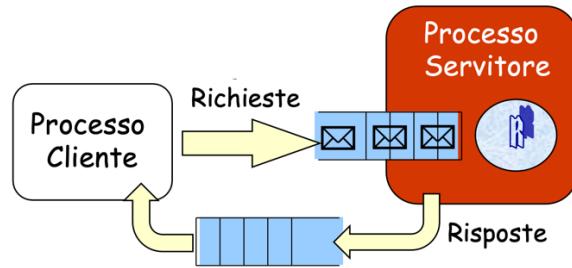
- **Comunicazione indiretta**, per la quale, il mittente nella `send()` e il destinatario nella `receive()`, fanno riferimento ad una stessa mailbox, dalla quale il destinatario preleva anche il messaggio tramite `receive()`.



Il vantaggio di una comunicazione indiretta risiede in una maggior indipendenza di sender e receiver, grazie proprio alla mailbox. I possibili schemi di comunicazione indiretta sono **one – to – one, one – to – many, many – to – one e many – to – many**.

Nel caso di **comunicazione diretta asimmetrica o indiretta many – to – one** si parla di **comunicazione client – server** e di **processo servitore**; quest'ultimo è **il processo che incapsula la risorsa comune**, offrendo a processi esterni le **funzionalità di accesso** alla risorsa. La pipeline di un processo servitore è la seguente:

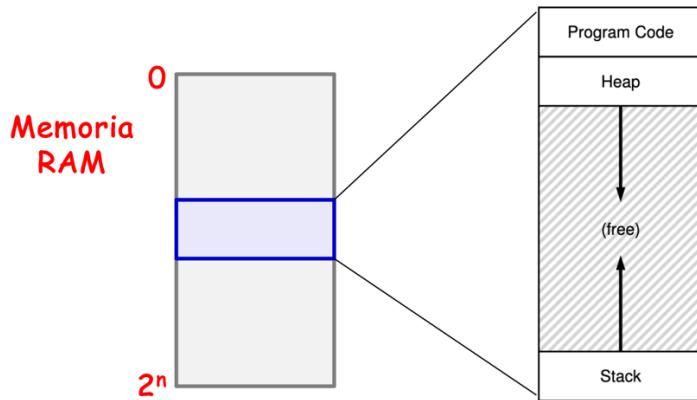
- **Riceve** messaggi di richiesta;
- **Opera** sulla risorsa;
- **Fornisce** eventuali risposte.



GESTIONE DELLA MEMORIA E DELL'I/O

INTRODUZIONE ALLA GESTIONE DELLA MEMORIA

La memoria principale costituisce, insieme alla CPU, una delle risorse principali per realizzare l'astrazione di processo; infatti, quest'ultimo dispone di un'area di memoria riservata, cioè non accessibile ad altri processi. Concettualmente, la memoria riservata di un processo può essere mappata con n bit di indirizzamento, che puntano ad un massimo di 2^n locazioni di memoria:



Ovviamente, come si può intuire, al processo non saranno assegnati tutte le 2^n locazioni di memoria, ma una porzione indirizzabile tramite n bit. Volendo scrivere un semplice programma che alloca una variabile e stampa a schermo il suo indirizzo in memoria, si ottiene una cosa di questo tipo:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int x = 3;
    printf("location of stack: %p\n", &x);

    char * p = malloc(1024);
    printf("location of heap : %p\n", p);

    printf("location of code : %p\n", main);
}
```

Il cui output è:

```
location of stack: 0x7fff691aea64
location of heap : 0x1096008c0
location of code : 0x1095afe50
```

Ci si chiede se esista davvero l'indirizzo **0x7fff691aea64** nella memoria RAM; la risposta può essere fornita andando ad utilizzare strumenti di debugging che permettono di osservare lo stato della memoria RAM durante l'esecuzione del programma, andando a rilevare come **l'indirizzo stampato non esista**.

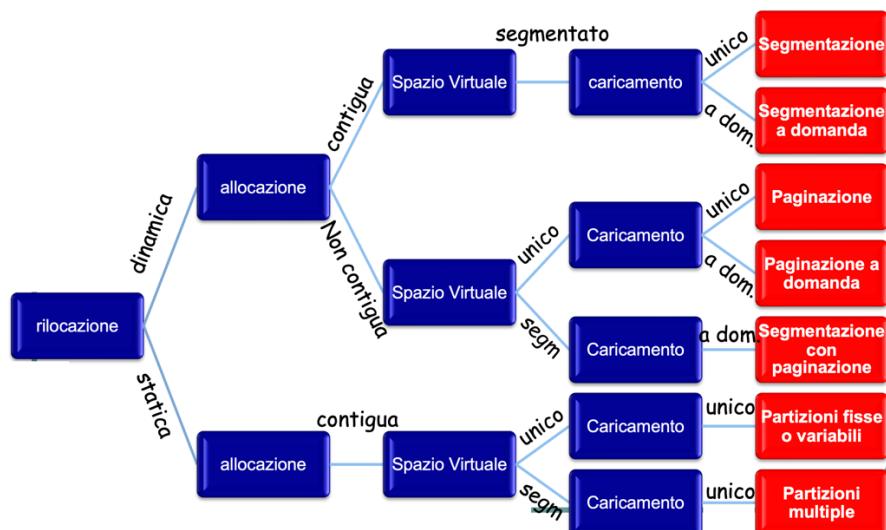
```
Unhandled exception at 0x7217C240 (mscorlib.ni.dll) in iisexpress.exe.3192.dmp:
0xC00000FD: Stack overflow (parameters: 0x00000001, 0x0A262FFC).
```

La posizione (ovvero gli indirizzi) di codice e dati nella memoria del processo è un'astrazione, ovvero indica indirizzi virtuali, perché la posizione effettiva in memoria fisica è gestita dal Sistema Operativo.

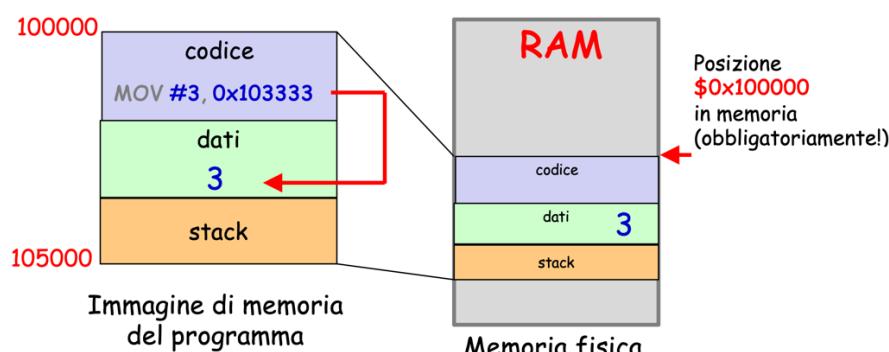
Per **gestione della memoria** si intende l'implementazione e l'osservazione di una serie di tecniche e di parametri come:

- **Supporto Hardware** per la gestione della memoria (MMU);
- **Organizzazione logica** della memoria virtuale;
- **Organizzazione fisica** (allocazione del processo);
- **Dimensione della memoria virtuale** (dimensione dell'immagine di un processo maggiore della dimensione fisica della memoria);
- **Rilocazione**, statica o dinamica;
- **Organizzazione** dello spazio virtuale, unico o segmentato;
- **Allocazione**, contigua o non;
- **Caricamento**, tutto insieme o a domanda.

Volendo entrare nel dettaglio delle tecniche di gestione della memoria, si può individuare il seguente schema per differenziare i vari tipi di implementazione:



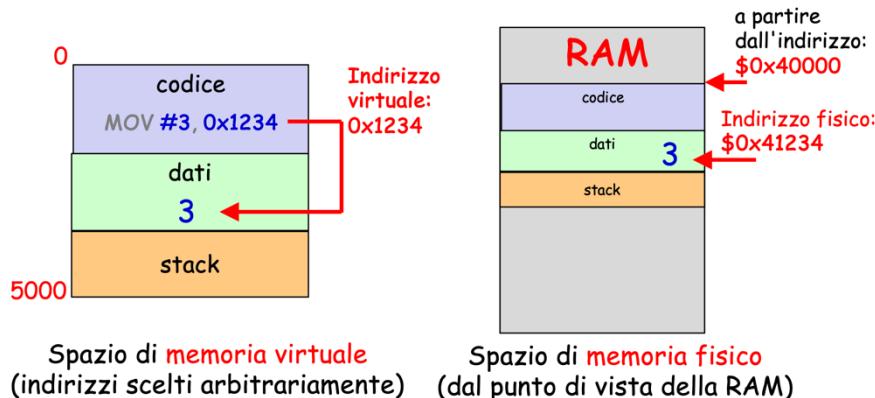
La rilocazione statica stabilisce gli indirizzi di codice e dati al momento della compilazione o caricamento e non permette la modifica della loro posizione una volta assegnata.



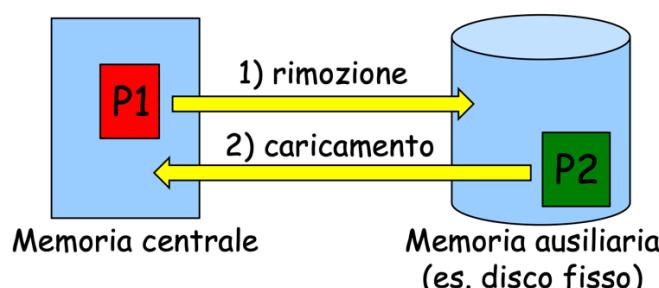
Con la **rilocazione dinamica**, invece, è necessario fare la seguente distinzione:

- **Indirizzo virtuale** (virtual o logical address), individuato come l'indirizzo acceduto dal programma (accessi a variabili, salti o chiamate di funzioni, ecc...) durante la sua esecuzione;
- **Indirizzo fisico** (physical address), individuato come l'indirizzo visto dall'unità di memoria e posizione effettiva del dato/istruzione.

Si noti, che **un processo allocato in un Sistema Operativo che gestisce dinamicamente la rilocazione non ha modo di conoscere gli indirizzi fisici dei propri dati/codici.**



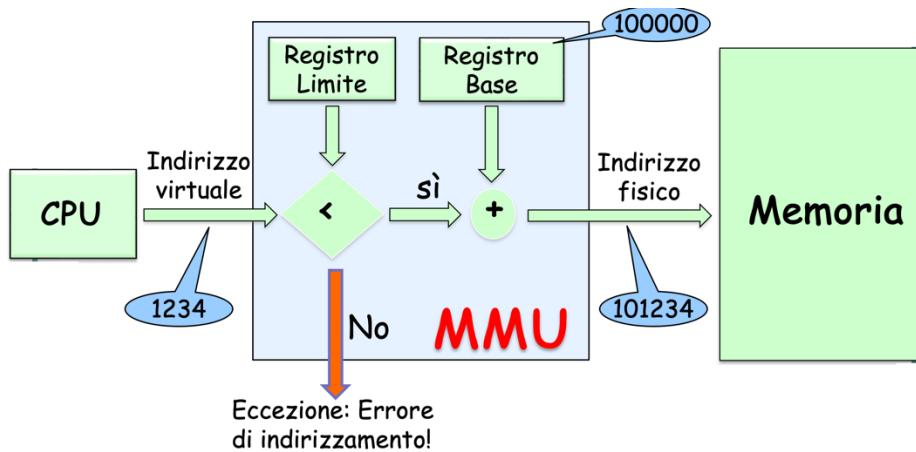
La **rilocazione dinamica**, come è facile intuire, **consente la pratica dello swapping**, ovvero la possibilità di sospendere temporaneamente un processo e trasferirlo in una memoria secondaria (come i dischi) per poi poter essere ricaricato in un'area di memoria differente.



L'**unità di gestione della memoria** (Memory Management Unit) è una **componente hardware** della CPU che **traduce gli indirizzi virtuali in indirizzi fisici** durante l'esecuzione in modo che, dopo la traduzione, la CPU sia in grado di accedere agli indirizzi fisici. Un **approccio basilare** (e obsoleto) di rilocazione dinamica e di uso della MMU prevede che **lo spazio virtuale sia unico e l'allocazione contigua**; la **MMU utilizza due registri speciali**, il **registro base** e il **registro limite**, in modo che:

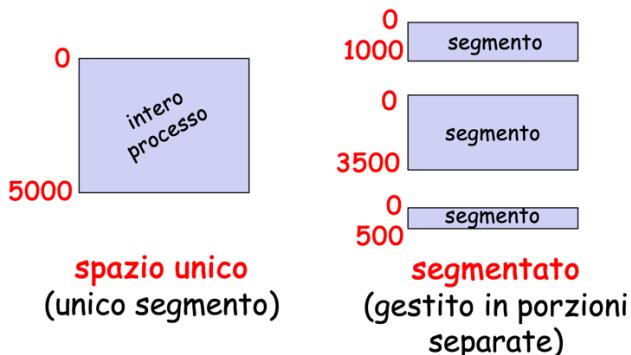
$$\text{indirizzo fisico} = \text{indirizzo virtuale} + \text{registro base}$$

In particolare, **prima di effettuare questa operazione, c'è un comparatore che verifica se l'indirizzo virtuale è inferiore al registro limite**, in modo da sollevare un'eccezione se l'indirizzo virtuale non è correttamente mappato.

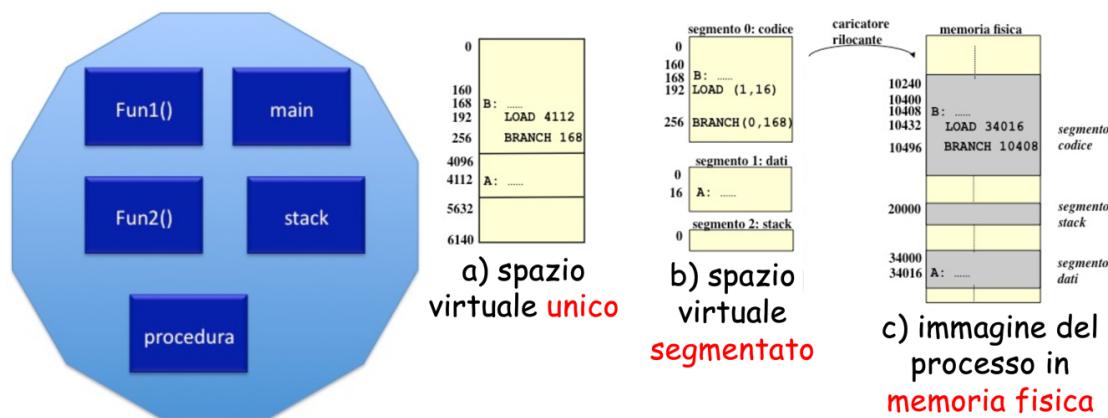


Per gestire lo spazio virtuale degli indirizzi sono possibili due approcci:

- **Spazio unico**, corrispondente all'intero processo;
- **Segmentazione**, la memoria del processo è gestita in porzioni separate.



Utilizzando uno spazio virtuale segmentato, a tempo di configurazione viene creato un diverso segmento per ciascun modulo del programma:



I vantaggi principali di questo approccio sono i seguenti:

- **Protezione dei segmenti** (ad esempio, durante le operazioni di scrittura e lettura o durante l'esecuzione);
- **Condivisione dei segmenti** (ad esempio, una libreria di codice utilizzata da più processi);
- **Allocazione indipendente** dei segmenti in memoria fisica (riducendo, pur non eliminando, il problema della frammentazione esterna);

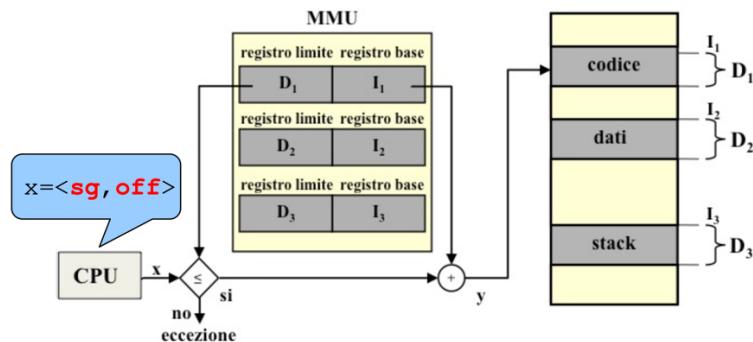
- **Separazione logica** delle differenti parti di un programma;
- **Semplificazione della gestione** di strutture dati complesse.

Praticamente, la **segmentazione viene implementata associando ad un indirizzo virtuale una coppia fatta di un identificativo del segmento** (che può essere il suo entry point) ed **uno scostamento** (o offset) che indica quanto in profondità il dato/codice è in quel segmento. Ad esempio, in architettura x86:

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

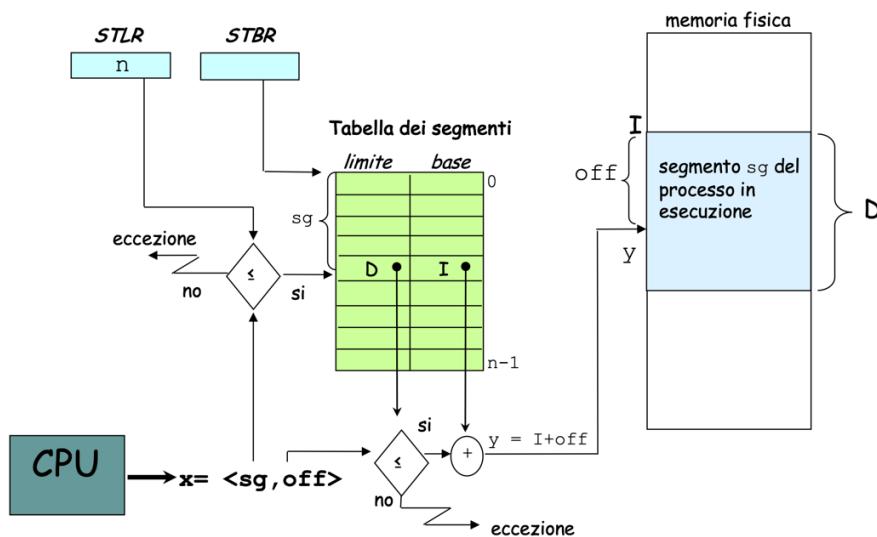
L'indirizzo virtuale di questa locazione è la coppia 0x1234 (identificativo) – 0x5678 (offset).

Anche la **segmentazione si basa su MMU**, dal momento in cui è **più che mai necessario tradurre indirizzi virtuali in indirizzi fisici**; nel caso di **pochi segmenti** (tipo 3) è sufficiente avere **nella MMU più coppie di registri base – limite**:



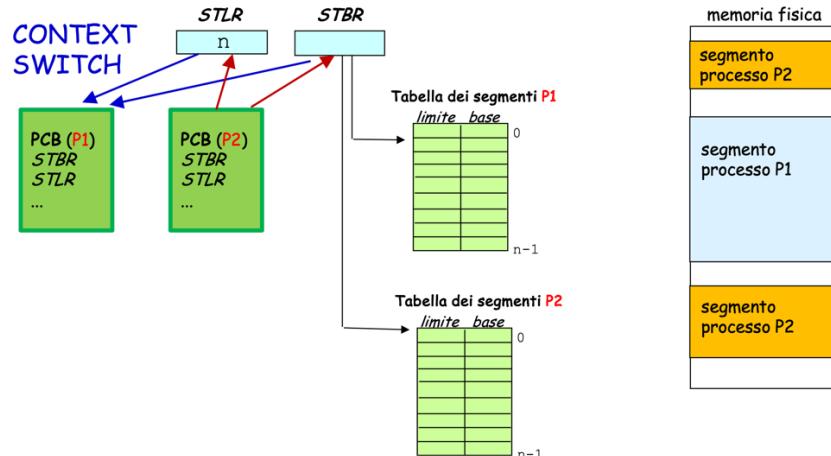
Per ottenere un **numero arbitrario di segmenti**, invece, è necessario **conservare tutte le coppie di registri base – limite per ogni segmento in memoria RAM** (dal momento in cui non è né possibile né pratico conservarle in MMU), in un'area apposita detta **tabella dei segmenti** (segment table). La **MMU, poi, gestisce la segment table con due appositi registri**:

- **STBR (Segment Table Base Register)**, indirizzo in memoria in cui si trova la segment table;
- **STLR (Segment Table Limit Register)**, dimensione della tabella dei segmenti, ovvero il numero di segmenti del processo.

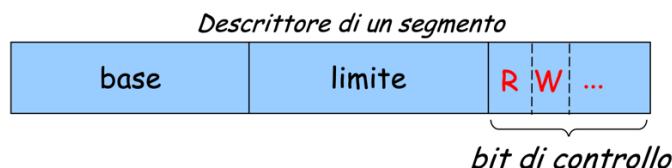


Si noti che la riga del segmento **sg** non contiene il valore di **sg** stesso; la MMU calcola la somma STBR+sg per trovare la posizione della riga (algoritmo di selezione lineare).

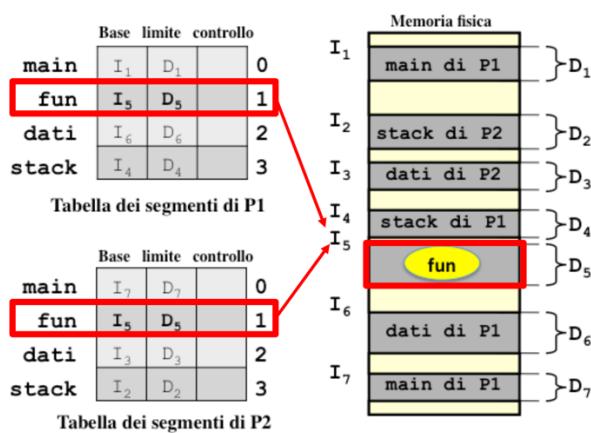
Ogni processo ha una segment table differente e, pertanto, i registri STBR – STLR sono configurati ad ogni context switch dei processi: il Sistema Operativo carica i valori di STBR – STLR dal PCB (Process Control Block).



La segmentazione permette anche di avere segmenti con differenti permessi di accesso, tramite dei **bit di controllo** in ogni riga della tabella dei segmenti e facendo in modo che la MMU produca un'eccezione se il programma non rispetta tali permessi.



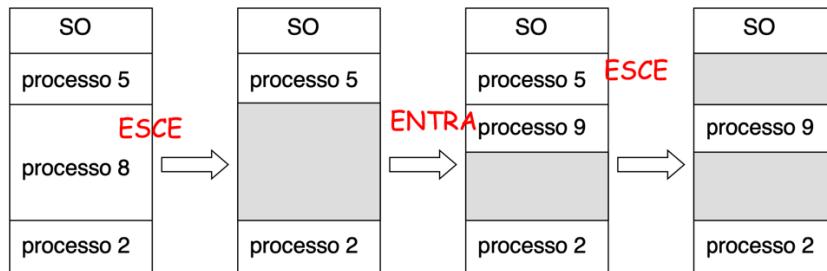
Infine, come precedentemente annunciato, è possibile condividere dei segmenti tra più processi, allocando in memoria fisica una sola copia del segmento:



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

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

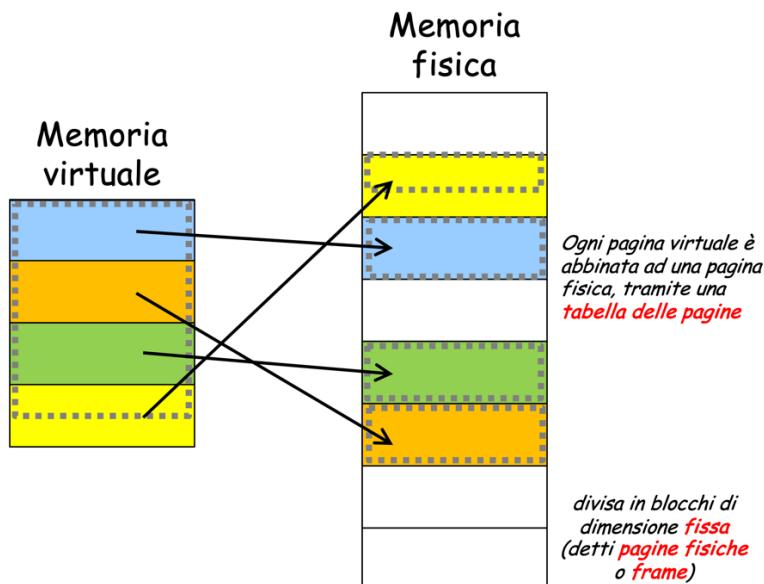
Per quanto riguarda l'allocazione contigua, il Sistema Operativo si occupa di collocare il proprio blocco di memoria virtuale, e quelli dei processi, in intervalli non sovrapposti della memoria fisica; quando un processo termina, la memoria fisica occupata si libera, creando un buco (o hole), mentre al caricamento di un processo occorre cercare un buco sufficientemente grande da contenerlo per intero:



Se ci sono più buchi liberi in cui un segmento può essere collocato, ci sono diversi criteri per scegliere la collocazione migliore:

- **First – fit**, per il quale si assegna il primo buco sufficientemente grande;
- **Best – fit**, per il quale si assegna il buco più piccolo tra quelli sufficientemente grandi;
- **Worst – fit**, per il quale si assegna il buco più grande tra quelli sufficientemente grandi.

In generale, gli schemi a partizioni di dimensione variabile soffrono del problema della frammentazione esterna, quel fenomeno per cui la quantità di memoria dello spazio è ridotta a causa degli spezzoni che si vanno a creare tra i diversi segmenti collocati, eventualmente troppo piccoli per collocarne altri (lo spazio di memoria totale sarebbe sufficiente a soddisfare una richiesta ma non è contiguo). Il problema della frammentazione viene risolto agendo sulla contiguità dell'allocazione; la paginazione è una tecnica di allocazione non contigua per la quale gli spazi/segmenti sono divisi in blocchi di dimensione fissa. Sebbene la paginazione eviti la frammentazione esterna, introduce il problema della frammentazione interna, per il quale un processo non utilizza appieno le pagine assegnate:



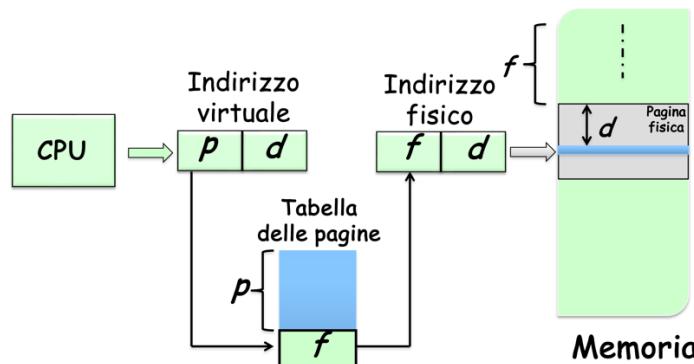
La frammentazione interna si manifesta come perdita di spazio di memoria a causa di un blocco assegnato ma non utilizzato per intero e occorre quando la dimensione del processo non è un multiplo intero della dimensione dei blocchi. Nonostante sia un problema non di poco conto, se la

dimensione dei blocchi (o pagine) è ridotta, lo si può trascurare, dal momento in cui **lo spazio perso è al più leggermente inferiore della dimensione della pagina**; tipicamente, la dimensione di una pagina è una potenza di 2, compresa tra i 512 byte e i 16MB.

Adottando la tecnica della paginazione, **un indirizzo contiene la coppia $p - d$** , con **p numero della pagina** (ovvero l'indice della pagina nella memoria fisica) e con **d scostamento di pagina** (ovvero la posizione dell'indirizzo all'interno della pagina). A differenza della segmentazione, **non sono due valori separati ma contenuti entrambi in un unico valore**; ad esempio, con un indirizzo virtuale a 16 bit:

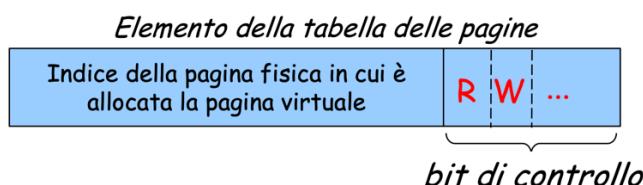
$$0x0803 \rightarrow 0000100000000011 = 000010 + 000000011 \rightarrow p = 2 \wedge d = 3$$

Con una dimensione delle pagine pari a 2^{10} bytes (1kB) e un numero massimo di pagine pari a $2^6 = 64$ pagine. L'architettura che permette la paginazione è schematizzata come segue:



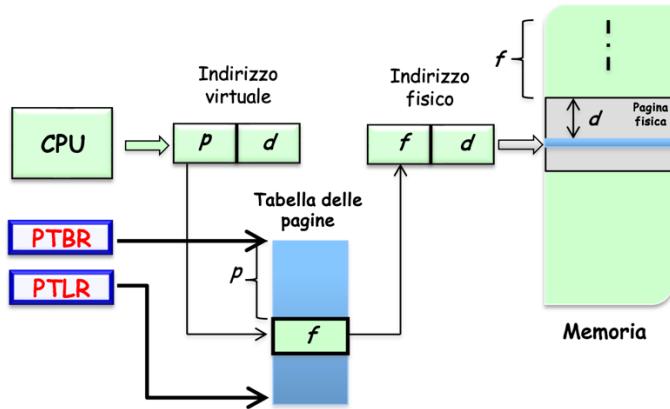
Oltre all'eliminazione della frammentazione esterna, **la paginazione semplifica l'allocazione in memoria e garantisce il supporto allo swapping e alla virtualizzazione della memoria**.

La tabella delle pagine ha una riga per ogni pagina virtuale del processo e contiene l'indice della pagina fisica e i bit di gestione (ad esempio, i permessi di accesso, ecc...):

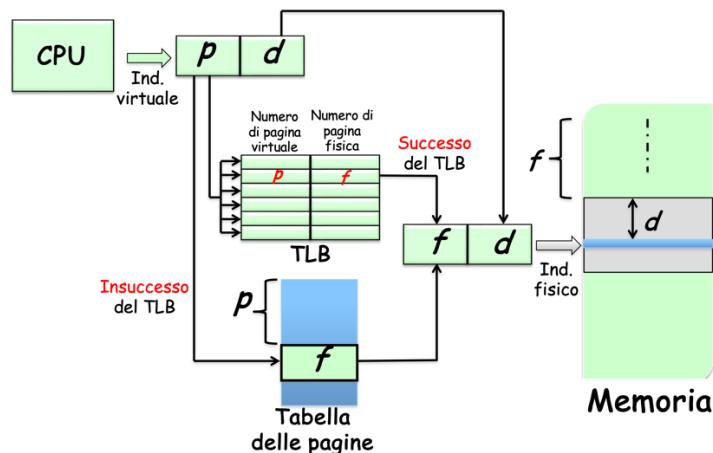


Facendo riferimento allo schema architettonico precedente, **la riga contiene l'indice della pagina fisica f , non quello della pagina virtuale p** . Sempre parlando di architettura, **la tabella delle pagine è in memoria principale e la MMU utilizza due registri per la sua gestione**:

- **PTBR** (Page Table Base Register), indirizzo base della tabella delle pagine;
- **PTLR** (Page Table Length Register), dimensione della tabella delle pagine.



Quindi, per accedere alla memoria occorrono due accessi: uno per leggere la tabella delle pagine e uno per accedere al vero e proprio dato/istruzione. Questa pipeline rallenta notevolmente gli accessi in memoria ma, per una maggiore efficienza, può essere usata una cache associativa detta TLB (Translation Look – aside Buffer):



Si noti che la selezione di una riga del TLB è associativa, cioè basata sul contenuto, mentre quella di una riga della tabella delle pagine in memoria è lineare, cioè basata sulla posizione.

Viene definito **tasso di successo**, o **hit ratio α** , la percentuale di volte che un numero di pagina virtuale si trova nel TLB; supponendo il lookup associativo pari a ε unità di tempo e l'accesso in memoria k unità di tempo, il **tempo effettivo di accesso** è pari a:

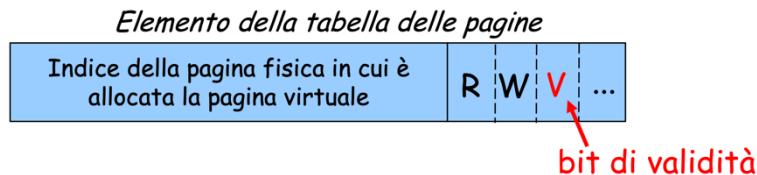
$$EAT = \underbrace{(k + \varepsilon)\alpha}_{\text{Caso di successo TLB}} + \underbrace{(2k + \varepsilon)(1 - \alpha)}_{\text{Caso di insuccesso TLB}} = (2 - \alpha)k + \varepsilon$$

Supponendo indirizzi a 32bit, con uno spazio di indirizzamento di 4GB, si consideri:

Dimensione delle pagine	Dimensione della tabella	Frammentazione interna media
1kB	4MB (non ottimale)	0.5kB (ottimale)
64kB	64kB (ottimale)	32kB (non ottimale)

Quindi, bisogna scegliere una dimensione di pagina che abbia un buon compromesso tra dimensione della tabella e frammentazione interna media.

Raramente un processo usa tutto il suo spazio di indirizzamento virtuale: generalmente per 4GB di spazio virtuale un'applicazione desktop ne usa circa 100MB. Il Sistema Operativo può marcare le pagine virtuali in uso usando un bit di validità nella page table, il quale viene attivato nel momento in cui la pagina è allocata dal processo (ad esempio, con `malloc()`):



Quando una pagina è marcata come non in uso ma il programma tenta di accedervi, la MMU solleva un'eccezione e il Sistema Operativo uccide il processo per **segmentation fault** (o segfault).

In caso di **context switch** tra due processi, il registro base verso la tabella delle pagine va modificato, visto che ogni processo ha una tabella differente; quando si cambia tabella, le righe nella cache del TLB vanno cancellate (flush) perché fanno riferimento ad una tabella non più attuale.

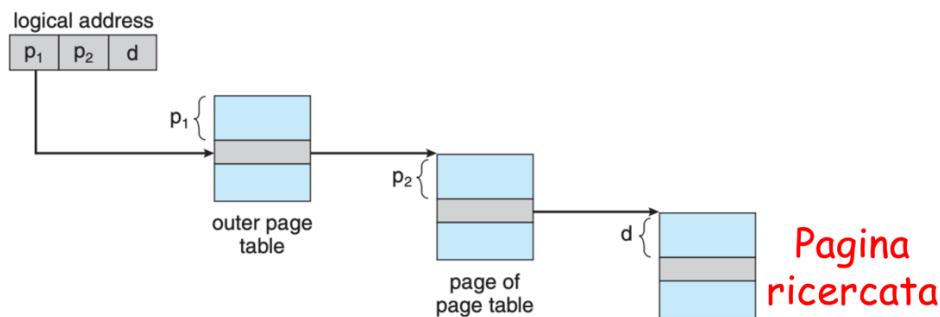
L'approccio appena illustrato ha **diversi problemi**, le tabelle delle pagine:

- Hanno grosse dimensioni;
- Sono numerose (una per processo);
- Sono sparse (poche pagine valide).

Le soluzioni possibili a questi problemi sono:

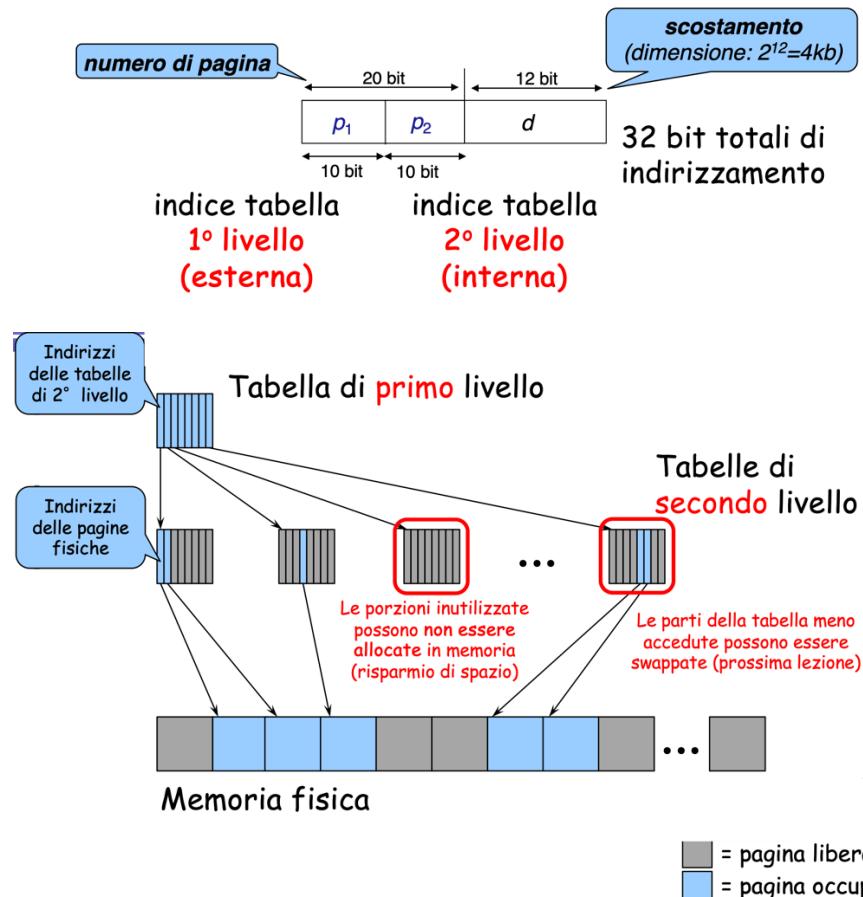
- **Paginazione gerarchica;**
- **Tabella delle pagine basata su hash;**
- **Tabella delle pagine invertita.**

Per **paginazione gerarchica** si intende la pratica per la quale la tabella delle pagine è suddivisa in parti più piccole, organizzate secondo una struttura gerarchica. La MMU divide l'indirizzo di pagina in più parti (p_1 e p_2) e, nella tabella di primo livello, trova l'indirizzo della tabella di secondo livello:

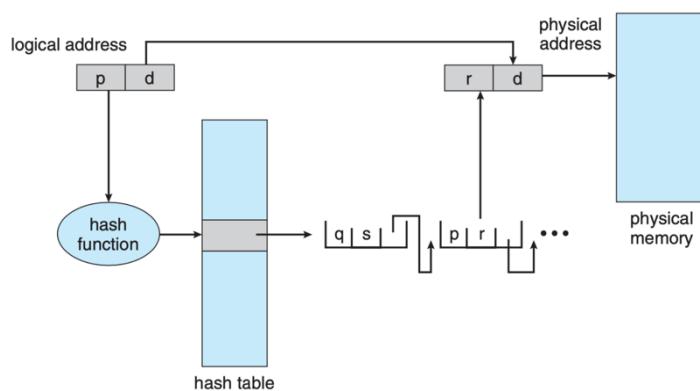


Si noti che la MMU impiega più tempo per attraversare la tabella gerarchica, aumentando i tempi di accesso alla memoria. Su due livelli, si trova una tabella di primo livello con N righe,

ognuna contenente gli indirizzi delle tabelle di secondo livello, e N tabelle di secondo livello con N righe; le singole tabelle sono più piccole (N righe) rispetto alla tabella non gerarchica ($N \cdot N$ righe):

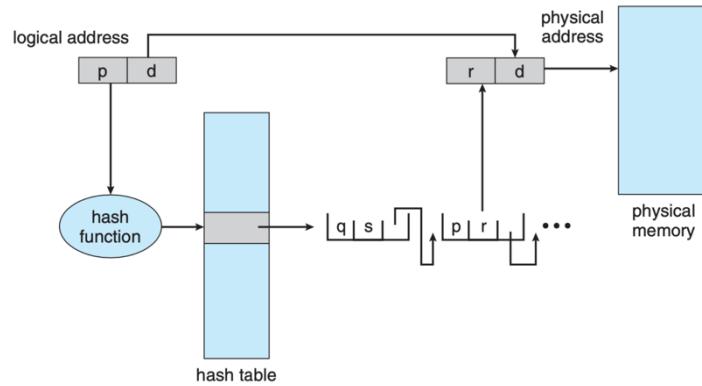


Per **tabella delle pagine basata su hash** si intende una strutturazione della tabella delle pagine tale che le righe sono organizzate usando una **linked list**: si memorizzano esclusivamente le righe per le pagine valide, con un ulteriore **risparmio di memoria** al costo di un **rallentamento della ricerca** (occorre scandire tutta la linked list, con una ricerca basata sul contenuto). Per **ottimizzare i tempi di ricerca**, si dividono le righe su tante liste concatenate di piccole dimensioni, usando una funzione di hash per il numero della pagina virtuale, così che elementi con lo stesso valore di hash sono collocati nella stessa lista concatenata.

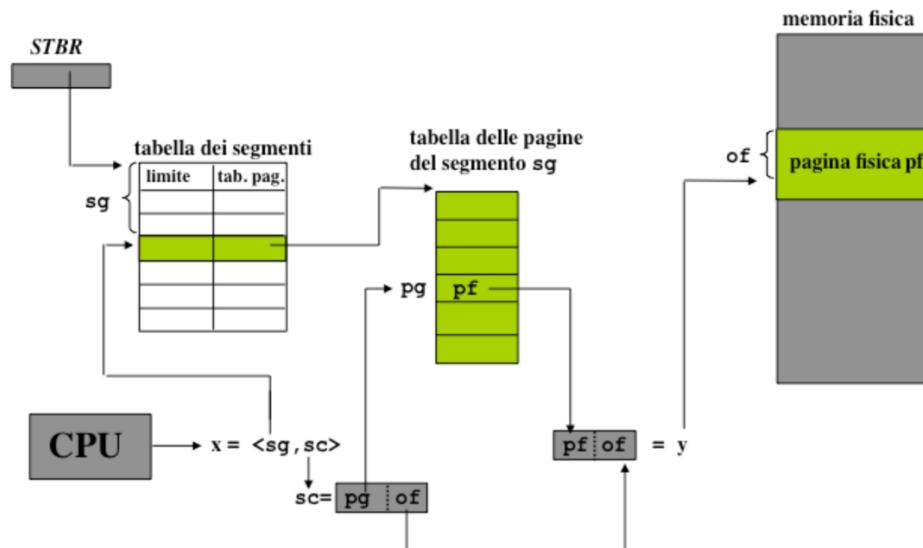


Partendo da un **indirizzo virtuale** da tradurre, si applica su di esso la **funzione di hash** (la stessa usata per dividere le liste) e viene scandita la **corrispondente lista concatenata** per trovare il **numero del blocco di memoria**.

Per **tabella delle pagine invertita** si intende una **strutturazione tale che una sola tabella delle pagine è comune a tutti i processi**, con un elemento per ogni pagina fisica contenente l'indirizzo virtuale della pagina memorizzata in quella locazione fisica, con informazioni sul processo che possiede tale pagina. Si noti che il **numero di righe è pari al numero di pagine fisiche invece che virtuali**.



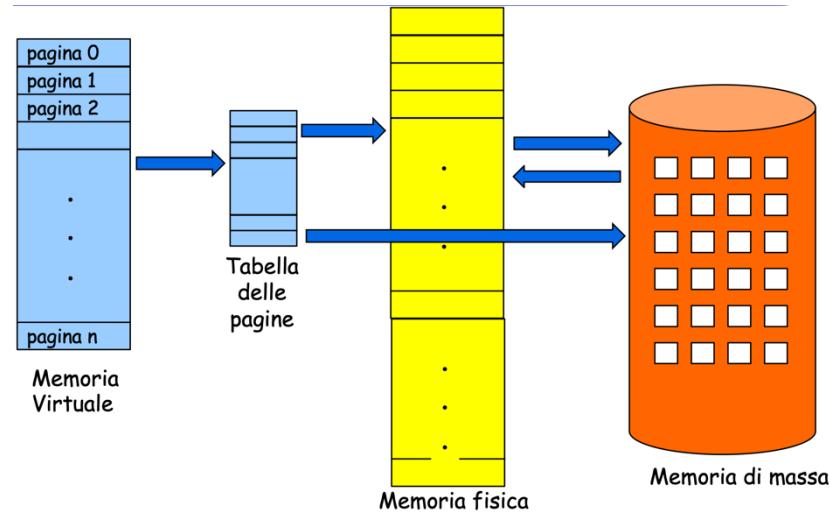
Volendo unire la segmentazione e la paginazione, la struttura architetturale assume la seguente forma:



LA MEMORIA VIRTUALE

La memoria virtuale separa la memoria fisica da quella vista da un processo, può anche essere più grande della sua controparte fisica e la sua realizzazione avviene tramite **paginazione o segmentazione su richiesta**.

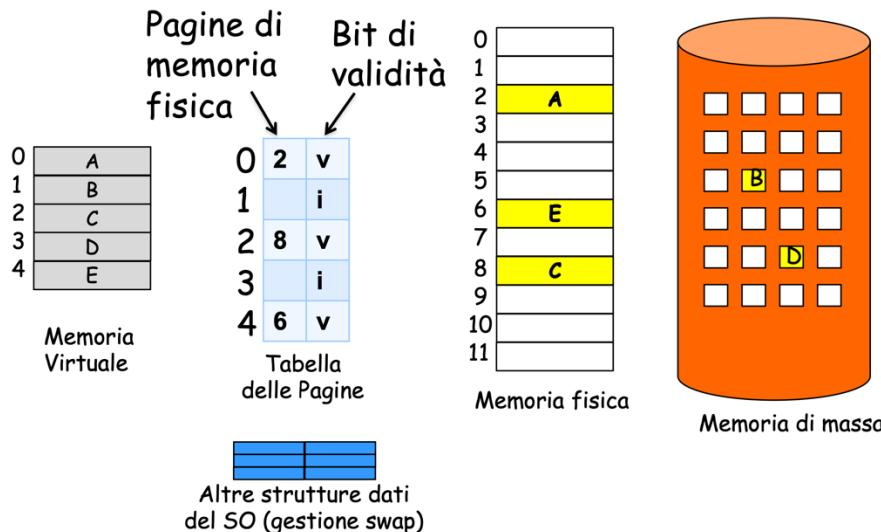
Uno **schema di memoria virtuale** si configura come segue:



La paginazione su richiesta (o demanding paging) è quella tecnica con la quale **solo le pagine effettivamente utilizzate**, dette **pagine residenti**, sono caricate in memoria principale, garantendo un **minore consumo di memoria fisica**, una **maggior quantità di utenti e processi** che possono eseguire sul sistema e un **caricamento più rapido dei processi**. Il tutto si riduce alla **valutazione del bit di validità V** nella tabella delle pagine precedentemente introdotta:

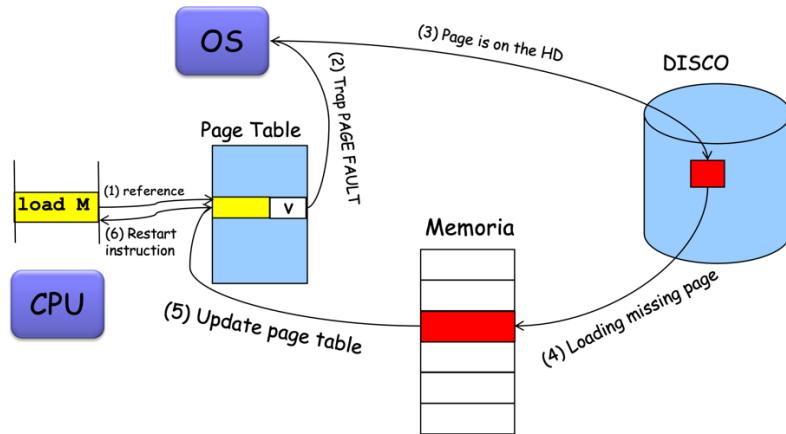
- $V=1$ se le seguenti condizioni sono entrambe vere:
 - La pagina virtuale è stata allocata al processo (ad esempio, con `malloc()`);
 - La pagina fisica corrispondente è residente in memoria;
- $V=0$ se anche solo una delle due condizioni appena enunciate è falsa.

Ad esempio:



Se un indirizzo virtuale fa riferimento ad una pagina non ancora caricata in memoria, la MMU nota che il **bit di validità non è attivo** e genera **un'eccezione di pagina mancante** (o page fault), una **ISR del Sistema Operativo** che gestisce l'eccezione nei seguenti step:

1. Individua una pagina fisica di memoria libera;
2. Trasferisce la pagina desiderata nella memoria libera;
3. Aggiorna le tabelle, ponendo il bit di validità a 1;
4. All'uscita, la CPU riavvia l'istruzione che ha causato l'eccezione.



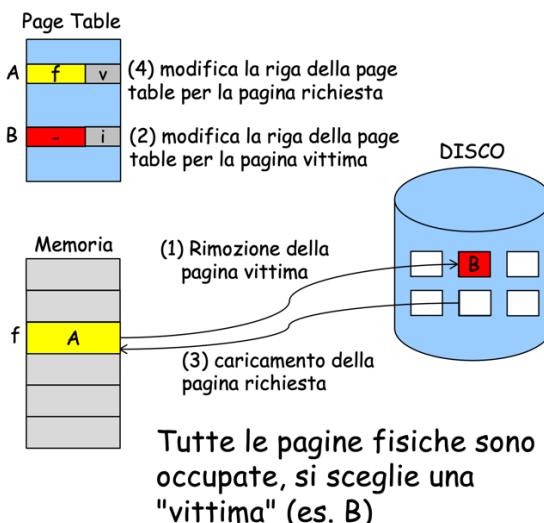
La ISR verifica anche che l'indirizzo virtuale sia stato allocato dal processo, usando ulteriori strutture dati del Sistema Operativo. In caso negativo, il processo viene terminato con **segmentation fault**. Detta $p \in [0,1]$ la probabilità di assenza di pagina, il tempo d'accesso effettivo è:

$$EAT = (1 - p) \cdot \text{accesso alla memoria}$$

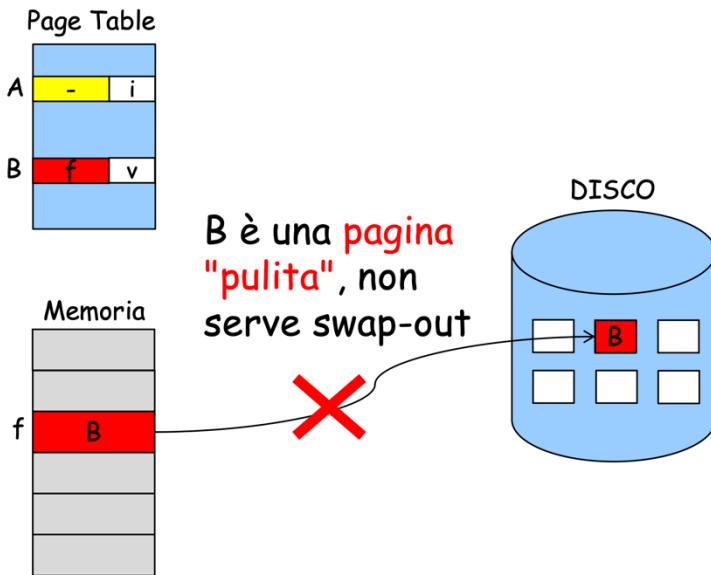
$$+ p(\text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead})$$

Da quanto si è precedentemente detto, è chiaro che serve avere una pagina fisica libera in cui caricare la pagina dal disco; per liberare spazio, si usa un algoritmo e un meccanismo di sostituzione per i quali una pagina "vittima" già in memoria viene spostata sul disco. Questo meccanismo di sostituzione viene così implementato:

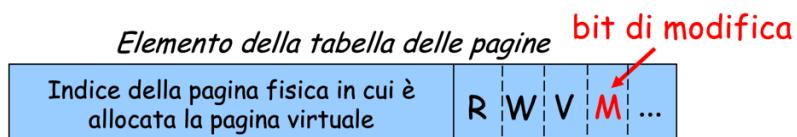
- Il Sistema Operativo individua la posizione sul disco della pagina richiesta;
- Il Sistema Operativo cerca una pagina fisica di memoria libera:
 - Se esiste, la usa;
 - Se non esiste, rimuove dalla memoria principale un'altra pagina, detta vittima, scelta con un algoritmo di sostituzione;
 - La pagina vittima viene scritta sul disco (swap out);
- Il Sistema Operativo scrive la pagina richiesta nella pagina di memoria appena liberata;
- Il Sistema Operativo modifica le tabelle delle pagine (sia del richiedente sia del processo vittima);
- Si riprende l'esecuzione del processo richiedente.



Una pagina è detta pulita, e non induce allo swap out, se è stata precedentemente copiata dal disco o se la copia in memoria non ha avuto modifiche:



Le pagine pulite sono identificate da un bit di modifica **M**, detto **dirty bit**, posto a 0 dalla MMU al caricamento in RAM (la pagina non è stata ancora modificata ma solo letta) e ad 1 alla prima modifica.

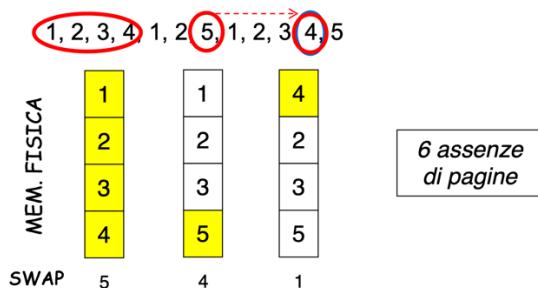


Per quanto riguarda gli **algoritmi di sostituzione** delle pagine, si tenga in considerazione il desiderio di **minimizzare la frequenza dei page fault**. Si confrontino i page fault ottenuti dai seguenti algoritmi simulando un processo con 5 pagine di memoria virtuale che vi accede in questa successione:

$1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$ (in ordine temporale, da sinistra a destra)

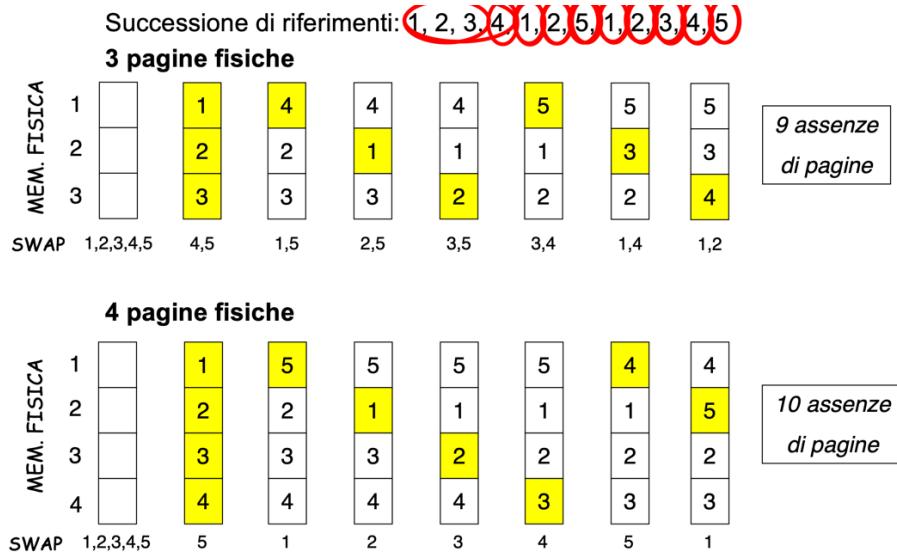
- Algoritmo “ottimo”

Si sostituisce la pagina che non si userà per il periodo di tempo più lungo. Con 4 pagine fisiche:

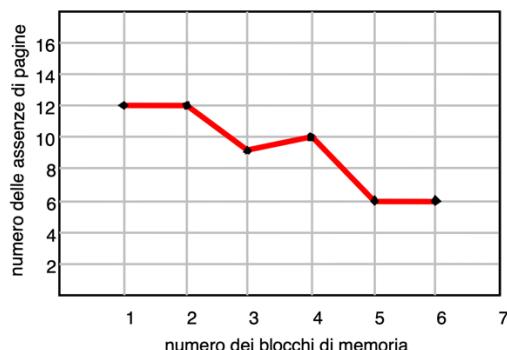


Purtroppo, **non è possibile conoscere in anticipo quali saranno i futuri riferimenti**; pertanto, questo è un algoritmo usato prevalentemente per **studi comparativi**, cioè per valutare le prestazioni di altri algoritmi.

- **FIFO**

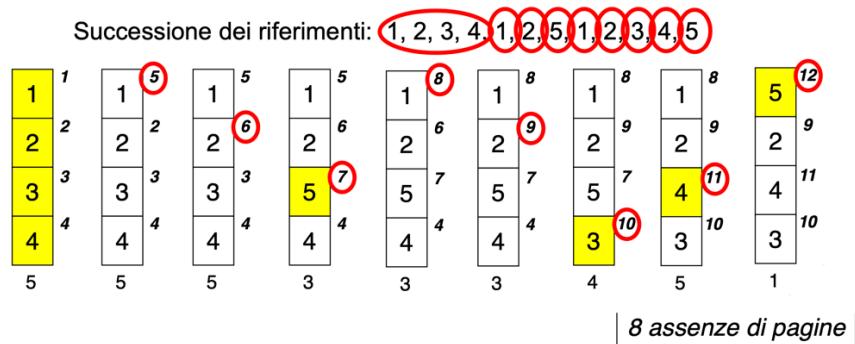


L'anomalia per la quale nell'algoritmo di sostituzione FIFO il **numero di pagine fisiche di memoria è proporzionale al numero di assenze di pagina** è detta **anomalia (o paradosso) di Belady**:



- **LRU (Last Recently Used)**

Ad ogni pagina si abbina un campo numerico, detto **“momento d’uso”**, che viene **aggiornato dalla CPU ad ogni accesso alla pagina** usando il valore di un contatore crescente; l'algoritmo sostituisce la pagina acceduta meno di recente (ovvero con il momento d'uso più basso), complicando la ricerca e la struttura della tabella delle pagine e della MMU.



La maggior parte dei sistemi utilizza un'approssimazione dello LRU in cui, al posto del contatore, si usa un singolo bit di riferimento, inizializzato dal Sistema Operativo a 0 e posto ad 1 ad ogni accesso dalla MMU.

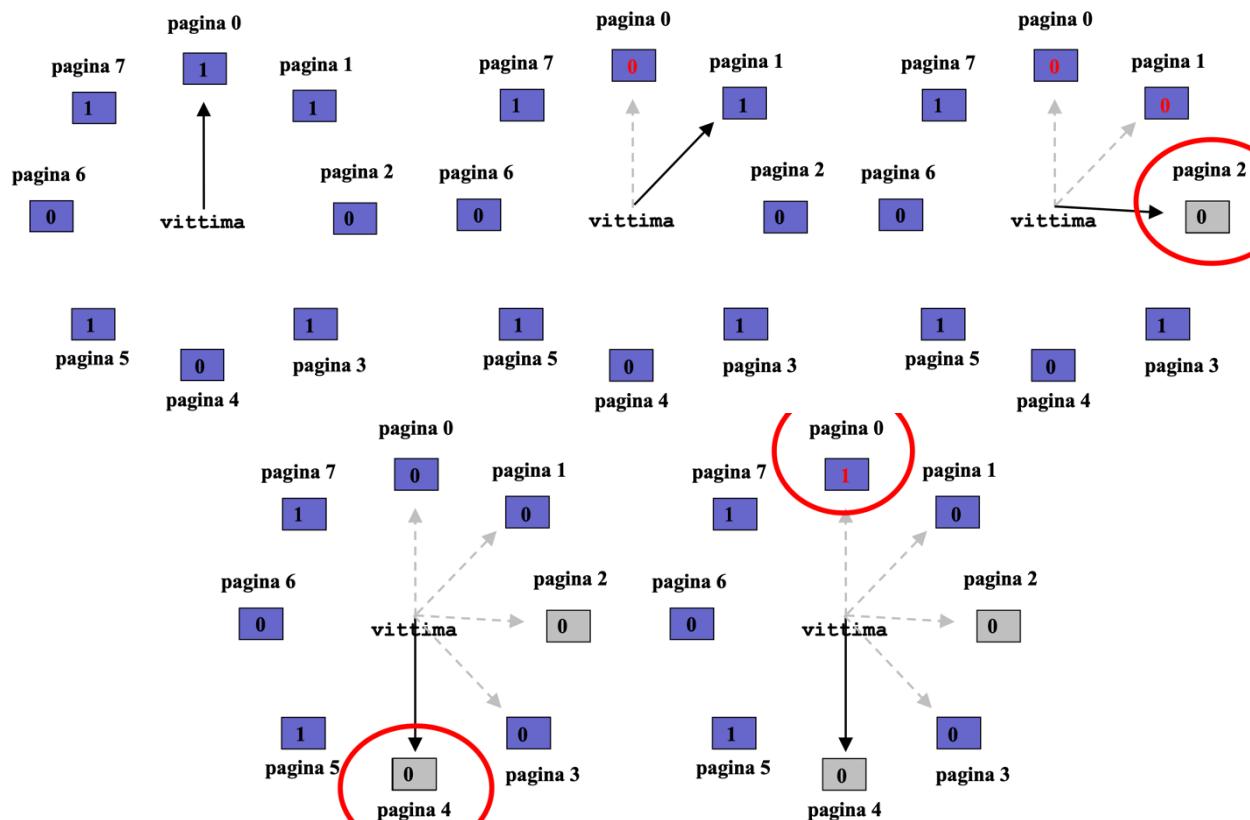


- **Second – chance**

Si itera circolarmente con un puntatore su una lista delle pagine:

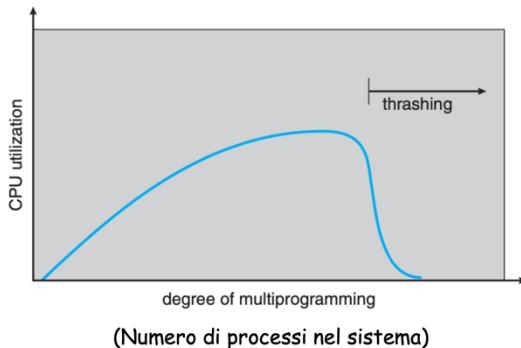
- Se la pagina candidata ha il bit di riferimento ad 1:
 - La pagina non viene sostituita;
 - Si pone il suo bit di riferimento a 0;
 - Si analizza la pagina successiva, ripetendo lo stesso controllo;
- Se la pagina candidata ha il bit di riferimento a 0, viene sostituita;
- Se la pagina viene acceduta e il bit di riferimento è a 0, lo si pone ad 1.

Ad ogni esecuzione, l'algoritmo riprende da dove si era fermato la precedente esecuzione; alla fine della lista si riprende dall'inizio.

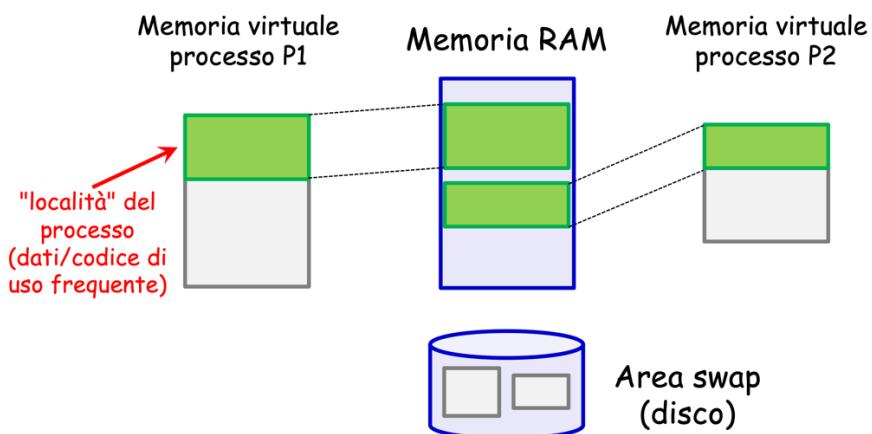


Se un processo non riceve un numero sufficiente di pagine fisiche, produrrà un **page fault** che sarà risolto andando a selezionare pagine da altri processi; ciò però fa innescare a catena un **page fault** dal processo "vittima", andando a far diminuire l'utilizzo della CPU, il che innesca, a sua volta, l'inserimento di nuovi processi che vanno ad alimentare questo circolo vizioso. L'attività

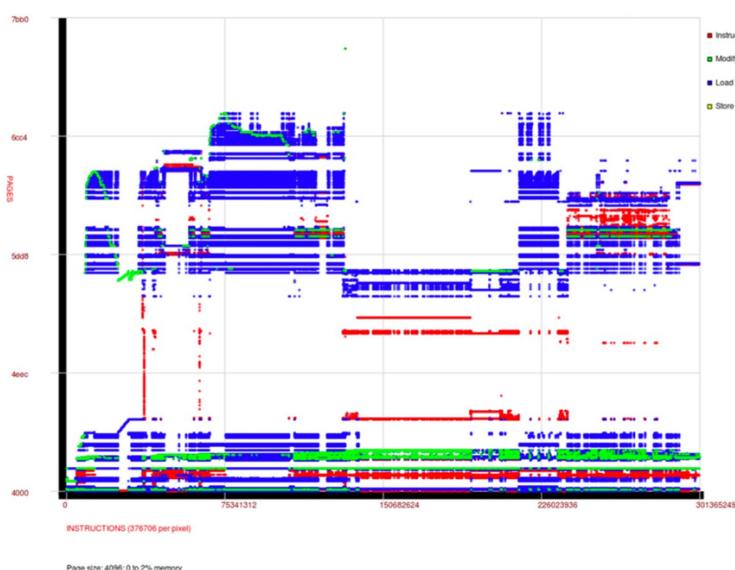
di paginazione può degenerare, così, in una situazione patologica che fa precipitare la produttività del sistema:



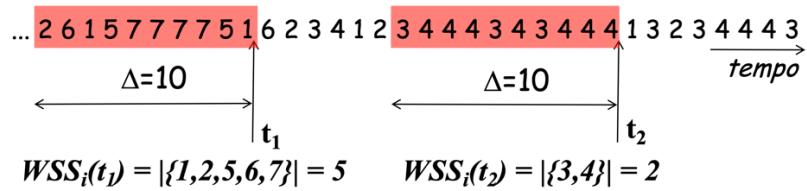
La paginazione a domanda funziona bene in virtù del principio di **località**, per il quale i processi tendono ad accedere ad un sottoinsieme di pagine della loro memoria (detto working set); per evitare il thrashing, è necessario che almeno il working set dei processi risieda in memoria.



Ad esempio, la page reference map di Firefox in Linux:

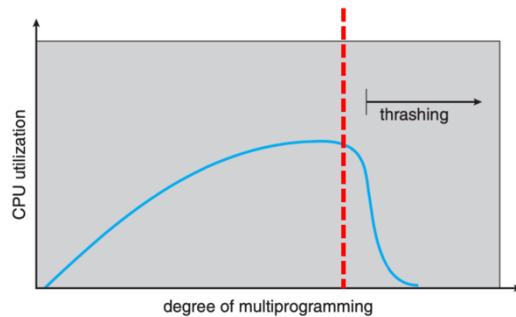


Viene detto $WSS_i(t)$ il working set del processo i all’istante t , inteso come il numero totale delle pagine cui il processo fa riferimento in un tempo Δ (finestra empirica di osservazione) e varia nel tempo in base all’attività del processo.



Definita $D = \sum_i WSS_i(t)$ la richiesta totale di memoria da parte di tutti i processi nel sistema e M la quantità totale di memoria fisica disponibile, l'attività di paginazione degenera (generando thrashing) se:

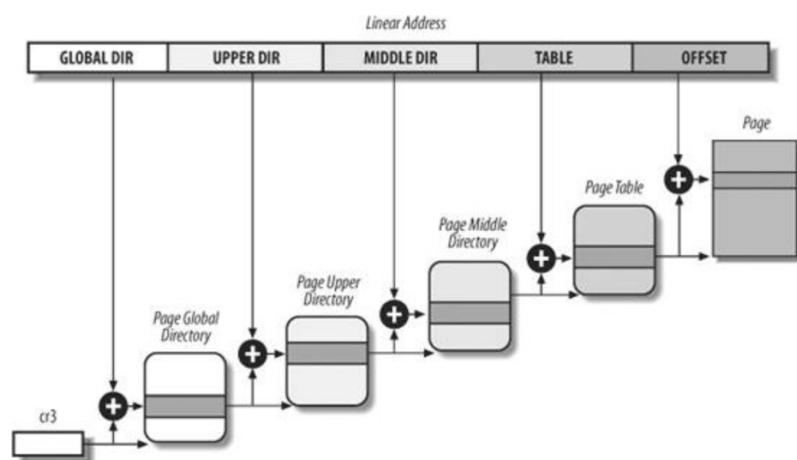
$$D > M$$



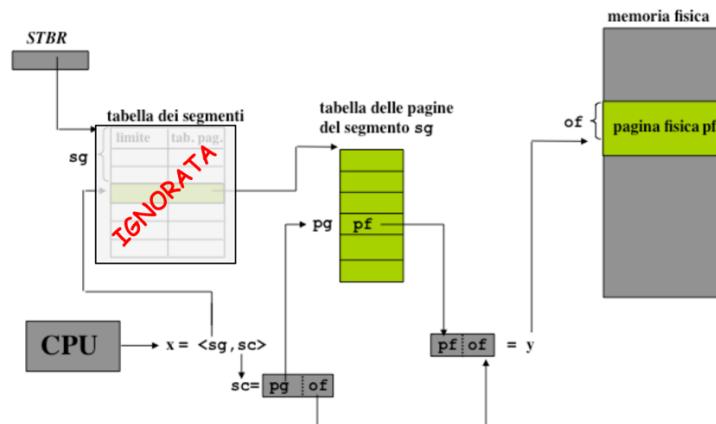
In caso di thrashing, occorre ridurre il grado di multiprogrammazione (ad esempio, sospendendo o terminando dei processi).

LA GESTIONE DELLA MEMORIA IN LINUX E IN WINDOWS

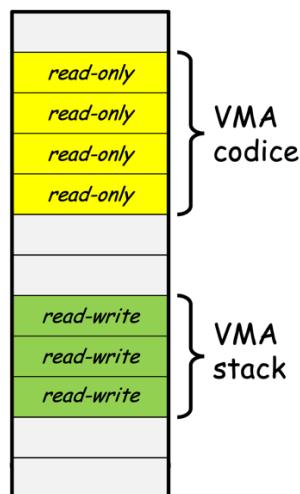
Linux adotta un sistema di **memory management paginato e segmentato**, in modo da garantire la **portabilità tra diverse architetture** e il **supporto per sistemi con grandi quantità di memoria** (NUMA, Non Uniform Memory Access) e **multiprocessore** (SMP). Il **sistema di paginazione** è **gerarchico a diversi livelli** e con **dimensione di pagina** (in x86) pari a 4kB o 4MB:



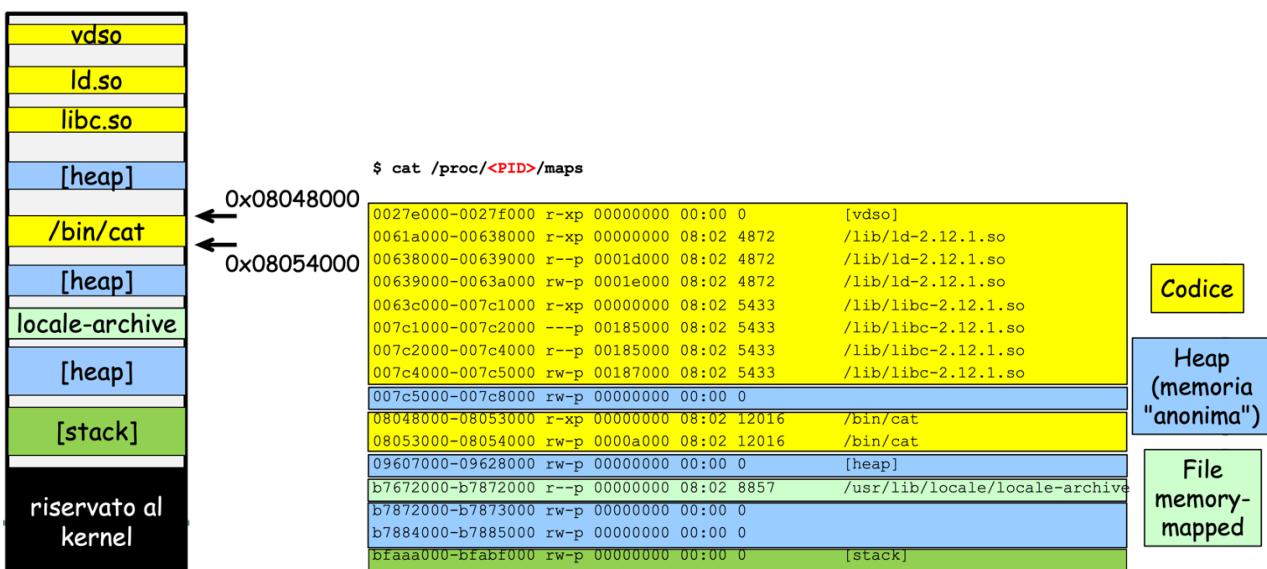
Nelle versioni moderne del Sistema Operativo, tuttavia, non ci si avvale del processore per la **segmentazione**, per motivi di **portabilità** ed **efficienza del context switch** e della traduzione degli indirizzi:

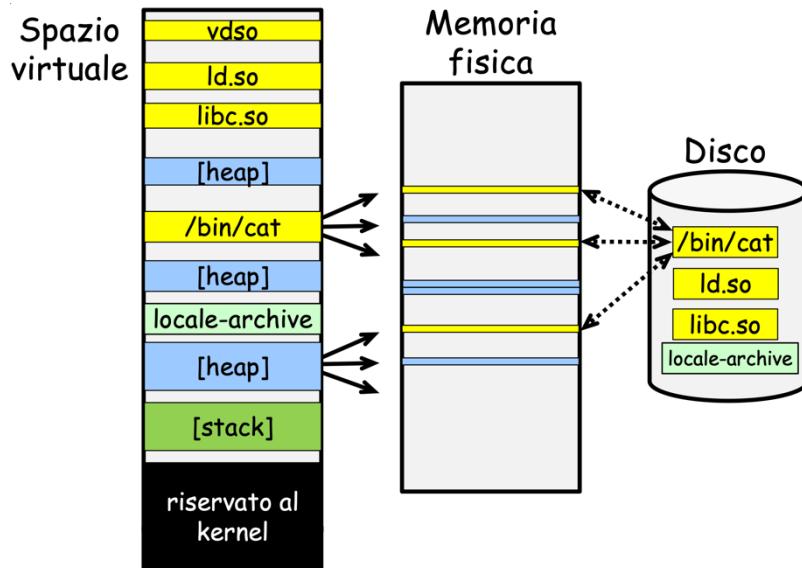


La memoria in sé e per sé è organizzata in Virtual Memory Areas (VMA), pagine virtuali contigue contenenti codice, stack, librerie, ecc... che garantiscono protezione e condivisione dei dati:



Lo spazio virtuale di un processo, invece, in Linux è strutturato come segue:



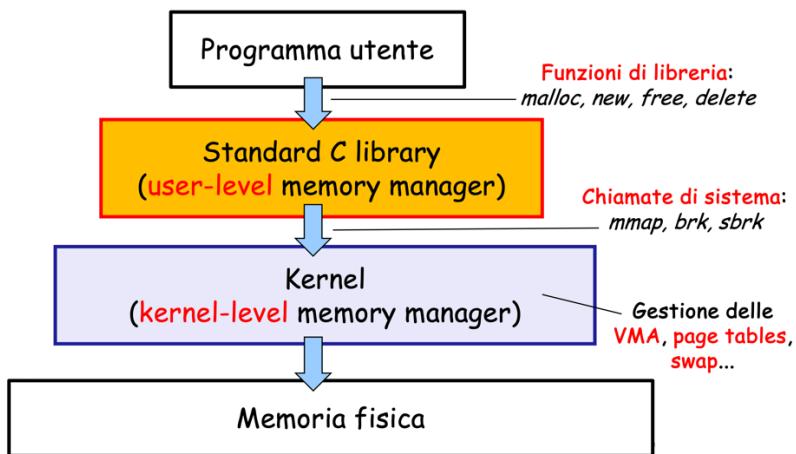


Le pagine possono essere abbinate a file su disco (memory mapped), mentre le altre sono dette anonime.

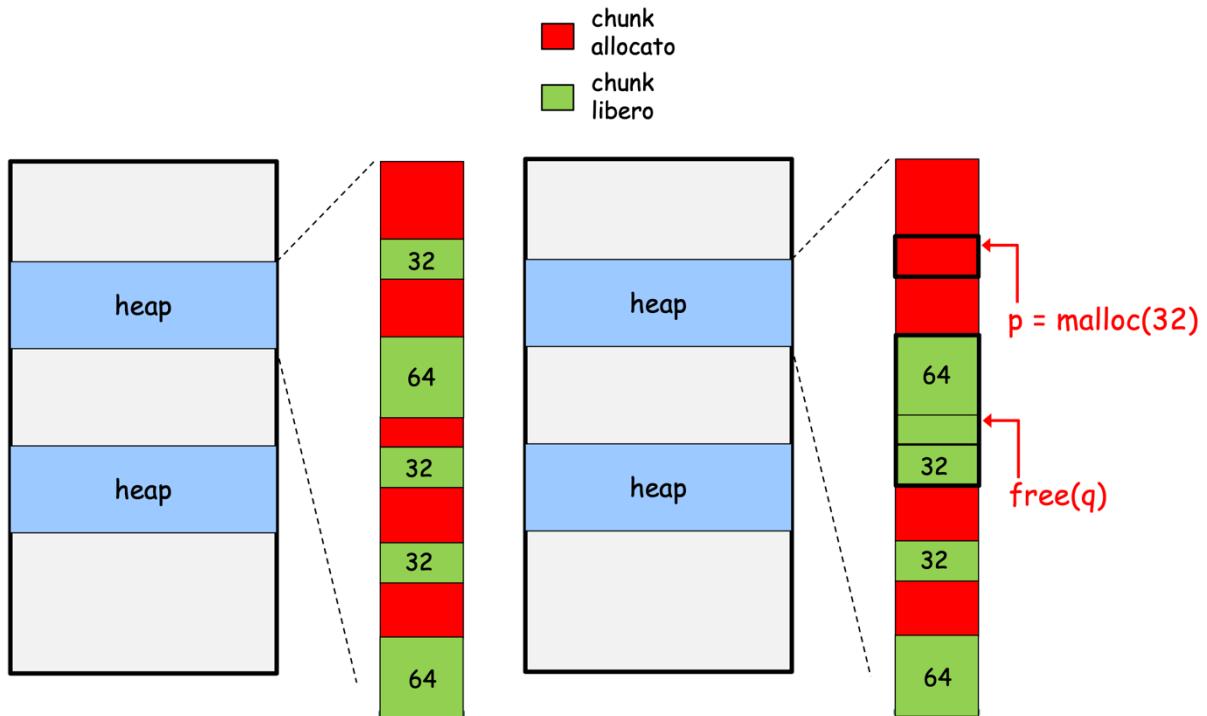
L'allocazione di memoria in Linux può avvenire su **due livelli**:

- **User – level allocator**, nella libreria C, e utilizza le funzioni `malloc()` e/o `free()` e le keywords `new` e/o `delete` per selezionare un blocco libero nell'area heap del processo;
- **Kernel – level allocator**, utilizza la funzione `malloc()` (se l'area heap è esaurita chiede al kernel nuove pagine) e le syscall `brk`, `sbrk` (per espandere una VMA) e `mmap` (per aggiungere una VMA).

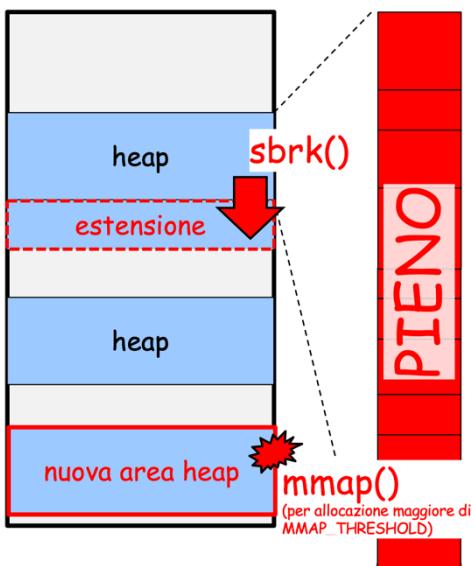
Si ripresenta anche nella gestione della memoria lo schema presentato all'inizio della trattazione per introdurre le diverse parti di un Sistema Operativo:



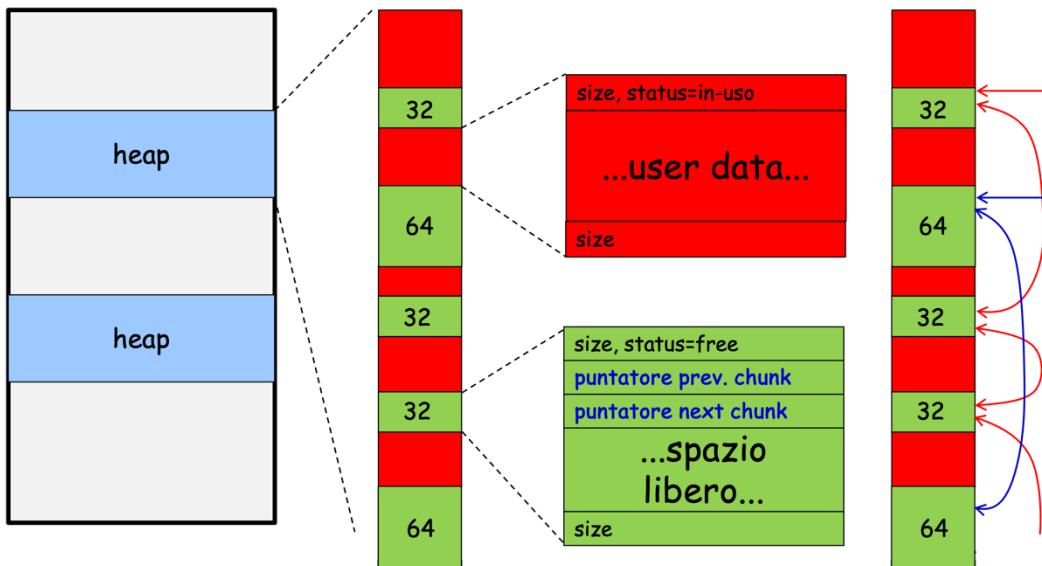
A livello utente, l'area heap alterna blocchi, di dimensione variabile, liberi e allocati; alla chiamata `malloc()`, viene cercato un blocco libero che soddisfi la richiesta (con algoritmi best-fit, ad esempio), mentre alla chiamata `free()` viene liberato un blocco, fondendolo con i vicini se possibile. Si noti che la memoria allocata non sempre è inizializzata a zero (ed è per questo che non è una buona pratica dichiarare una variabile senza inizializzarla).



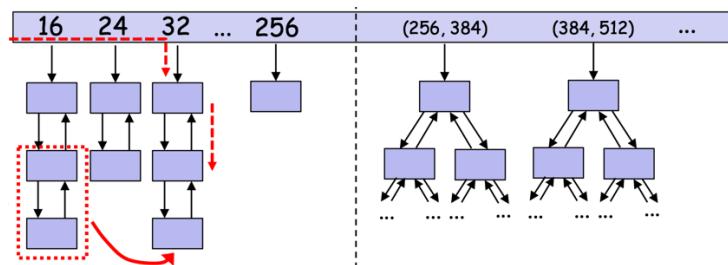
Quando, eventualmente, l'area heap si satura, le syscall introdotte in precedenza agiscono sulla dimensione della VMA:



La libreria C in Linux utilizza l'allocatore `ptmalloc`, un derivato dell'allocatore tradizionale di Doug Lea (`dlmalloc`), con la quale **ogni chunk**, allocato o meno, è strutturato come segue, tenendo anche **traccia dei blocchi di memoria libera tramite linked list**, memorizzati all'interno dei blocchi liberi stessi:

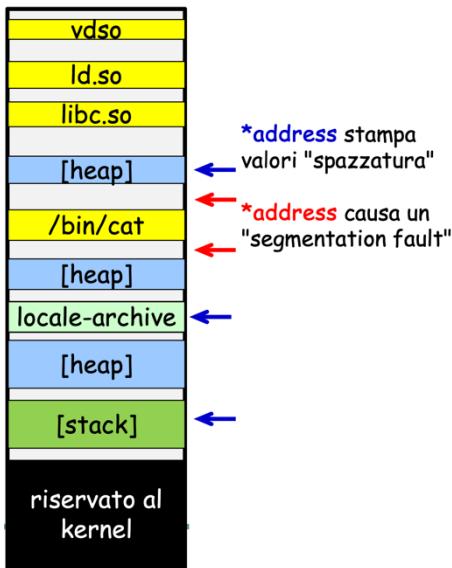


I blocchi di memoria sono organizzati in liste di chunk di dimensione fissa ma crescente (approssimativamente secondo un andamento logaritmico) ed ogni blocco è allocato con criterio best – fit with coalescing, cioè con la possibilità di fondere due chunk liberi vicini per ridurre la frammentazione in memoria; in determinati casi, per mantenere la località spaziale, si procede con un approccio next – fit piuttosto che best – fit. Per blocchi di dimensioni maggiori (ad esempio, superiori a 256), i chunk sono organizzati in liste più complesse di dimensione variabile.



La MMU verifica la validità di un indirizzo tramite la tabella delle pagine; nel caso di puntatori non validi nel programma, il processo può essere ucciso dal Sistema Operativo: la MMU genera un page fault, la ISR del Sistema Operativo verifica se l'indirizzo è incluso o meno nell'intervallo degli indirizzi (VMA) di una delle aree heap, stac, ecc... e, in caso negativo, il Sistema Operativo stesso si occupa di uccidere il processo per segmentation fault. Il motivo di questo comportamento risiede nel fatto che la prima pagina virtuale di un processo (indirizzi 0 – 4095) ha sempre i permessi di accesso disattivati, in modo da impedire qualsiasi accesso; questo approccio è utile ai fini di debugging. Un puntatore è detto non valido non solo se è NULL ma anche se non è inizializzato; tuttavia, questo comportamento è undefined nello standard del linguaggio C e il risultato di un'eventuale computazione dipende dal Sistema Operativo, dal compilatore e dalla CPU. Ci sono, però, due possibilità:

- Il puntatore (per puro caso) ha un indirizzo valido in una delle aree del processo;
- Il puntatore contiene un indirizzo non valido (fuori dalle aree).



Sebbene possa essere facile evitare direttamente NULL pointers o puntatori non inizializzati, **queste condizioni si possono verificare anche al di fuori della volontà del programmatore**; ad esempio, se l'operazione di `malloc()` fallisce (per spazio di memoria, virtuale o fisico, esaurito) viene ritornato un puntatore NULL e la seguente operazione costituisce una NULL pointer exception:

```
char* address = malloc(10)      // Nonostante la correttezza sintattica
```

In caso di **scarsità di memoria a runtime** (sia su RAM che su swap), invece, **il kernel uccide un processo (Out – Of – Memory killer)**:

```
int main() {
    void* block = NULL;
    int count = 0;
    while (1) {
        block = (void *) malloc(1024*1024); // Alloca 1MB
        if(!block) break;
        // Scrive degli "1" sulle pagine allocate
        memset(block, 1, 1024*1024);
        printf("Currently allocating %d MB\n", ++count);
    }
    printf("Done\n");
}
```

In output:

```
...
Currently allocating 2881 MB
Currently allocating 2882 MB
Currently allocating 2883 MB
Done
```

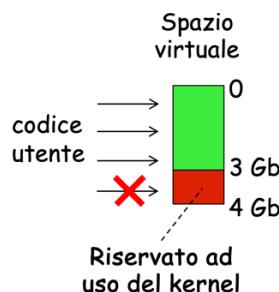
Il processo satura lo spazio su RAM e swap. Se, però, al posto di scrivere sulle pagine, si lascia inutilizzato il blocco:

```
int main() {
    void* block = NULL;
    int count = 0;
    while (1) {
        block = (void *) malloc(1024*1024); // Alloca 1MB
        if(!block) break;
        // Il blocco rimane inutilizzato
        printf("Currently allocating %d MB\n", ++count);
    }
    printf("Done\n");
}
```

In output:

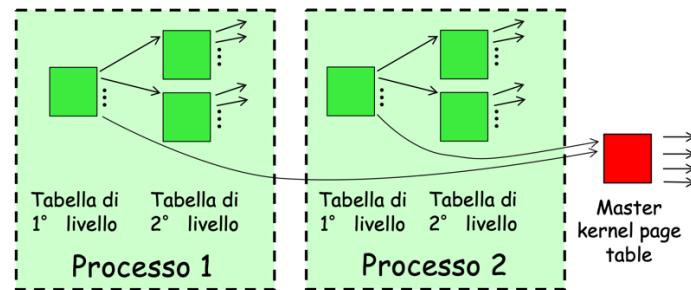
...
Currently allocating 4072 MB
Currently allocating 4073 MB
Currently allocating 4074 MB
Currently allocating 4075 MB
Done

Il processo riempie tutto il suo spazio di memoria virtuale (4GB) perché è stata assegnata memoria virtuale senza mapparla effettivamente in memoria fisica. Questo approccio è detto **Deferred (o Lazy) Memory Allocation** e prevede che **il sistema allochi la memoria fisica solo quando il programma utilizza effettivamente la memoria virtuale allocata;** nel programma appena proposto, non andando mai ad utilizzare la memoria virtuale (che arriva fino a 4GB), la memoria fisica (che arriva fino a 2GB) non viene mai allocata e il programma non termina prima, come nel caso precedente.



Dello spazio virtuale di un processo Linux, **una porzione è riservata ad un insieme di indirizzi usati dal kernel** (per le syscall, ad esempio). Anche il Sistema Operativo utilizza indirizzi virtuali, però **la MMU non utilizza una page table dedicata al Sistema Operativo per motivi di efficienza prestazionale** (il TLB flush al cambio di contesto farebbe calare le prestazioni del sistema); pertanto, **la tabella delle pagine al cambio di contesto rimane inalterata e il kernel può riutilizzare parte dello spazio virtuale del processo interrotto**, migliorando ulteriormente le prestazioni.

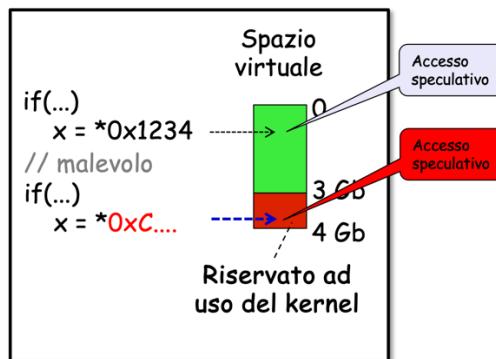
Lo spazio virtuale riservato al kernel è gestito dalla master kernel page table, uguale e condivisa fra tutti i processi:



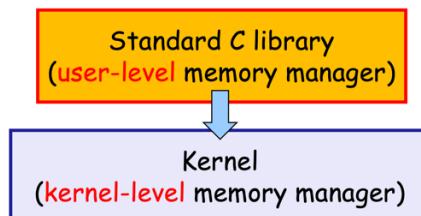
Tra gli indirizzi virtuali e quelli fisici, la master kernel page table effettua un'associazione di tipo lineare:

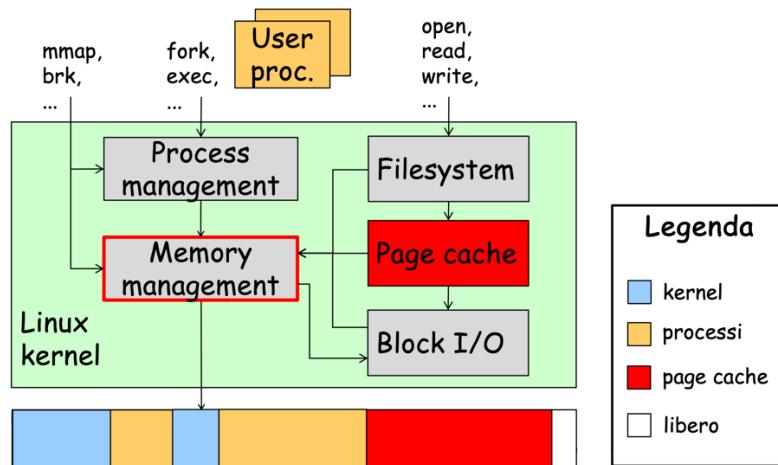
Indirizzo virtuale	Indirizzo fisico	Offset
0xC0000123	0x000000123	0xC0000000

L'approccio di user – kernel split e l'assenza di una page table dedicata al kernel sono stati causa di problemi di sicurezza; le moderne CPU fanno esecuzione speculativa, cioè tentano di “indovinare” le prossime istruzioni, eseguendole in anticipo ma lasciando tracce nella cache, nel branch predictor, ecc... in questo modo può essere acceduto lo spazio virtuale dedicato al kernel e rompere l'isolamento tanto vantato fino a questo momento. Per prevenire questo tipo di attacchi, occorre dedicare al kernel una page table separata, rallentando però le performance del sistema.

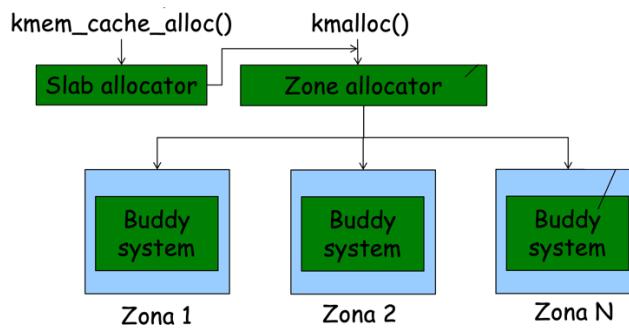


Il kernel contiene a sua volta un gestore della memoria fisica, che alloca pagine sia ai processi sia per uso interno al kernel (buffer per trasferimenti I/O, coaching di dati recenti letti/scritti sui dischi, descrittori dei processi, tabelle delle pagine, metadati del filesystem, ...):

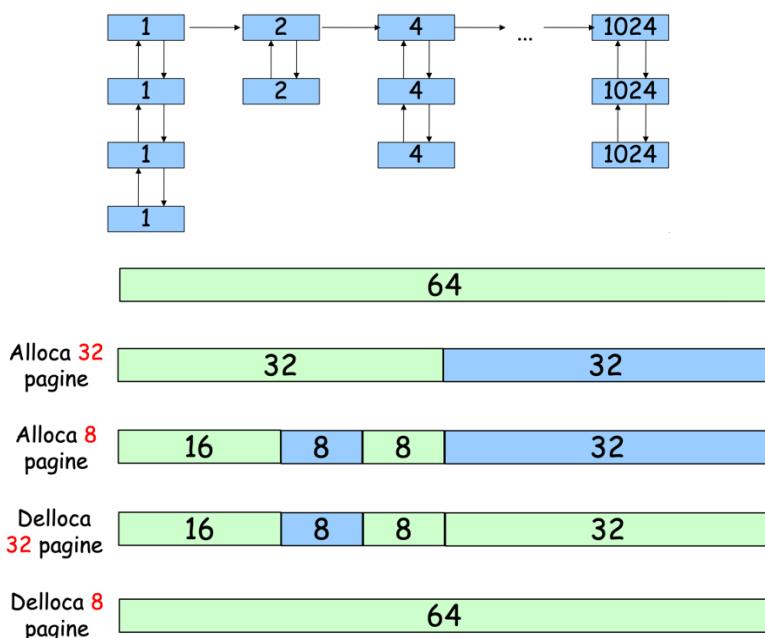




L'allocatore in kernel – mode può lavorare sia a frazioni di pagina (con `kmem_cache_alloc()`) sia a multipli di pagina (`kmalloc()`); dopo aver scelto il carico di lavoro, viene selezionata una zona di memoria fisica e vi vengono allocate le pagine:



Il **Buddy System** è un algoritmo usato per allocare gruppi di pagine fisiche contigue. Allocare pagine fisiche contigue non è necessario ma farlo permette di ridurre la dimensione della tabella (gerarchica) delle pagine, avere minor consumo di cache e di TLB e avere compatibilità con DMA e dispositivi di I/O legacy (obsoleti). Una allocazione, secondo il Buddy System, avviene per multipli di 2^k pagine contigue, con il kernel che tiene traccia dei blocchi liberi tramite le linked list (possono essere viste tramite il file `/proc/buddyinfo`).

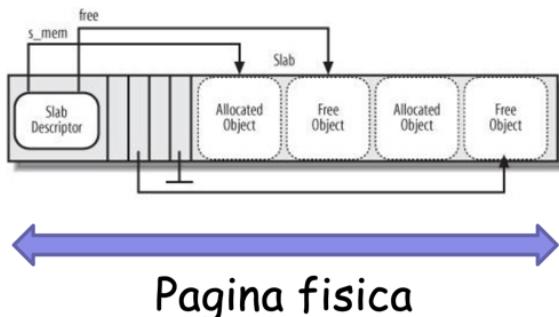


L'algoritmo, ricorsivo, può essere così schematizzato:

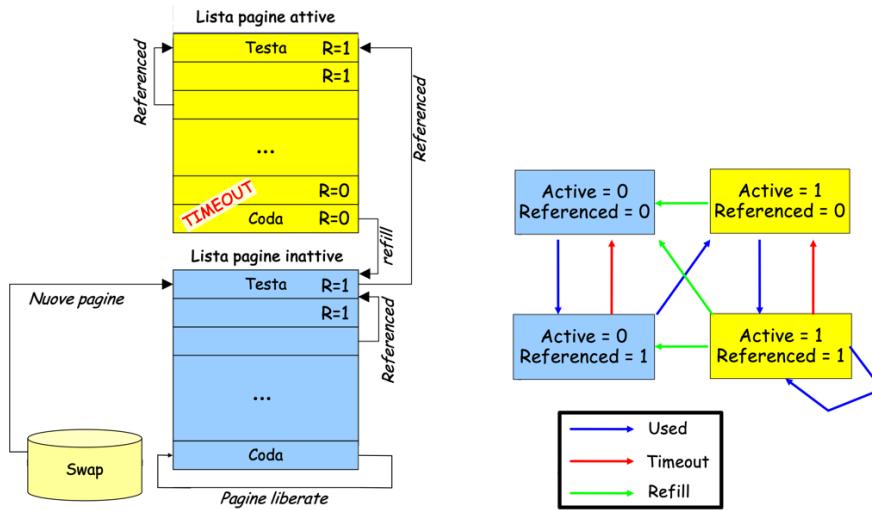
- Se un blocco di dimensione 2^k è già disponibile, la richiesta è subito soddisfatta;
- Altrimenti, si verifica se esiste un blocco di dimensione $2^{(k+1)}$:
 - Se sì, lo si divide in due parti: la prima metà viene allocata, la seconda inserita nella lista di blocchi con 2^k pagine;
 - Altrimenti, si verifica se esiste un blocco di dimensione $2^{(k+2)}$:
 - Se sì, lo si divide in quattro parti: una viene allocata, gli altri tre blocchi vengono inseriti nella lista di blocchi con $2^{(k+1)}$ pagine;
 - ...

Quando un blocco di 2^k pagine viene deallocated, si aggiunge un blocco alla lista dei blocchi di dimensione 2^k e, se vi è un altro blocco libero adiacente con 2^k pagine, i due blocchi vengono uniti, venendo anche rimossi dalla lista con i blocchi di 2^k pagine e aggiungendo un blocco libero alla lista di quelli con $2^{(k+1)}$. L'unione di più blocchi contigui viene ripetuta ricorsivamente finché possibile.

Lo Slab Allocator interviene quando si chiede all'allocatore in kernel – mode di lavorare a frazioni di pagina, dal momento in cui permette di allocare porzioni piccole di memoria di dimensione variabile tra i 32 e i 4080 byte (le pagine contengono oggetti pre – allocati di stessa dimensione) ed è utile per strutture dati piccole, come PCB, buffer di I/O, descrittori di file aperti, ecc...



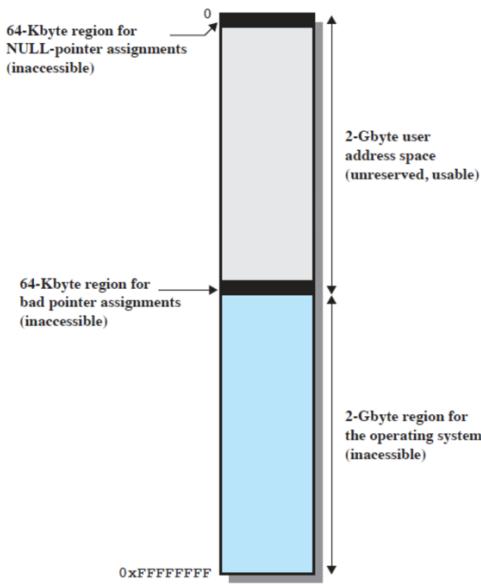
L'algoritmo di rimpiazzo delle pagine, il PFRA (Page Frame Reclaiming Algorithm), è una variante dell'algoritmo second chance che viene eseguito quando la memoria libera è al di sotto di una certa soglia o periodicamente. Le nuove pagine sono inserite in una lista globale di pagine inattive, scansionata dal PFRA che sposta le pagine referenced verso la cima; la lista "active" approssima il working set e le pagine non utilizzate sono degradate, finendo infine nella swap. Una pagina è referenced quando è stata recentemente acceduta (è un parametro azzerato periodicamente), mentre è active se è stata acceduta per più volte nel recente passato; la selezione del PFRA si concentra sulle pagine non attive.



Sulle piattaforme a 32 bit, ogni processo Windows vede 4GB di memoria; una porzione di questo spazio, circa la metà, è dedicata al Sistema Operativo ed è condivisa tra tutti i processi, lasciando 2GB di spazio di indirizzamento virtuale al singolo processo. Un'opzione, poi, consente di aumentare la memoria utente a 3GB e ridurre a 1GB quella di sistema. Sulle piattaforme a 64 bit, invece, ogni processo utente può avere fino ad 8TB di memoria virtuale.

A 32 bit, lo spazio di indirizzamento è così diviso:

- 0x00000000 – 0x0000FFFF, allocate per individuare i NULL pointer;
- 0x00010000 – 0x7FFEFFFF, allocate ai processi utente;
- 0x7FFF0000 – 0x7FFFFFFF, allocate per individuare i bad pointer;
- 0x80000000 – 0xFFFFFFF, allocate al sistema operativo.



Windows usa la paginazione su richiesta per gruppi di pagine (demand paging with clustering) con regioni di indirizzi virtuali contigui di 64kB. Ogni cluster della memoria può essere in uno dei seguenti stati:

- Available, memoria non utilizzata dal processo;

- **Reserved**, memoria messa da parte per il processo dal Sistema Operativo e preservata per successivi usi (ad esempio, lo spazio preallocato per far crescere lo stack);
- **Committed**, memoria allocata ed utilizzata dal processo, le cui pagine possono risiedere sia in memoria centrale sia sul disco (nel file di paging).

La **distinzione tra reserved e committed** è utile per **usare efficientemente lo spazio di memoria virtuale e per consentire di riservare pagine prima della loro effettiva allocazione**.

Il Sistema Operativo tiene anche traccia del numero (variabile) di pagine attribuite ad ogni processo e la sostituzione delle pagine avviene nel contesto del processo che produce assenze di pagina. Inoltre, è garantito dal Sistema Operativo un minimum working set di 50 pagine e, se vi è molta memoria disponibile, un maximum working set di 350 pagine; quando occorre un page fault, la pagina richiesta è aggiunta al processo senza rimuoverne una vecchia, accrescendo il working set. Se la memoria libera, poi, scende sotto una soglia critica, si applica una regolazione automatica (Automatic Working Set Trimming) per riportare il valore sopra la soglia, riducendo i processi che hanno ottenuto un numero di pagine superiori a quello minimo o che sono stati in idle per molto tempo e non hanno acceduto alle pagine del working set. La gestione del working set appena mostrata è fatta da una routine denominata **Working Set Manager**, eseguita nel contesto di un thread di sistema (Balance Set Manager Thread).

I/O E GESTIONE DEI DISCHI

Il sistema di I/O, in particolare lo **storage**, è spesso un bottleneck delle prestazioni e causa di perdita di dati; generalmente gli indicatori che permettono di quantizzare queste proprietà sono i bassi tempi di risposta (ovvero, bassa latenza), un alto numero di richieste servite (throughput, o rendimento), un'alta percentuale di utilizzo dei dispositivi e l'equità (no starvation).

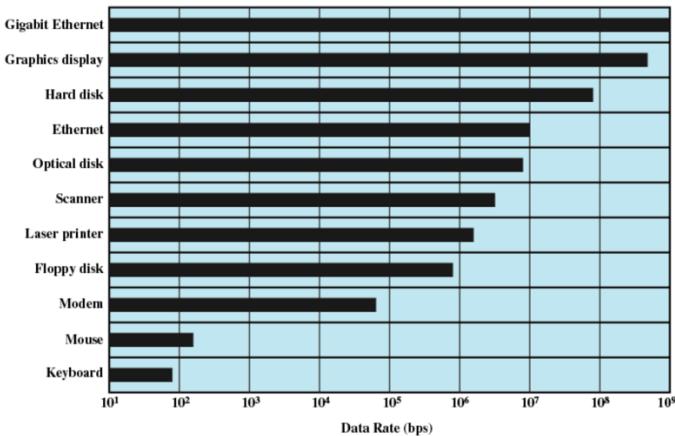
Le periferiche di I/O possono essere categorizzate nelle seguenti classi:

- **Human readable**, cioè che prevedono l'interazione con l'utente (come tastiere, schermi, mouse, ecc...);
- **Machine readable**, cioè che prevedono l'interazione con i dispositivi locali (come storage, sensori, ecc...);
- **Comunicazione**, cioè che prevedono l'interazione con dispositivi remoti (come schede di rete, modem, ecc...).

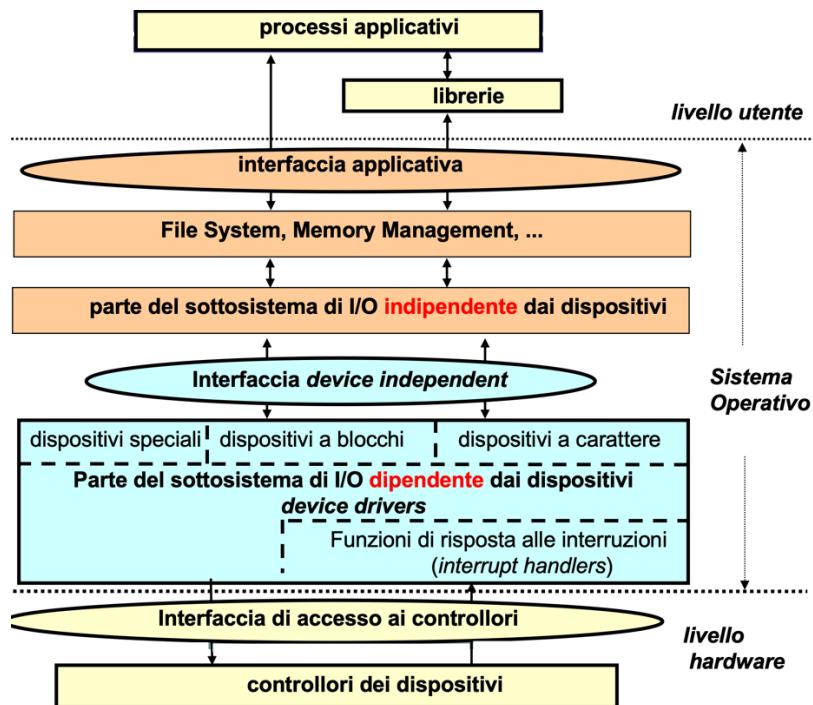
La categorizzazione, tuttavia, può essere fatta anche sulla base dei dati a cui si accede:

- **Character devices**, prevedono l'accesso per flussi di dati sequenziali a piccole unità, ad esempio byte (come tastiere, porte seriali, ecc...);
- **Block devices**, prevedono l'accesso a posizioni arbitrarie, per blocchi, ad esempio 1kB (come HDD, USBD, SD card, ecc...).

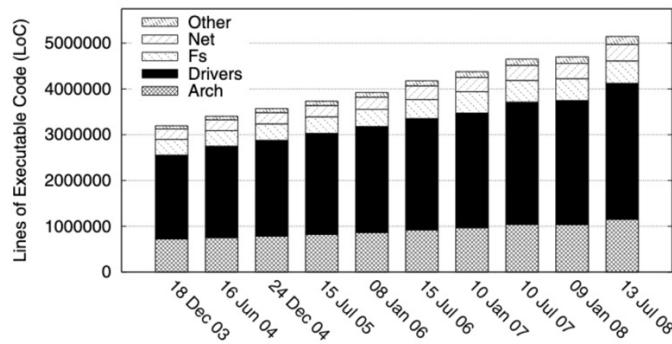
In realtà, le categorizzazioni non terminano qui, dal momento in cui i dispositivi di I/O sono numerosi ed eterogenei, nonché estremamente lenti; tenendo, poi, in considerazione che le applicazioni sono sensibili alla latenza, il tema della gestione delle periferiche è cruciale.



Il sistema di I/O di un Sistema Operativo fornisce un’astrazione dei dispositivi ai programmi utente, in modo che questi accedano ai dischi utilizzando **sempre le stesse primitive** (read, write, open, close, ...), **indipendentemente dalla tecnologia dei dispositivi** e nascondendone l’eterogeneità intrinseca. Questa modularità è un vantaggio anche nel momento in cui si vuole aggiungere, ad esempio, il supporto ad un nuovo tipo di dispositivo, dovendo solo caricare un modulo nel kernel.

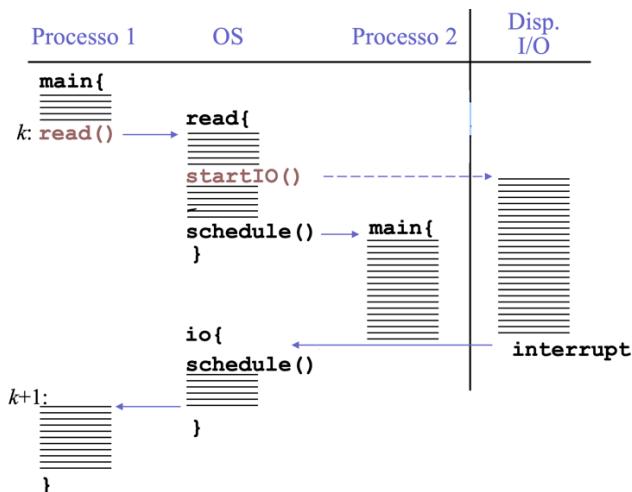


L’ultimo baluardo **prima del livello hardware** è il **responsabile dell’astrazione appena menzionata**; infatti, se a livello dell’interfaccia applicativa è comandato di “Leggere un file in posizione X”, questa istruzione è passata alla parte di sottosistema di I/O indipendente dai dispositivi come insieme di primitive, “Leggi il blocco alla posizione Y del disco”, per poi arrivare al blocco in questione come informazione specifica per quella particolare periferica, “Leggi il settore W sulla traccia Z del disco”. Per quest’ultimo passaggio **sono essenziali i device drivers**, programmi specifici per il singolo dispositivo di I/O (come ISR, indirizzi dei registri, comandi di I/O) e **rappresentano la maggior parte del Sistema Operativo** (circa il 70% del codice di Linux) nonché **tra le maggiori cause di malfunzionamenti**.



Ciò che, in sé e per sé, permette l'**astrazione**, come già annunciato, è la parte del **sottosistema di I/O indipendente dai dispositivi**. La parte in questione si occupa del naming, della virtualizzazione delle periferiche, dell'allocazione dei dispositivi e spooling, dell'I/O scheduling, caching e buffering e della gestione dei guasti; si pone, quindi, come **vero e proprio ponte tra le applicazioni utente e i dispositivi di I/O**.

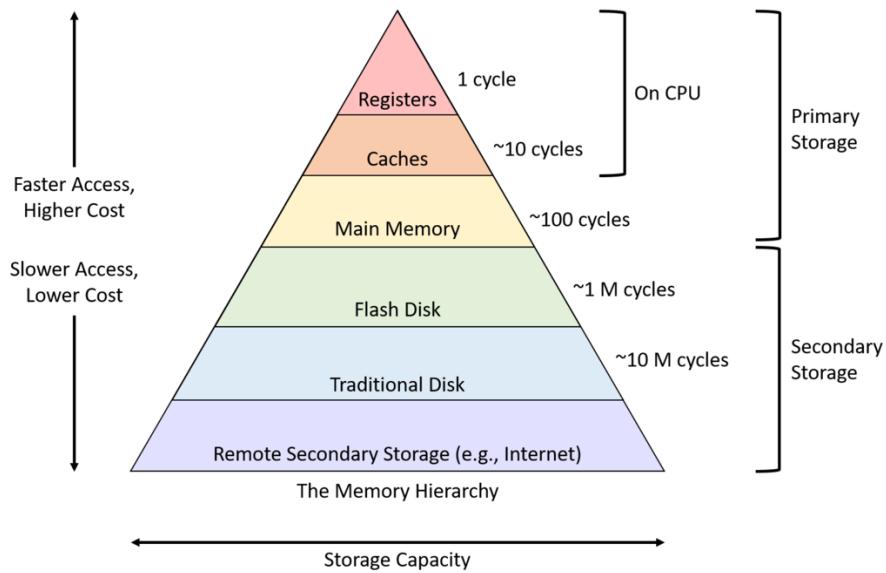
Così come è stato necessario sincronizzare più processi in architetture multiprocessore, è altrettanto necessaria la sincronizzazione nei device drivers; in particolare, va sospeso il processo richiedente durante le operazioni di I/O, così che un altro processo può occupare la CPU e le altre risorse senza perdita di produttività.



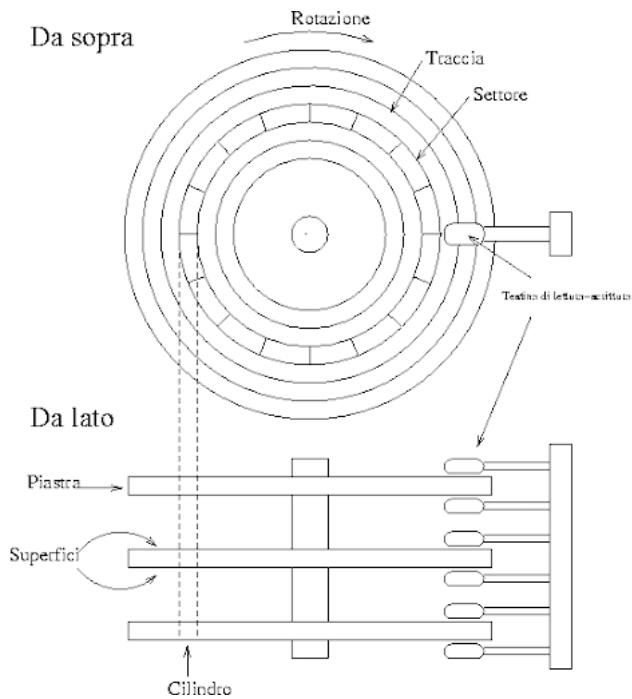
Un processo sta eseguendo, il driver di un dispositivo avvia l'I/O e il processo chiamante viene messo in sospensione con WAIT su un semaforo (inizializzato a 0); a questo punto, un secondo processo viene attivato e, in contemporanea, la ISR del device esegue. Quando quest'ultima ha terminato, si sospende il secondo processo e riattiva il flusso precedentemente sospeso con SIGNAL.

Tra tutti i dispositivi di I/O, i **dischi risultano essere quelli più critici** a causa delle **operazioni di salvataggio dei file sul filesystem**, di **gestione della memoria virtuale** (swapping) e di **gestione delle aree di transito verso altre periferiche** (spooling). I dischi, tuttavia, sono necessari per la loro **estesa capacità** (anche al costo di ridotte prestazioni), per la loro **non volatilità** e per il loro **costo ridotto**.

	GB/dollar	dollar/GB
RAM	0.1	\$9.5
Disks	18	50¢

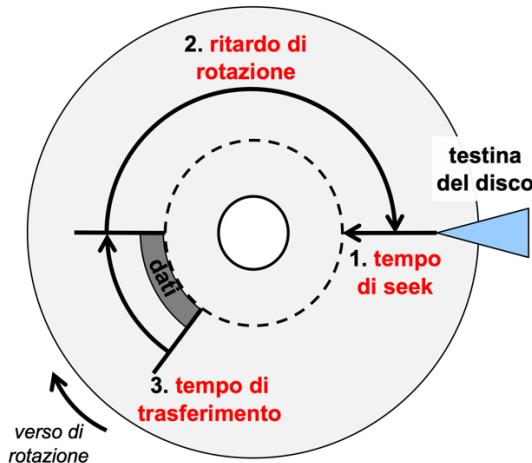


Per comprendere la **struttura di un disco** (in particolare di un disco rigido, o HDD), si faccia riferimento alla seguente figura:



I cui **principali parametri prestazionali** sono:

1. **Tempo di seek**, il tempo di ricerca per posizionare la testina sulla traccia posizionata;
2. **Ritardo di rotazione**, il tempo necessario affinché l'inizio del dato ruoti presso la testina;
3. **Tempo di trasferimento**, il tempo necessario alla rotazione dell'area del dato sotto la testina.



Il **tempo di trasferimento totale** è quantificabile come:

$$T = T_s + T_{rl} + T_{tr}$$

Con:

$$T_{rl} = \frac{1}{2r}$$

$$T_{tr} = \frac{b}{rN}$$

Denotando con **r** il numero di rotazioni al secondo, **b** i byte da trasferire e **N** il numero di byte di una traccia. Il **ritardo di rotazione** T_{rl} è definito come, in media, il tempo necessario ad effettuare la metà (visto che nel caso medio il settore a cui accedere si trova in posizione diametralmente opposta) di una rotazione completa, che è $1/r$, mentre il **tempo di trasferimento** T_{tr} è definito come il tempo di una rotazione completa moltiplicato per la frazione della traccia da percorrere, che è b/N .

Volendo calcolare il **tempo di trasferimento totale di un file di 2500 settori** (1,22MB circa), con un disco di 15000rpm, settori da 512byte, 500 settori per traccia e con un tempo di seek di 4ms, nei **due seguenti casi**:

1. Il file è memorizzato in maniera compatta, occupando cinque tracce adiacenti (**organizzazione sequenziale**)

Visto che a 15000rpm sono effettuate 250 rotazioni al secondo, il tempo di lettura sulla prima traccia è:

$$T_1 = 4ms + \frac{1}{2 \cdot 0.250} ms + \frac{1}{0.250} ms = 10ms$$

Le tracce rimanenti possono essere lette consecutivamente, quindi senza seek time:

$$T_{i \in [2,5]} = \frac{1}{2 \cdot 0.250} ms + \frac{1}{0.250} ms$$

Per un tempo di trasferimento totale di:

$$T_1 + \sum_{i \in [2,5]} T_{i \in [2,5]} = 34ms = 0.034s$$

2. Il file è memorizzato in settori distribuiti randomicamente sul disco (**organizzazione casuale**)

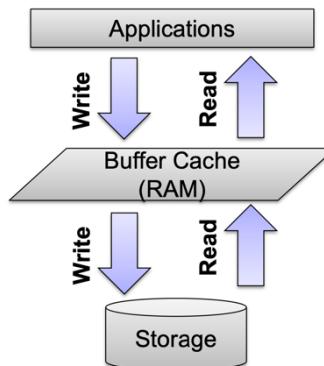
Visto che a 15000rpm sono effettuate 250 rotazioni al secondo e che il tempo di un 500 – esimo di rotazione è $1/(500 \cdot 0.250)ms$, il tempo di lettura di un singolo settore è:

$$T_i = 4ms + \frac{1}{2 \cdot 0.250}ms + \frac{1}{500 \cdot 0.250}ms = 6.008ms$$

Per un tempo di trasferimento totale di:

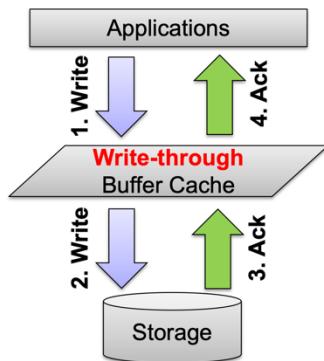
$$T = 2500 \cdot T_i = 2500 \cdot 6.008ms \approx 15s$$

Per accedere ed utilizzare i dati memorizzati su un disco, un'applicazione necessita di un'area di memoria principale che contenga una copia dei dati in transito da e verso il disco; l'area in questione è detta **Buffer – Cache**. Buffer perché i dati sono dapprima copiati in memoria, posticipando l'eventuale scrittura sul disco, e Cache perché conserva i dati recenti in RAM, in modo da velocizzare il processo di lettura in caso di secondo accesso alla stessa informazione.

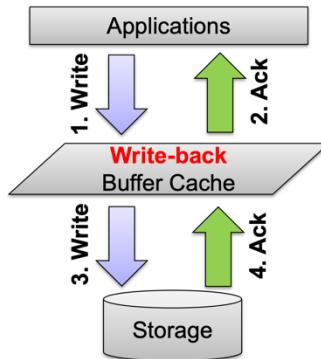


In realtà, il modo in cui il buffering di un disco è effettuato non è univoco, bensì può seguire due politiche:

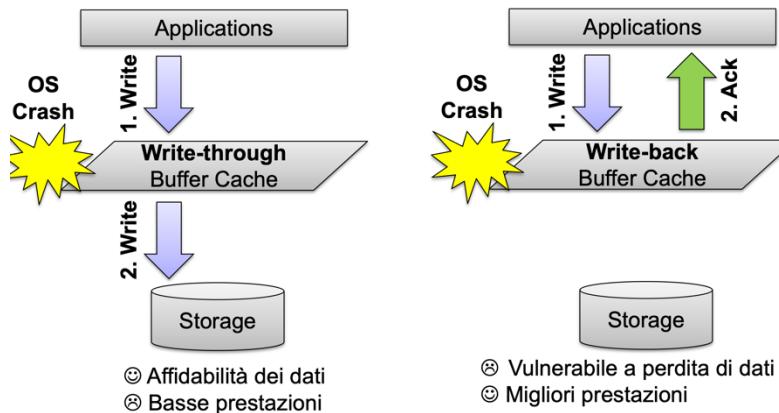
- **Write – through**, per la quale i dati vengono scritti sia su buffer sia su disco immediatamente, ad ogni modifica;



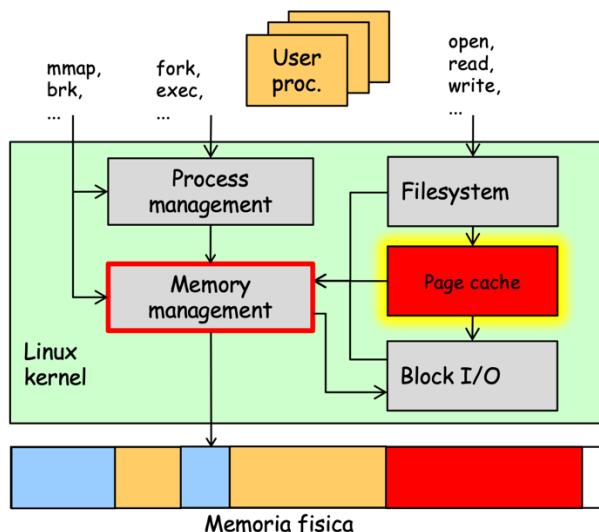
- **Write – back**, per la quale i dati non sono scritti subito su disco ma svuotati (flush) periodicamente, o quando si accumulano.



La politica **write – back** ha migliori prestazioni ma può causare la perdita di dati in caso di guasti come power outage:



In Linux, tipicamente, tutta la memoria RAM residua non usata dai processi viene dedicata alla **page cache**:



Si può, artificialmente, riempire la page cache con il seguente comando, cercando tutti i file in /usr, leggendone il contenuto con cat, ignorando l'output con /dev/null ma facendo mantenere una copia in page cache al kernel:

```
find /usr -type f -exec cat {} \; > /dev/null
```

È anche possibile **modificare alcuni parametri del page caching in Linux**:

- **Tempo massimo dopo cui un blocco “sporco” in memoria viene aggiornato su dico** (30 secondi nell'esempio):

```
# echo 3000 > /proc/sys/vm/dirty_expire_centisecs
```

- **Percentuale di memoria “sporca” oltre la quale forzare l'aggiornamento su disco** in modo da favorire o meno applicazioni “write – heavy” (10% nell'esempio):

```
# echo 10 > /proc/sys/vm/dirty_background_ratio
```

- **Swappiness**, per regolare l'algoritmo di selezione delle pagine vittima per lo swap out (al minimo, tende a spostare in swap le pagine della buffer/cache per preservare i processi, al massimo, tende a spostare in swap le pagine dei processi per preservare la page cache e l'I/O):

```
# echo 60 > /proc/sys/vm/swappiness
```

I/O SCHEDULING

Il Sistema Operativo riceve richieste di I/O da più processi differenti (grazie ai principi di multiprogrammazione) **che accedono a file differenti**; per **I/O scheduling** si intende quella pratica, messa in atto dalla parte del sottosistema di I/O indipendente dai dispositivi, con la quale: **si pongono le richieste in arrivo in una coda, si seleziona l'ordine con cui servire tali richieste e si ottimizzano le prestazioni, cercando di evitare la starvation.**

Per un disco, ci sono essenzialmente due politiche di scheduling:

- **Selezione secondo il richiedente** (adotta algoritmi FIFO, a priorità e LIFO);
- **Selezione secondo l'elemento richiesto** (adotta algoritmi SSTF, SCAN, C – SCAN, N – step – SCAN e FSCAN).

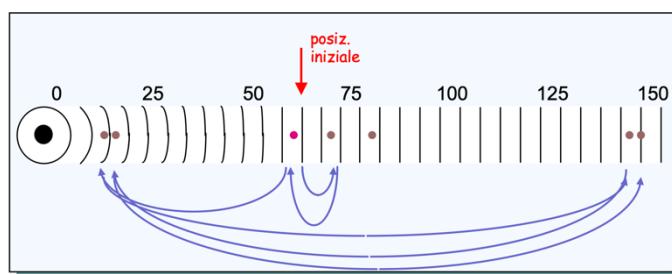
Si vogliono approfondire, in relazione all'I/O scheduling di un disco, gli algoritmi:

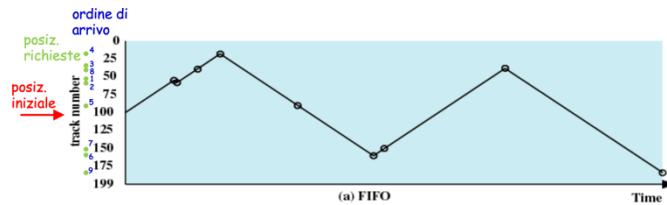
- **FIFO**

Le richieste vengono servite in maniera sequenziale, secondo una politica fair su tutti i processi; ha basse prestazioni in presenza di un alto numero di richieste, soprattutto perché richiede molti movimenti superflui della testina del disco:

Coda richieste:

83	72	14	147	16	150
----	----	----	-----	----	-----





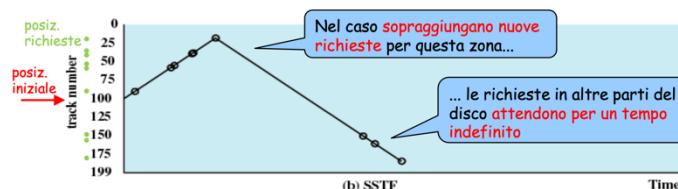
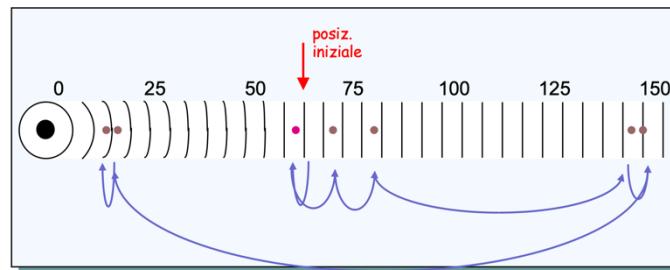
- **A priorità**

È progettato come FIFO ma i processi sono organizzati in diverse code a seconda della loro priorità (ad esempio, si privilegiano i processi interattivi); questo algoritmo, può indurre starvation per i processi a bassa priorità.

- **SSTF (Shortest Service Time First)**

Seleziona la richiesta che richiede il minimo movimento dei braccetti del disco rispetto alla posizione corrente (minimo seek time). È un algoritmo ottimo per le prestazioni ma non per la sua possibilità di indurre starvation; infatti, se arrivano molte richieste relative ad una zona del disco, l'algoritmo trascura le altre:

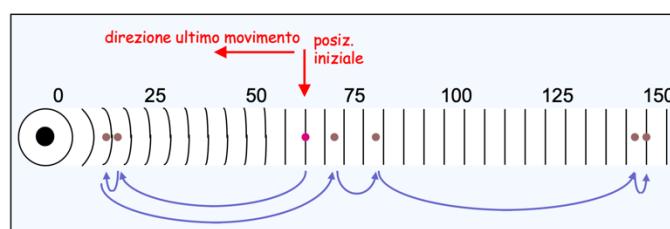
Coda richieste: [83 | 72 | 14 | 147 | 16 | 150]



- **SCAN (o algoritmo dell'ascensore)**

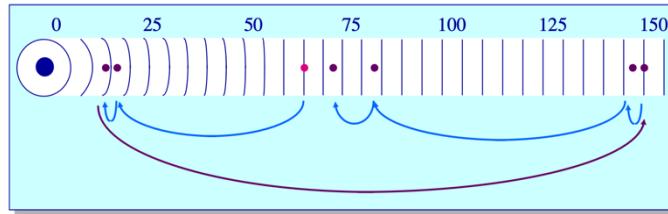
I bracci del disco si muovono solo in una direzione, soddisfacendo tutte le richieste pendenti; in particolare, si decide una direzione in cui far muovere il braccio, fino a fargli raggiungere l'ultima traccia, a partire dalla quale la direzione viene invertita:

Coda richieste: [83 | 72 | 14 | 147 | 16 | 150]



- **C – SCAN (Circular SCAN)**

Applica l'algoritmo SCAN, precedentemente illustrato, secondo un'unica direzione; quando è raggiunta l'ultima traccia, i braccetti del disco portano la testina alla parte opposta del disco, ricominciando così lo scanning.



- **N – step – SCAN**

Sono implementate più code di richieste, di lunghezza N, che vengono servite una alla volta secondo l'algoritmo SCAN; mentre la coda corrente è servita, le nuove richieste sono accumulate in un'altra coda.

- **FSCAN (Freeze SCAN)**

Sono implementate due code, senza limite numerico di dimensione. All'inizio, l'algoritmo SCAN serve le richieste di una coda, mentre l'altra, inizialmente vuota, è dedicata alle nuove richieste entranti.

Riassumendo:

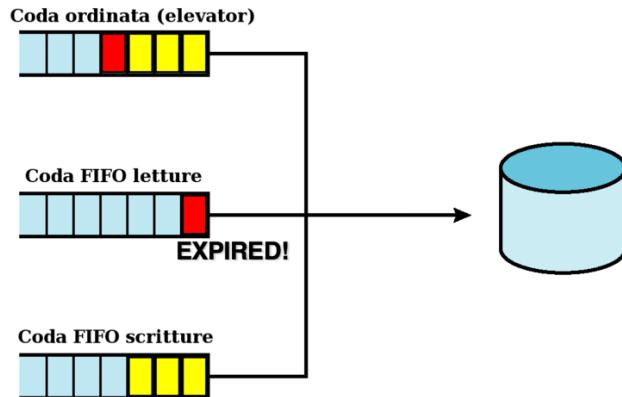
(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	Average seek length	27.5	Average seek length	27.8	Average seek length	35.8

Linux, dalla v2.4, ha adottato l'algoritmo C – SCAN, detto **Linus Elevator** e opera tramite:

- **I/O sorting**, con cui si ordinano le operazioni secondo il blocco richiesto;
- **I/O merging**, con cui si unificano due operazioni su due blocchi vicini.

Gli sviluppatori di Linux riscontrarono una **particolare forma di starvation**, detta **writes – starving – reads**, per la quale la quantità di scrittura è molto superiore a quella di lettura, penalizzando queste ultime. Le scritture sono **bufferizzate** (ad esempio, con una politica write – back) ma un processo può fare molte scritture senza sospendersi per attendere la fine delle operazioni; le letture, invece, non sono bufferizzate (sono operazioni sincrone) e un processo che legge si deve

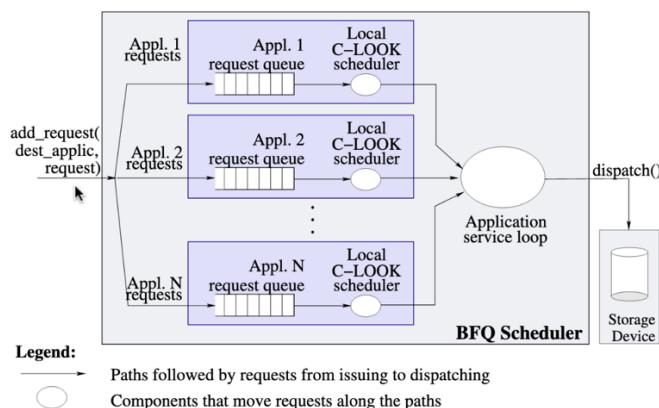
sospendere perché ha bisogno del dato per continuare ad eseguire. Il Linus Elevator è stato modificato, aggiungendovi due code FIFO, una per le letture e una per le scritture, in modo che ognuna abbia un **tempo massimo di attesa** (deadline, 500ms per le letture e 5s per le scritture, in modo da favorire le prime), scaduto il quale la richiesta è servita immediatamente:



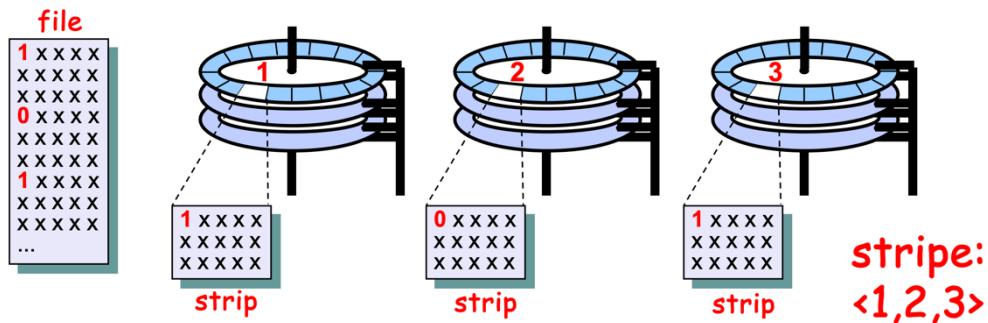
Tuttavia, **questo sistema introduce un ulteriore problema con le letture**: dopo aver letto un blocco, il processo si riattiva, elabora il dato appena letto (“think time”) e, nel frattempo, **il braccio del disco si sposta**; il lettore a questo punto effettua **una nuova lettura**, vicina al blocco appena letto in precedenza, ma, **poiché il braccio si è spostato, i tempi di seek sono allungati**. Per risolvere questo problema è possibile implementare un **meccanismo di scheduling anticipatorio**, per il quale **dopo che è stata terminata una lettura, si tiene fermo il disco per brevi periodi di tempo** in modo da **attendere eventuali altre letture** senza servire altre richieste; sebbene possa sembrare controtuitivo, **le prestazioni complessive del sistema migliorano** e prevengono il fenomeno dei writes – starving – reads. Volendo comparare il tempo medio di compilazione dei sorgenti del kernel di Linux utilizzando dapprima il meccanismo di deadline e anticipazione e poi il Linus Elevator:

	Deadline + Anticipatory	Linus Elevator
Tempo medio di compilazione dei sorgenti del kernel Linux	7.149 s	55.057 s

Le versioni più recenti di Linux utilizzano un **algoritmo di scheduling denominato BFQ** (Budget Fair Queueing), che **incorpora le tecniche degli algoritmi precedentemente illustrati**: le richieste sono separate su **più code**, **una per ogni processo**, ad ognuna delle quali è assegnato un **budget di numero di settori da attraversare**, proporzionale alla priorità.



Finora anche parlare di disco è stato un'astrazione, vista l'infinita quantità di dischi differenti che possono esistere. Una variante non di poco conto è il RAID, Redundant Array of Independent Disks, un insieme di dischi visto come un'unica entità logica dalle applicazioni; l'utilità di dischi ridondanti risiede nell'aumento delle prestazioni, grazie alla possibilità di accessi paralleli, e dell'affidabilità, grazie alla possibilità di progettare la ridondanza dei dati (alla quale è dedicata una parte dello spazio dei dischi, sotto diverse forme). I dati di ogni blocco logico sono divisi su più blocchi fisici (strip o chunks) di stessa dimensione, distribuiti su più dischi e con la possibilità di essere ridondati; una stripe, poi, si definisce come insieme di strip su posizioni omologhe:



La dimensione degli strip condiziona le prestazioni del sistema: con strip piccoli, un singolo file tende ad essere sparpagliato su più dischi, riducendo il tempo medio di servizio delle singole richieste, mentre con strip grandi, un file può entrare per intero in un singolo strip su un singolo disco, massimizzando il throughput di richieste servite.

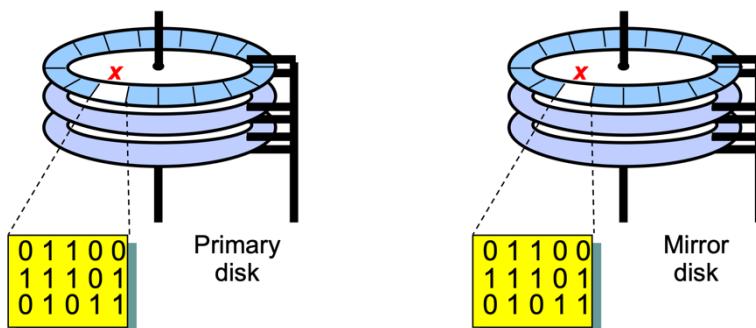
Si analizzino i diversi tipi di ridondanza dei dati che possono essere implementati con i RAID:

- RAID 0

Il sistema di RAID può anche essere ingegnerizzato per non prevedere la ridondanza (RAID 0), in modo da puntare tutto sulla maggiore efficienza prestazionale con il lavoro parallelo su più dischi (dischi fisici = quantità di dati logici = N); il data striping senza ridondanza, però, non garantisce l'integrità dei dati se uno dei dischi subisse un guasto.

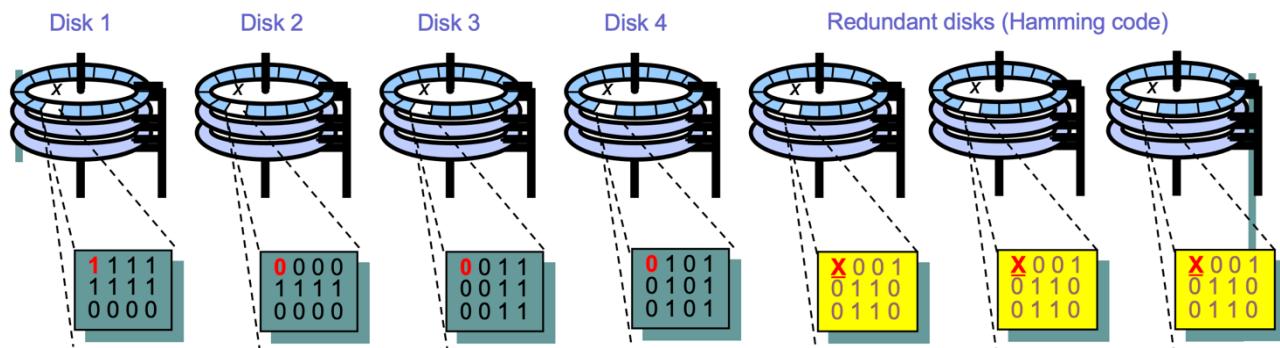
- RAID 1

Una prima tecnica di ridondanza prende il nome di mirroring e può essere implementata con il RAID 1, ovvero con un insieme di due dischi, uno primario e uno di mirroring, con la possibilità di copiare su entrambi lo stesso dato. Questa tecnica migliora il throughput e i tempi di servizio (più di RAID 2, RAID 3, RAID 4 e RAID 5 ma minore di RAID 6), evitando le penalità degli altri RAID e richiedendo un numero di dischi doppio rispetto alla quantità di dati logici (2N).



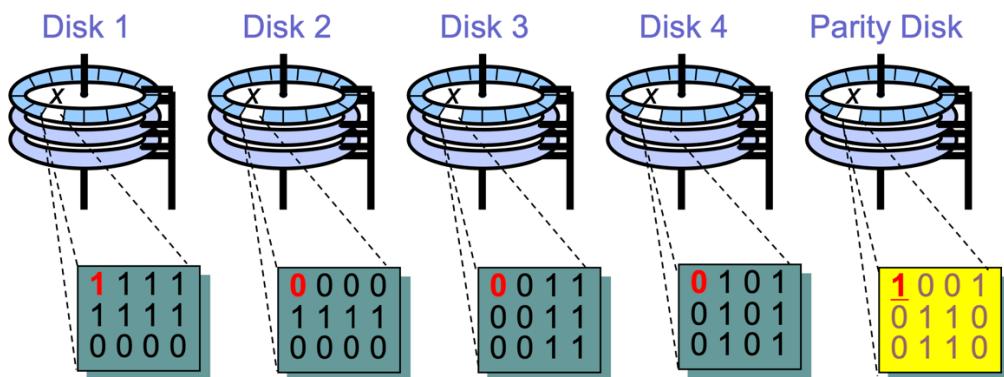
- RAID 2

Lo striping avviene a livello di bit (quindi il primo bit sul primo disco, il secondo bit sul secondo disco e così via) e la ridondanza avviene tramite codici di Hamming. Questa configurazione, detta anche a codici di Hamming, ha un'elevata affidabilità (migliore di RAID 3 e RAID 4) ma ha un basso tempo di servizio (come RAID 3) e sono richiesti un numero $\log N$ di dischi in più rispetto alla quantità di dati logici ($N + \log N$). Infine, questa configurazione può rilevare guasti in più di un disco (cioè, è in grado di rilevare errori che coinvolgono bit memorizzati su più dischi contemporaneamente) ma è considerata una soluzione eccessiva se i singoli dischi sono molto affidabili.



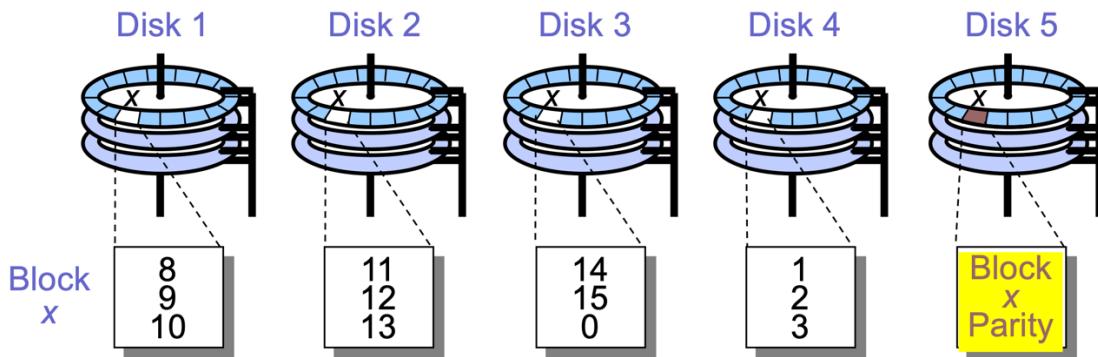
- **RAID 3**

Lo striping avviene a livello di bit (quindi il primo bit sul primo disco, il secondo bit sul secondo disco e così via) e la ridondanza avviene tramite un bit di parità su un apposito disco (ovvero, il bit in quella posizione è 1 se nella posizione omologa su tutti gli altri dischi c'è un numero dispari di 1). Questa configurazione, detta anche a parità bit – interleaved, ha una buona affidabilità (comparabile con RAID 4 e RAID 5) ma ha un basso tempo di servizio, visto che tutti i dischi devono essere acceduti in parallelo, e necessita di un numero di dischi pari alla quantità di dati logici più uno ($N+1$).



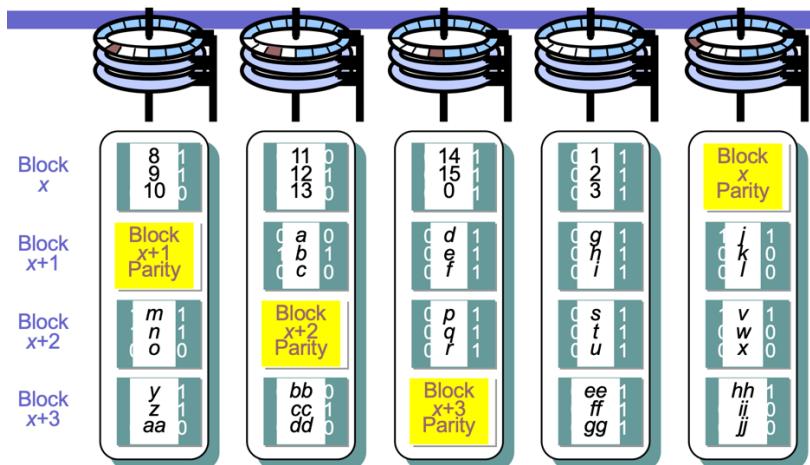
- **RAID 4**

Lo striping avviene su blocchi di dati e la ridondanza con bit di parità per blocco. Questa configurazione, detta anche a parità a livello di blocco, ha una buona affidabilità (comparabile con RAID 2, RAID 3 e RAID 5) e un throughput elevato per le letture (simile al RAID 0) grazie al parallelismo, pur penalizzando le scritture (il disco di parità è un bottleneck). È richiesto lo stesso numero di dischi del RAID 3 ($N+1$).



- **RAID 5**

La ridondanza avviene con bit di parità per blocco, distribuiti su più dischi. Questo approccio ha una buona affidabilità (comparabile con RAID 2, RAID 3 e RAID 4) e in lettura la capacità di trasferimento è simile al RAID 0, penalizzando anche qui la scrittura. È richiesto lo stesso numero di dischi del RAID 3 ($N+1$).



- **RAID 6**

La ridondanza avviene con doppio bit di parità per blocco, con due schemi differenti P e Q e distribuiti su più dischi. Tra le soluzioni presentate, questa è quella che ha il miglior livello di affidabilità. In lettura, la capacità di trasferimento è simile al RAID 0, mentre la scrittura è penalizzata (è il peggioro in questo ambito). È richiesto un disco in più rispetto a quelli del RAID 3 ($N+2$).

Quando si utilizza un **bit di parità**, è necessario effettuare quattro operazioni di I/O per ogni scrittura (detto problema del **read – modify – write**):

1. Lettura della **strip dati originaria**;
2. Lettura della **strip di parità**;
3. Scrittura della **nuova strip dati**;
4. Scrittura della **nuova strip di parità**.

$$\text{Nuova Parità} = (\text{Vecchia Parità}) \oplus (\text{Vecchio Dato}) \oplus (\text{Nuovo Dato})$$

Ciò **rallenta il throughput** rispetto a sistemi senza parità (come il RAID 5 o il RAID 6) o con striping assoluto (come il RAID 0).

Riassumendo:

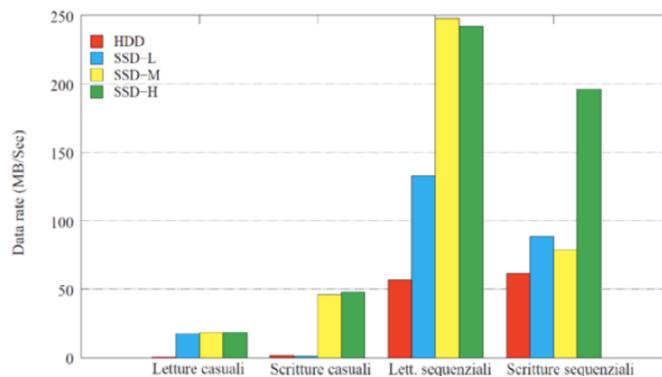
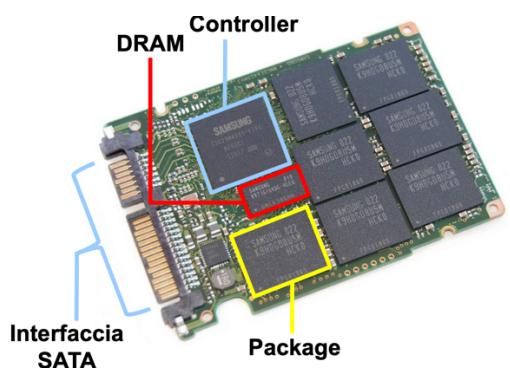
Table 11.4 RAID Levels

Category	Level	Description	Disk required	Data availability	Large I/O data transfer capacity	Small I/O request rate
Striping	0	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

N = number of data disks; m proportional to $\log N$

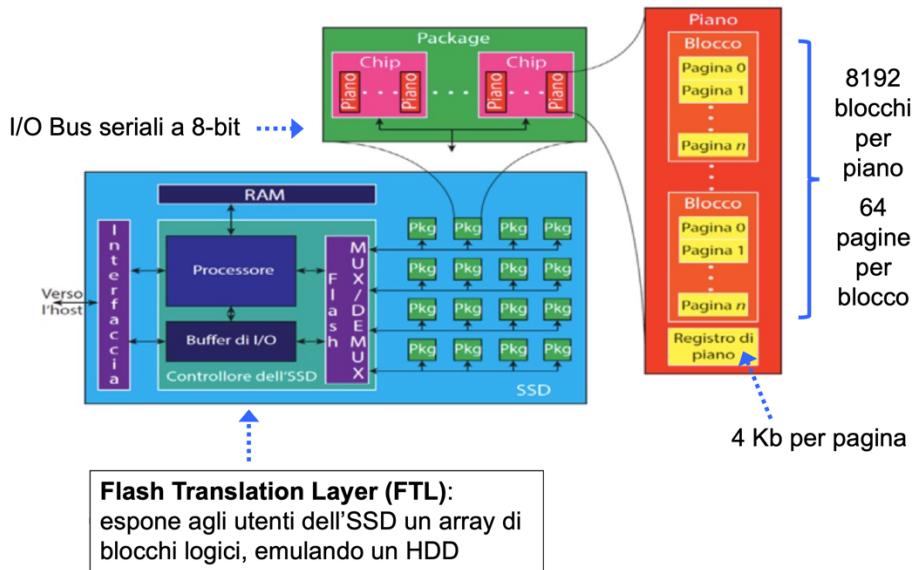
DISCHI A STATO SOLIDO

I dischi a stato solido, o SSD, sono **dispositivi di memorizzazione non volatili**, basati su **tecnologia Flash** e composti di **celle di transistor ad accesso random e riprogrammabili** (quindi hanno poco a che vedere con i dischi nel senso materiale del termine).



L'impiego sempre più frequente degli SSD rispetto agli HDD (Hard Disk Drive) è dovuto ad un **miglioramento prestazionale**, di **robustezza fisica** (gli HDD sono fragili e ingombranti a causa del disco stesso), di **risparmio energetico**, di **costo** e di **usura**; l'unico **parametro che farebbe preferire un HDD ad un SSD è la quantità di informazioni massima che vi si può immagazzinare all'interno**, anche se al giorno d'oggi il miglioramento del processo produttivo di un SSD ha mitigato questo problema.

Entrando nel dettaglio, l'**architettura di un SSD è semplificata dallo schema seguente:**



I Sistemi Operativi sono progettati in base alle seguenti ipotesi implicite sul comportamento degli HDD:

1. Gli accessi sequenziali sono molto più performanti, sia per le letture che per le scritture;
2. Il tempo per accedere ad un settore aumenta con la distanza dall'ultimo settore acceduto;
3. Viene scritto solo il settore che è strettamente richiesto;
4. La vita del supporto non dipende dal numero di operazioni effettuate;
5. L'attività di background sul dispositivo è inesistente.

Con gli SSD alcune di queste ipotesi non sono più valide:

1. Gli accessi sequenziali sono molto più performanti, ma ciò vale più per le scritture che per le letture;
2. Il tempo per accedere ad un settore non aumenta con la distanza dall'ultimo settore acceduto (sono memorie RAM nel senso elettronico del termine);
3. Viene scritto non solo il settore che è strettamente richiesto (write amplification);
4. La vita del supporto dipende dal numero di operazioni effettuate;
5. L'attività di background sul dispositivo non è trascurabile.

Per **write amplification** si intende quella proprietà per la quale anche se si modifica una singola pagina, le scritture avvengono su interi blocchi:

Lettura di un intero blocco di pagine in un buffer → modifica sul buffer
→ sovrascrittura dell'interoblocco

Gli **accessi casuali** (random, da cui RAM) tipici di un SSD tendono a peggiorare le prestazioni e causare usura; quando si scrive sequenzialmente su tutto il blocco, invece, si evita questo overhead accumulando molte scritture vicine e scrivendole tutte insieme.

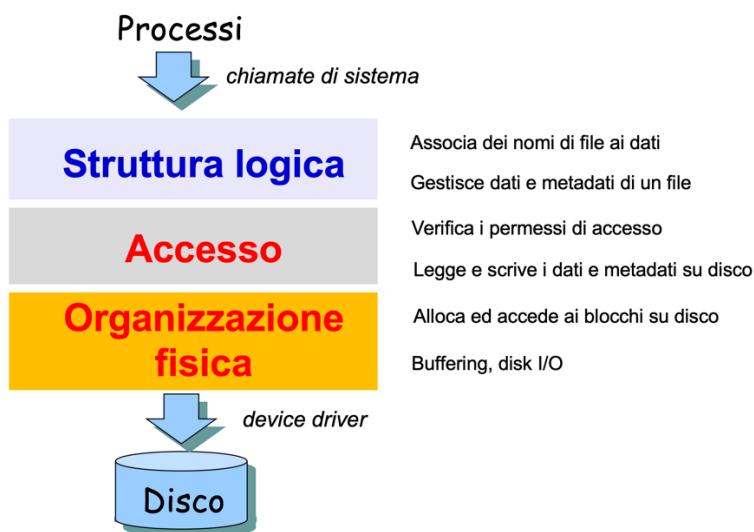
Un **vantaggio** che introduce notevoli miglioramenti prestazionali è dovuto alla proprietà degli SSD per la quale l'accesso è diretto; infatti, viene a mancare la necessità di ordinare gli accessi in base alla loro posizione, potendo tranquillamente utilizzare uno scheduler FIFO senza preoccuparsi troppo di starvation o prestazioni. Il kernel Linux fornisce uno scheduler (opzionale) di tipo FIFO denominato **noop** (no – operation). Può essere usato anche il **Linus Elevator**, favorendo la sequenzialità degli accessi mentre l'anticipazione protegge da eventuali starvation i processi che

effettuano le letture. **Poiché gli SSD sono più veloci**, occorre **ridurre/disabilitare i tempi di attesa** per **non penalizzare gli altri processi**.

Gli **HDD ibridi** (ad esempio, Apple Fusion Drive) includono una **SSD come cache invisibile al Sistema Operativo**, ma è possibile anche usare un **SSD indipendente come cache affiancata ad un HDD** (ad esempio, Windows ReadyBoost o Linux Bcache), permettendo un **risparmio di RAM** e la persistenza dei dati frequentemente acceduti.

IL FILESYSTEM

Il **filesystem (FS)** è l'**organo responsabile della gestione e dell'organizzazione dei file su storage di un sistema** ed è un fattore importante per le prestazioni e la sicurezza del sistema stesso. Un FS si pone di **garantire la persistenza, il recupero, uno schema di naming, la condivisione e la protezione dei file al suo interno**, fornendo alle applicazioni **un'interfaccia unica per la gestione dei file**. Strutturalmente, un FS si compone come segue:



Si vogliono analizzare **singolarmente le componenti di un FS**:

- **Struttura logica**

Un **file** è la **prima astrazione realizzata dal FS** ed inteso come un **insieme di informazioni raggruppate e salvate in una memoria secondaria a cui è assegnato un identificativo simbolico (il nome) ed i seguenti metadati** (o attributi):

- Tipo;
- Lunghezza;
- Data di ultima modifica;
- Proprietario;
- Posizione fisica;
- ...

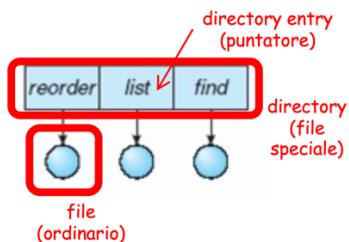
Dal punto di vista del Sistema Operativo, **un file è un vettore di byte**:

0x63
0x69
0x61
0x6F
0x0A

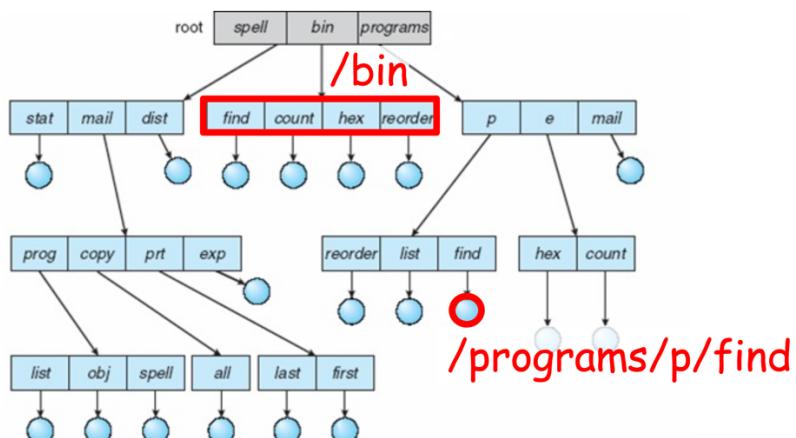
≈ C I A O \n

La **directory** (o cartella) è la **seconda principale astrazione dei FS**, fatta per “raggruppare” più file insieme e sono trattate dal Sistema Operativo come file “speciali”, che non contengono dati ma puntatori ai file (le **Directory Entry**, dei collegamenti logici che mettono in relazione il nome di un file al contenuto fisico sul disco):

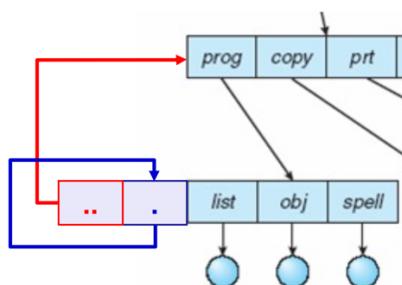
DIRECTORY ENTRY = NOME FILE + RIFERIMENTO DEL FILE SU DISCO



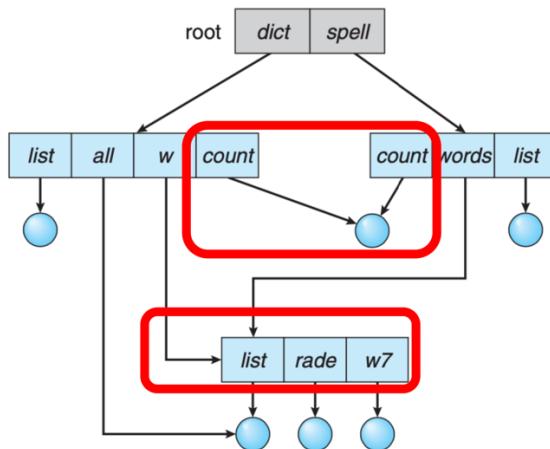
Nei Sistemi Operativi, spesso, si preferisce adottare una struttura delle directory ad albero per motivi di praticità e semplicità di gestione:



In Linux/UNIX, ogni cartella include automaticamente due Directory Entries, . e .. per, rispettivamente, linkare a sé stessa e alla cartella – padre:

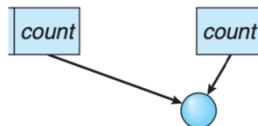


Molti sistemi permettono a più directory di condividere lo stesso file o sotto – directory mediante un meccanismo di **linking multiplo**:

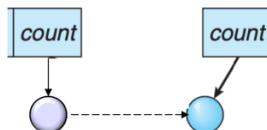


Questo meccanismo può essere **implementato in due modi**:

- **Hard linking**, le entry nella directory hanno **entrambe un puntatore allo stesso settore su disco**, garantendo un accesso **più veloce** (anche se dipende più dalla struttura fisica);



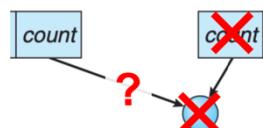
- **Soft linking** (usato in Windows), è **un file fittizio distinto da quello condiviso** (ad esempio, contiene il percorso del file condiviso) e **non dipende dalla posizione dei dati sul disco fisico**, permettendo di **collegare file su FS di tipo differente**.



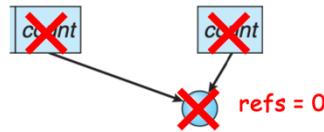
Sebbene in Windows sia possibile solo il meccanismo di soft linking, in Linux/UNIX con i seguenti comandi può essere usato, rispettivamente, sia il hard linking che il soft linking:

```
ln <file_iniziale> <file_nuovo_link>
ln -s <file_iniziale> <file_nuovo_link>
```

Per **rimuovere un file**, occorre **rimuovere la sua entry dalla directory che lo contiene**; nel caso di link, invece, occorre prestare attenzione a quando rimuovere il file dal disco, essendoci il **rischio di avere link ad un file non più esistente su disco**:



Un file è conservato su disco fin quando esiste almeno un riferimento che punta ad esso (ad esempio, in Linux/UNIX, la system call `unlink()` permette di rimuovere un link); pertanto, è richiesto **per ogni file un reference counting** che tiene in considerazione il numero di riferimenti al file stesso. Nell'esempio precedente, il file non verrebbe ancora rimosso dal disco perché ci sarebbe un riferimento attivo; se anch'esso fosse rimosso:



Il reference counting restituirebbe zero e il file sarebbe rimosso dal disco. Quello del reference counting **non è l'unico approccio possibile**, con l'approccio a backpointers si rimuovono tutti i link che puntano ad un file sul disco quando anche solo uno di essi è cancellato (ovviamente, richiede un doppio collegamento, backpointer, tra directory e file), rimuovendo il file dal disco anche eliminando un solo link.

UNIX distingue **sei tipi di file**:

- Ordinary (file dati, sequenza di byte);
- Directory;
- Hard link;
- Symbolic link;
- Special (astrazione dei dispositivi di I/O);
- Named Pipe (meccanismo di IPC via file).

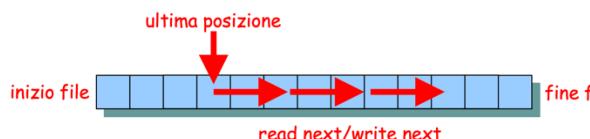
• Metodi di accesso

Su un file **possono essere fatte le seguenti operazioni**:

- Creazione;
- Lettura;
- Cancellazione;
- Apertura (open);
- Scrittura;
- Riposizionamento nel file (seek);
- Chiusura (close).

Molti FS forniscono un'interfaccia conforme allo standard POSIX. **L'accesso ad un file può essere di due tipi**:

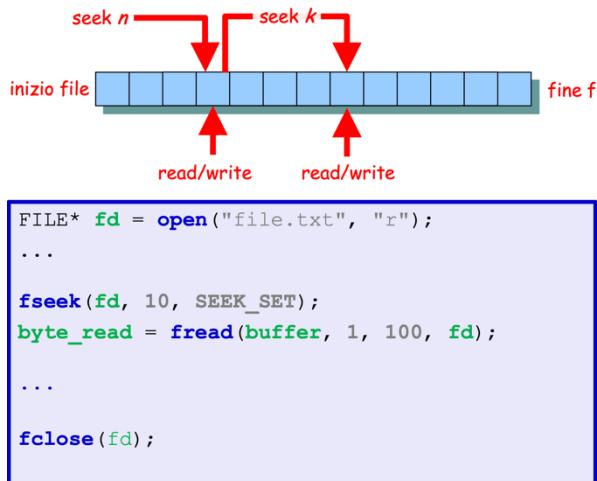
- **Accesso sequenziale**, per ogni file aperto il FS ha un puntatore all'ultima posizione acceduta il quale, dopo un'operazione di lettura/scrittura, avanza al byte successivo;



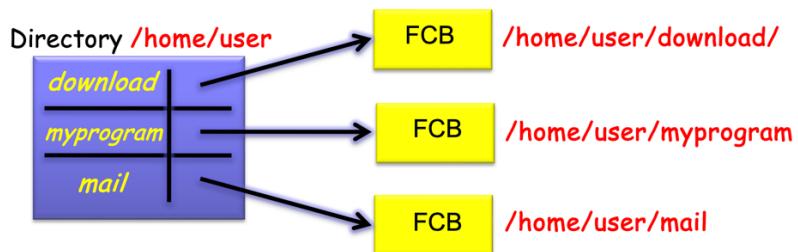
```
int fd = open("file.txt", O_RDONLY);
...
do {
    byte_read = read(fd, buffer, 100);
    ...
} while(byte_read > 0);

close(fd);
```

- **Accesso diretto**, il processo seleziona (seek) una posizione arbitraria (random) da accedere, che sarà letta/scritta alla prossima operazione.



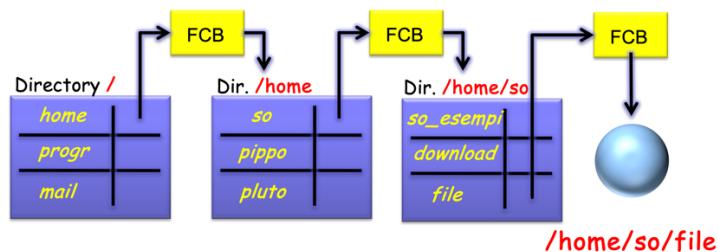
Per ogni file, la Directory Entry punta ad un File Control Block (FCB), una struttura dati con le informazioni sul file:



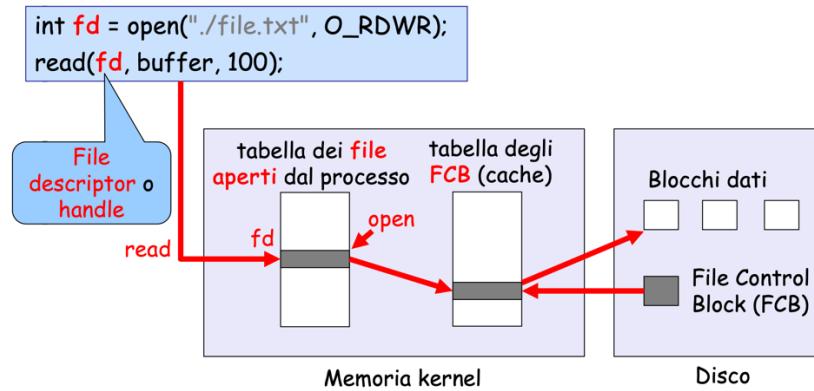
Un tipico FCB (o descrittore del file) contiene:

- Permessi del file;
- Date del file (creazione, ultimo accesso e ultima modifica);
- Proprietario, gruppo e ACL (Access Control List) del file;
- Dimensioni del file;
- Data blocks del file.

Si noti come **il nome del file non sia un attributo del FCB**, visto che è **conservato nella Directory Entry**. Per accedere ad un percorso (path), il FS deve percorrere tutte le directory e i FCB fino ad arrivare ai blocchi su disco del file; **ognuno di questi è, a sua volta, un accesso al disco**, rendendo necessaria la **conservazione dei FCB più usati in RAM (caching)**.

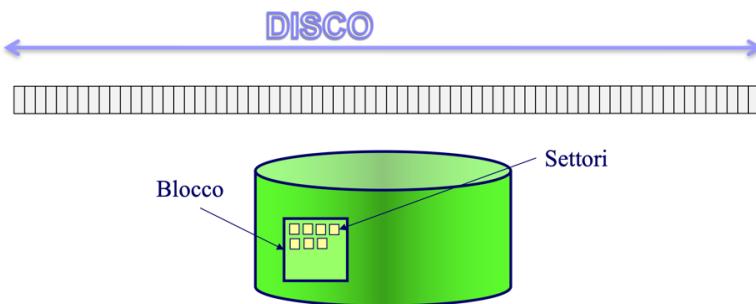


Ad esempio, per la lettura di un file:

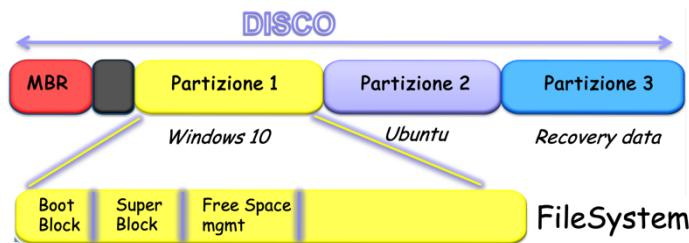


- **Organizzazione fisica**

Dal punto di vista del Sistema Operativo, **il disco è un vettore lineare di blocchi**:



Si parla di blocchi piuttosto che di settori perché questi sono l'unità atomica di gestione del FS ed in genere includono più di un settore (la dimensione di un blocco è tipicamente uguale a quella di una pagina in memoria); pertanto, è ragionevole pensare che **ogni blocco contenga un insieme di settori contigui**. Tutto ciò è fatto per **migliorare l'efficienza di trasferimento dei dati** (evitando i tempi di seek inutili per il principio di località).

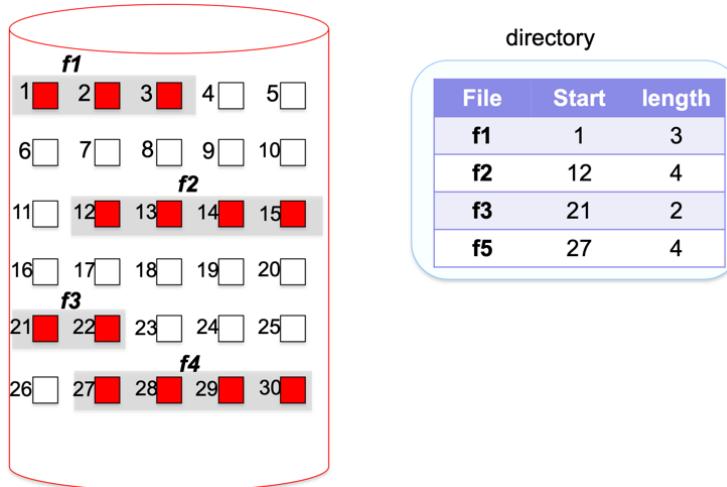


Un disco può essere suddiviso in partizioni, ognuna delle quali dotata di **proprio filesystem**. Per tenere **traccia delle posizioni iniziale e finale** su disco di ogni partizione e del loro numero è inserita automaticamente una **tabella delle partizioni** (la partizione nera in figura); inoltre, è inserita automaticamente anche una **partizione denominata MBR** (Master Boot Record), un settore speciale che **contiene un programma di avviamento** (il bootloader) che si fa da **intermedio tra il BIOS e il kernel**:

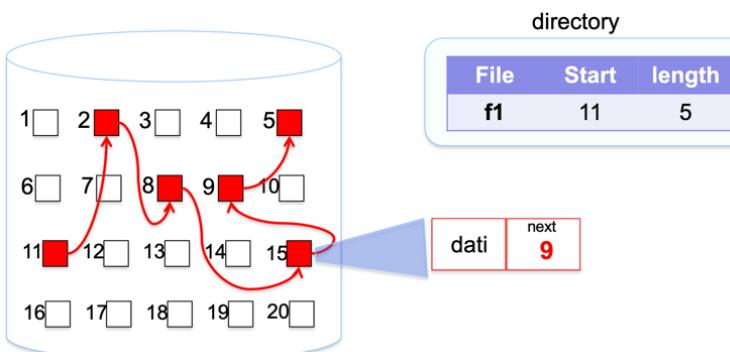
1. L'utente avvia il sistema, invocando il BIOS;
2. Il BIOS preleva ed avvia il bootloader;
3. Il bootloader carica il kernel da una partizione e lo avvia.

I metodi di allocazione definiscono il modo in cui i blocchi sono assegnati ai file e sono tre:

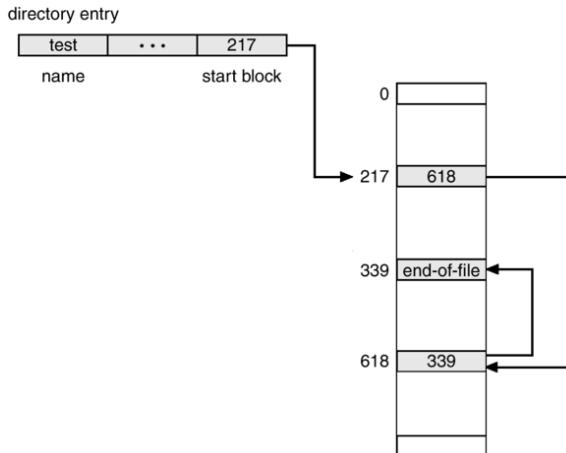
- **Allocazione contigua**, prevede che ciascun file occupi un insieme di blocchi contigui sul disco:
 - **Vantaggi**, favorisce sia l'accesso sequenziale che diretto e, per accedere al file, è sufficiente conoscere solo la sua posizione di inizio (blocco x) e la sua lunghezza (numero di blocchi);
 - **Svantaggi**, produce frammentazione esterna (come nell'allocazione della memoria e introduce il problema della compattazione del disco) e non permette una facile crescita dei file;



- **Allocazione concatenata** (o linked), prevede che ciascun file sia una lista concatenata di blocchi su disco, potendo così essere disposti ovunque:
 - **Vantaggi**, non c'è frammentazione esterna, c'è una maggiore facilità nella scelta dei blocchi da allocare, c'è una maggiore flessibilità (sulla crescita dei file, ad esempio) ed è favorito l'accesso sequenziale;
 - **Svantaggi**, c'è la possibilità di errore se un link viene danneggiato, c'è un maggior consumo di spazio ed è sfavorito l'accesso diretto (casuale);

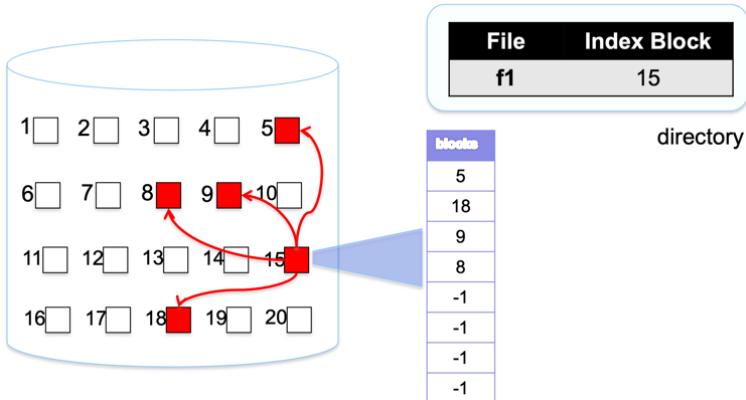


Un esempio comune di FS ad **allocazione concatenata** è **FAT** (File Allocation Table), introdotto in MS – DOS e usato in Windows fino a ME, oggi è una **lingua franca per dischi acceduti da più Sistemi Operativi** (come dischi esterni USB). La FAT **concentra i puntatori in un'unica area sul disco**, invece che disperderli nei blocchi, garantendo **migliore affidabilità** (la tabella può essere replicata) e una **maggior velocità di accesso diretto** (si evita il seek tra blocchi diversi).

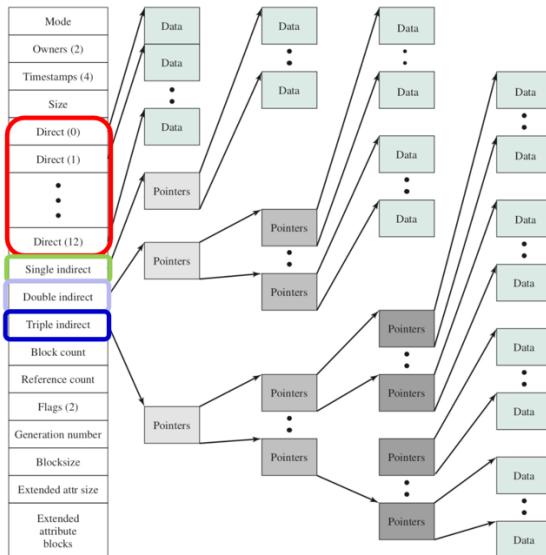


FAT table

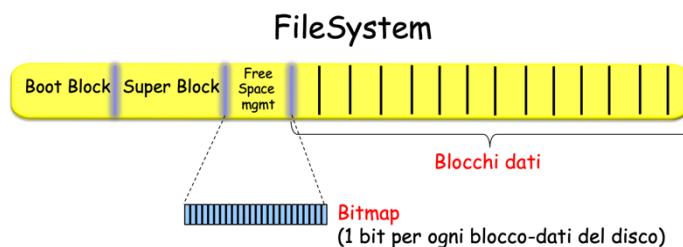
- **Allocazione indicizzata**, prevede che tutti i puntatori siano raggruppati in un unico blocco indice (detto index node):
 - **Vantaggi**, non c'è frammentazione esterna, c'è maggiore facilità nella scelta dei blocchi da allocare, c'è una maggiore flessibilità (sulla crescita dei file, ad esempio) ed è favorito l'accesso diretto (casuale);
 - **Svantaggi**, c'è un maggior consumo di spazio per i blocchi indice e si introduce il problema della dimensione del blocco indice (un indice piccolo evita lo spreco all'interno dell'indice, utile per file piccoli, mentre un indice grande può indicizzare file di maggiori dimensioni, molti blocchi, ma spesso si usano schemi basati su blocchi indice concatenati e/o a più livelli);



In UNIX, **ogni file ha una struttura di controllo (FCB) denominata inode** (index node) dotata di **identificativo univoco** (inode number) e capaci di avere **più riferimenti** (per gli hard link); inoltre, sono adottati **inodes con puntatori sia diretti che indiretti** (sono puntatori multilivello) e, **se un file è piccolo, è sufficiente che il FS utilizzi solo i puntatori diretti**.



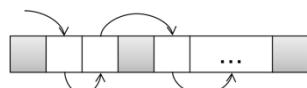
Per gestire lo spazio libero su un disco è usato un blocco apposito, il **FSM** (Free Space Manager) che contiene un bitmap, una serie di bit per ogni blocco dati del disco mappati in modo che riferiscano ad un blocco dati del disco (1 se il blocco è occupato e 0 se è libero):



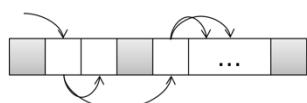
Il bitmap consuma a sua volta dello spazio sul disco, in proporzione al numero totale di blocchi; ad esempio, per un block size di 2^{12} byte (4kB) e un disk size di 2^{30} byte (1GB), ci sono $2^{30}/2^{12} = 2^{18}$ blocchi e un bitmap di altrettanti bit (32kB), ovvero il 0.00305%.

Quello proposto non è l'unico approccio alla gestione dello spazio libero, ce ne sono altri come:

- **Lista concatenata** (free list), per la quale ogni blocco libero contiene un puntatore al successivo blocco libero;



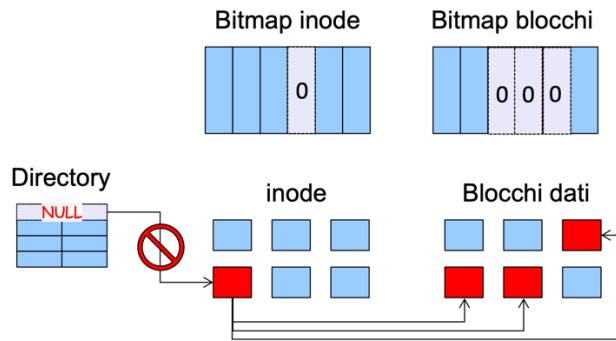
- **Raggruppamento**, variante della free list per la quale ogni blocco libero ha un puntatore ad n blocchi liberi, l'ultimo dei quali indirizza altri blocchi, e così via;



- **Conteggio**, per il quale per ogni gruppo di blocchi liberi, si memorizza in una lista l'indirizzo del primo blocco libero e il numero di altri blocchi liberi (contigui) da cui è seguito.

Inizio	N
2	2
5	1
...	...

Per cancellare un file velocemente, il FS si limita a cancellare solo i metadati del file e a marcare come disponibili i blocchi dati relativi. Essi non sono più raggiungibili ma contengono ancora i dati del file cancellato, almeno finché non vengono sovrascritti da altri dati; in poche parole, è come se si producesse un volontario **memory leak** che non permette di accedere alla sezione di memoria che contiene quei dati.

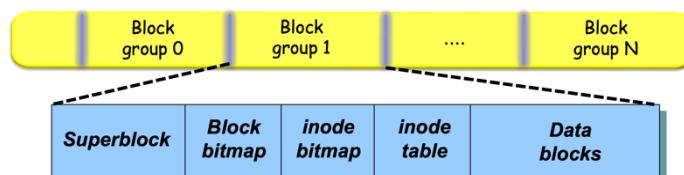


FILESYSTEM IN LINUX E IN WINDOWS

Il **second extended filesystem** (ext2) è un FS che supporta tutte le caratteristiche di un FS standard UNIX ma con diverse estensioni:

- Dimensione dei blocchi configurabile in fase di installazione;
- Attributi dei file ereditabili dalle directory;
- Link simbolici veloci (memorizzati nell'inode, non in un blocco dati);
- Supporto a file **read – only** e **append – only**;
- Gestione dei nomi di file lunghi (256 caratteri, estensibili fino a 1012) con una dimensione massima del file di 4TB.

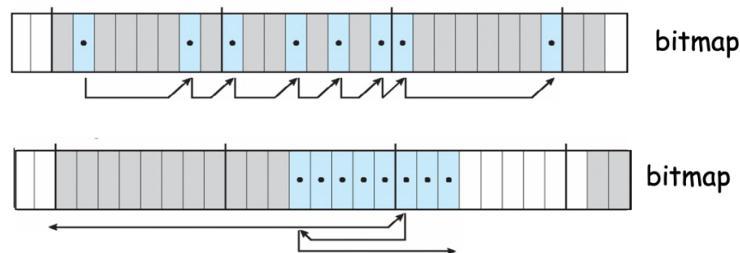
Una partizione ext2 divide il disco in gruppi di blocchi, ognuno dei quali include (approssimativamente) blocchi di una stessa traccia o tracce vicine del disco, in modo da favorire la località degli accessi:



I blocchi – dati e lo inode di uno stesso file sono allocati preferibilmente nello stesso gruppo, mentre gli inode stessi sono preferibilmente allocati nello stesso gruppo della directory che contiene il file, queste ultime distribuite uniformemente tra i gruppi. Ogni gruppo di blocchi include:

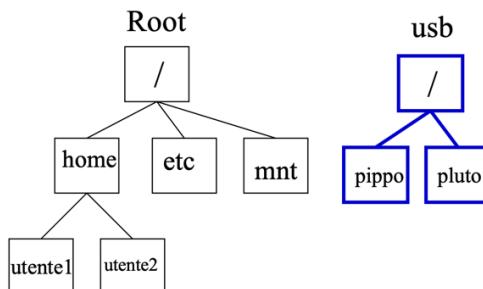
- **Superblock**, contiene le informazioni sulla partizione (ripetute in ogni gruppo), come il numero totale di blocchi e di inode, la dimensione del block group, ecc...;
- **Inode table**, contiene gli inode dei file nel gruppo;
- **Data block**, contiene i blocchi che memorizzano i dati;
- **Block bitmap**, contiene il bitmap per la gestione dello spazio dati libero;
- **Inode bitmap**, contiene il bitmap per la gestione dello spazio in inode table.

L'allocazione dello spazio libero ai nuovi file avviene, come si è potuto intuire, tramite **bitmap**: sono allocati blocchi vicini al primo blocco, anche se non contigui, essendo ancora nella stessa traccia e, pertanto, accessibili senza fare seek; altrimenti, si cerca un gruppo di blocchi contigui liberi (alias, un byte libero nel bitmap).

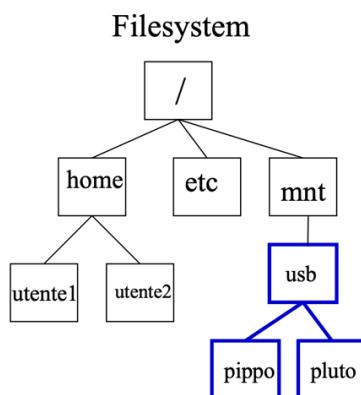


Il comando **df** elenca, per tutte le partizioni attive nel Sistema Operativo, il dispositivo fisico (nel caso di FS per I/O sui dischi), il loro mountpoint (percorso di accesso) e la quantità di spazio disponibile. La variante **df -i**, poi, mostra informazioni sull'utilizzo degli inode del FS (in ext2 e successori, è una quantità massima fissata); si noti che, anche se lo spazio sul disco non è esaurito, è possibile che si riempia la tabella degli inode.

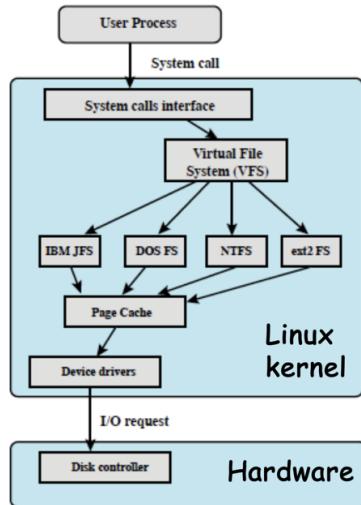
In Linux/UNIX, i FS di tutti i dispositivi sono collegati (mount) ad un unico albero di file e directory:



Il FS integrale:



Con mountpoint /mnt/usb. **Linux è gestito in modo da poter avere collegati diversi filesystem** (ext2, FAT, ecc...), **tutti collegati ad un VFS (Virtual File System)** che riceve le chiamate di sistema e le instrada verso la posizione in cui il file ricercato è conservato, trovando autonomamente il mountpoint e il FS.

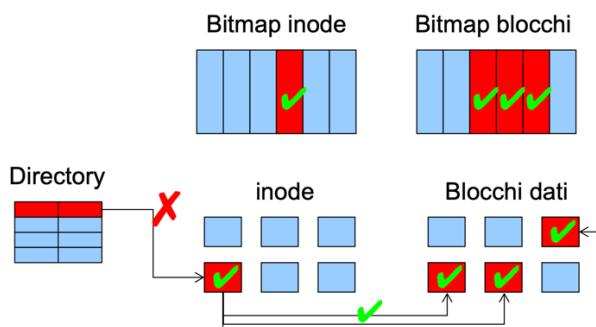


Il comando `mount` fornisce informazioni dettagliate sui FS in uso (se lo si usa, si noti che Ubuntu Linux e molte altre distribuzioni utilizzano ext4, un successore di ext2). **Il comando `sudo cfdisk`, invece, mostra la tabella delle partizioni** (attenzione a non effettuare modifiche sul disco di root della macchina mentre in esecuzione).

Il filesystem **ext3** è stato introdotto nel 2001 nel kernel **Linux 2.4.15** e, rispetto al suo predecessore, introduce:

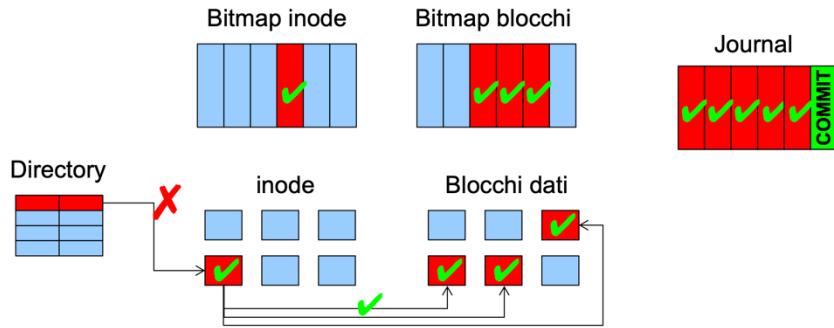
- **Il journaling;**
- **Il ridimensionamento del FS “a caldo”;**
- **L'indicizzazione basata su strutture ad albero** per directory molto grandi.

I **malfunzionamenti** (sia hardware che software) **possono interrompere le operazioni sul FS**, creando inconsistenze tra le varie strutture dati; ad esempio, **se durante la creazione di un file, viene allocato un FCB e dei blocchi ma il sistema va in crash** prima di aggiornare la directory, ciò che ne risulta è:



Il journaling è una pratica per la quale viene gestito un diario (journal) delle modifiche sul disco in memoria transazionale; in questo modo, **un’operazione è fatta “o tutto o niente”**, evitando che un guasto possa lasciare i file in uno stato inconsistente e la necessità di ricorrere a tool di recupero (come e2fsck). **In un journaling FS, le modifiche ai metadati e ai dati vengono prima annotate** (sequenzialmente) **nel journal**, se poi la scrittura ha successo il journal ha un marcatore

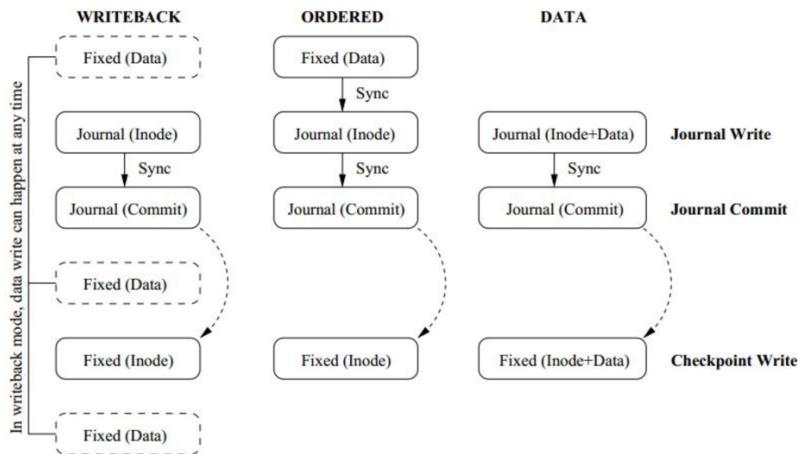
(commit); a questo punto, la scrittura dei dati/metadati nella loro posizione effettiva su disco (checkpointing) può avvenire subito o essere posticipata. Riprendendo l'esempio grafico precedente, al reboot le transazioni committed e pendenti sul journal sono ri – eseguite dall'inizio, mentre quelle non – committed sono annullate e cancellate dal log.



Con questo tipo di journaling, detto **in data mode**, sia i dati che i metadati passano prima per il journal e poi finiscono nella loro posizione effettiva. È una particolare forma di buffering che migliora l'affidabilità a scapito delle prestazioni, spiegando così l'estensivo uso nei Sistemi Operativi commerciali.

Un approccio diverso di journaling prende il nome di **writeback mode** e prevede il **journaling** (così come è stato conosciuto) esclusivamente per i metadati, con la scrittura dei dati che può avvenire in qualsiasi momento; infatti, le scritture ai metadati passano prima per il journal e poi per la loro posizione effettiva, mentre i dati possono essere scritti sia prima che dopo le modifiche ai metadati. Questo secondo approccio migliora le prestazioni ma ne risente in termini di affidabilità.

Una terza modalità di journaling, di default per ext3 e detta **ordered mode**, prevede la scrittura iniziale dei dati nella loro posizione effettiva, senza il passaggio per il journal, per poi salvare i metadati passando, invece, per il journal. Questo approccio offre un buon compromesso tra affidabilità e prestazioni.



Il filesystem **ext4** è stato introdotto nel 2006 nel kernel Linux 2.6.19 e, rispetto al suo predecessore, introduce:

- Il supporto a volumi grandi, fino a 16TB;
- L'utilizzo di gruppi contigui di blocchi, noti come extent, per memorizzare file di grosse dimensioni (massima dimensione di un extent pari a 128MB);

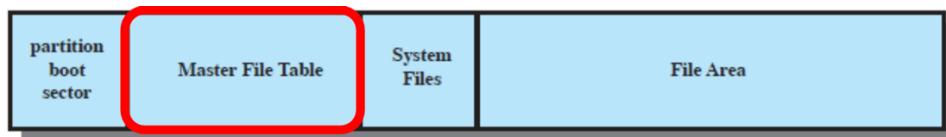
- **L'allocazione multiblocco** (in ext3 la decisione su dove allocare i blocchi di un file era fatta blocco per blocco);
- **L'allocazione ritardata**, per effettuare scelte di allocazione ottimizzate solo quando si decide effettivamente di salvare il file.

Windows, invece, **utilizza un filesystem nominato NTFS**, introdotto negli anni Novanta con Windows NT e le cui principali caratteristiche sono:

- **Recuperabilità**, attraverso un sistema transazionale (basato su journal) per la gestione delle modifiche ed introducendo ridondanza per le strutture dati critiche;
- **Sicurezza**, ad ogni file è associato un descrittore per specificare attributi di sicurezza e ACL secondo il Windows Object Model;
- **Il supporto a file e dischi di grosse dimensioni**;
- **Il supporto a nomi di file lunghi**.

In un FS NTFS, **per MFT (Master File Table) si intende una lista di tutti i file e le cartelle**, organizzata come database relazionale **con righe di 1024 byte contenenti**:

- **Gli attributi del file o cartella**, trattati come metadati;
- **I dati del file**, se esso è abbastanza piccolo da essere contenuto in una singola entry di MFT;
- **I puntatori ai cluster che contengono i dati del file o a dei cluster “indice”** (allocazione indicizzata multilivello).



Un volume, come quello indicato nell'immagine, è **una partizione logica del disco**, mentre un **sector** è **la più piccola unità di storage** (512 byte) e **il cluster uno o più settori contigui**, la cui dimensione va dagli 1 ai 128 settori e dipende dalla dimensione complessiva del volume in maniera proporzionale.

La regione **system files** del volume **contiene**:

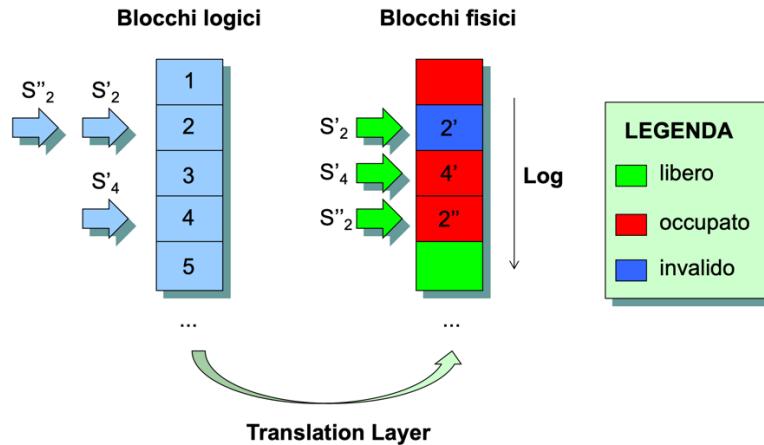
- **MFT – 2**, un mirror delle prime tre righe della MFT, utile per recuperare il suo accesso in caso di guasto di un settore;
- **Log file**, il journal in cui sono memorizzati i passi delle transazioni, per garantire la recuperabilità;
- **Cluster bitmap**, per la gestione dello spazio libero;
- **Attribute definition table**, che definisce gli attributi supportati dal volume (come lo “schema” di un database).

Gli SSD hanno un “contratto non scritto” con il Sistema Operativo, diverso rispetto agli HDD, che prevede:

- **Uniformità dei tempi di lettura**;
- **Maggiore velocità e durata di vita delle celle in caso di scritture sequenziali**.

Con un SSD è utile implementare un filesystem di tipo log – structured (o LFS), visto che **adottano le proprietà appena elencate** (anche se nacquero per ottimizzare l'I/O sequenziale e “write – bound” sui dischi, non per gli SSD). In un LFS, **i blocchi sono gestiti come una coda circolare**

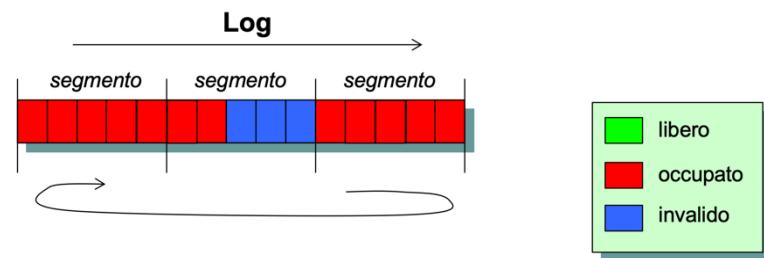
(log) e le scritture avvengono sempre nel blocco fisico successivo nella coda in coda in questione; in questo modo, il log è indipendente dalla posizione “logica” dei dati.



Con questa organizzazione, il Sistema Operativo vede lo SSD come un vettore di blocchi ad accesso casuale. La sovrascrittura di un blocco logico invalida il blocco fisico con la versione precedente, rendendo valida solo l'ultima versione e abilitando le precedenti al riutilizzo. La gestione del riutilizzo può avvenire in due modi:

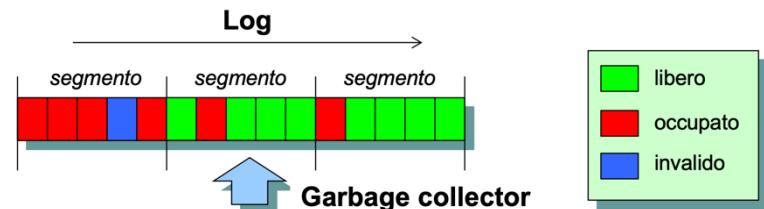
- **Log filettato**

Una volta esauriti i blocchi fisici liberi, il FS riutilizza i blocchi invalidi gestendoli ancora come una coda circolare (saltando i blocchi occupati). Questa soluzione tende a frammentare il log, spezzando la sequenzialità degli accessi.



- **Garbage collector**

Un algoritmo esamina periodicamente i blocchi a gruppi (segmenti), spostando quelli validi da un segmento frammentato ad uno non e andando a reclamare segmenti con soli blocchi invalidi, che vengono poi dichiarati liberi. Questo approccio favorisce le scritture sequenziali su gruppi di blocchi liberi contigui ma genera un’attività di background con un relativo overhead.

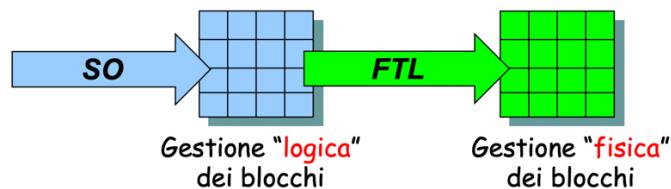


Esistono diverse politiche di garbage collection, tra cui:

- **Collection del segmento in testa al log** (con un elevato overhead nel caso di molti blocchi validi nel gruppo);
- **Collection greedy**, con la quale si libera il segmento con il minor numero di blocchi validi;
- **Collection basata sulla dinamicità dei dati** (caldi, acceduti frequentemente, vs freddi, acceduti meno frequentemente), con la quale i segmenti sono liberati se il numero di blocchi validi è sotto una soglia (i segmenti con dati freddi hanno soglia più alta e sono liberati prima).

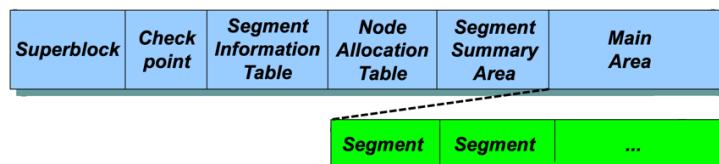
Sia il Sistema Operativo che il controller SSD (FTL, Flash Translation Layer, mappa i blocchi logici gestiti dal FS sui blocchi fisici del dispositivo) possono adottare un log – structured FS secondo diverse configurazioni:

- LFS gestito dal Sistema Operativo (JFFS, YAFFS, ...) e nessun FS nel controller;
- FS tradizionale gestito dal Sistema Operativo (NTFS, FAT, ...) e LFS gestito dal controller;
- LFS gestito dal Sistema Operativo (F2FS) e LFS gestito dal controller.



Quando un blocco logico viene cancellato da un FS tradizionale, è necessario informare la FTL della sua eliminazione, altrimenti il corrispettivo blocco fisico sullo SSD non verrà liberato e il garbage collection non sarà efficace; tutto ciò è effettuato dal comando TRIM, che mette in comunicazione eventuali cancellazioni del FS con la FTL.

Per Flash Friendly File System (F2FS) si intende un filesystem sviluppato da Samsung per il Sistema Operativo Linux ed usato per SSD in dispositivi Android. Assume che il controller SSD utilizzi un LFS e favorisce ulteriormente la sequenzialità delle scritture, allineando le strutture dati alle componenti dello SSD ed ottimizzando la gestione degli indici ai blocchi validi; il tutto garantisce un garbage collection efficace. Il layout di un F2FS è schematizzato come segue:



Dove:

- **Superblock**, contiene informazioni su partizione e parametri di F2FS;
- **Main area**, è divisa a sua volta in 6 log, in base alla dinamicità dei blocchi (per ottimizzare il garbage collection);
- **Segment Information Table (SIT)** e **Segment Summary Area (SSA)**, contengono informazioni sui segmenti, tra cui il bitmap e il numero totale dei blocchi validi nel segmento;
- **Node Allocation Table (NAT)**, contiene gli indirizzi dei blocchi “nodi” (inode, blocchi – puntatore);
- **Checkpoint**, contiene il bitmap per SIT e NAT e informazioni sullo stato del FS.

F2FS utilizza più log separati, distinguendo tra nodi (ossia, inode e puntatori di primo e secondo livello) e dati:

- **Nodi caldi**, blocchi con nodi diretti verso le cartelle;
- **Nodi tiepidi**, blocchi con nodi diretti e non caldi;
- **Nodi freddi**, blocchi con nodi indiretti;
- **Dati caldi**, blocchi dati di tipo cartella;
- **Dati freddi**, blocchi con dati multimediali oppure blocchi che sono stati migrati dall'algoritmo di garbage collection;
- Dati tiepidi, blocchi contenenti dati non classificati né come caldi né come freddi.

Il **garbage collection** è effettuato sia **on – demand** (quando i segmenti liberi sono sotto una soglia, con politica greedy) sia **periodicamente** (basato sulla dinamicità dei dati).

LE MACCHINE VIRTUALI E LA VIRTUALIZZAZIONE DELLA CPU

Una Macchina Virtuale (VM) è un'emulazione, mediante tecniche sia hardware che software, di una macchina reale, eseguendo un proprio Sistema Operativo e le proprie applicazioni come se fosse un sistema a sé stante, ed è gestita da un Virtual Machine Monitor (VMM, o hypervisor); tutte le VM che eseguono su una macchina fisica condividono le sue risorse. Sebbene offrano notevoli vantaggi in termini di prestazioni, affidabilità e sicurezza, le macchine fisiche sono difficili da gestire in termini infrastrutturali ed economici; in quest'ottica, le VM offrono notevole flessibilità, pur rinunciando ai vantaggi offerti dalle macchine fisiche. Ad esempio, si faccia l'esempio di un ambiente di lavoro in cui sono necessari applicativi specifici per Linux, altri specifici per Windows, altri per macOS e altri applicativi legacy; senza virtualizzazione, servirebbero almeno quattro macchine fisiche, rendendo il tutto insostenibile dal punto di vista gestionale.

Le VM possono essere facilmente create, configurate, monitorate e migliorate attraverso strumenti software centralizzati e disponibili sulla macchina fisica, riducendo i costi e semplificando la manutenzione; inoltre, avere più VM su una stessa macchina fisica (pratica denominata workload consolidation) permette di sfruttare appieno le capacità hardware e di ridurre (paradossalmente) i consumi energetici:



Un singolo server

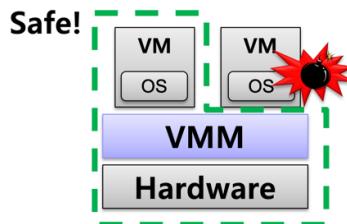
Server consolidati

Per le **applicazioni legacy** la virtualizzazione gioca in casa; infatti, i **Sistemi Operativi obsoleti** su cui girano queste applicazioni possono essere simulati in un ambiente virtuale, portando dietro di sé non solo l'ambiente di esecuzione ma anche la versione specifica del Sistema Operativo, le librerie e i file di configurazione.

Nel caso in cui le **performances ridotte delle VM** fossero un **collo di bottiglia** per il proprio **workflow**, la soluzione può essere offerta dal **cloud computing**, pratica analoga alla virtualizzazione

che permette l'outsourcing delle VM in centri di calcolo privati o di terze parti (pay – per – use); i fornitori di questi servizi hanno **ampie risorse e personale specializzato**, e possono **condividere le risorse tra più clienti** (multi – tenancy) per realizzare economie di scala e gestire efficientemente il carico di lavoro, mentre i clienti sono sollevati dai costi di acquisizione e manutenzione di un centro di calcolo (delega).

Un altro motivo per cui le VM hanno la reputazione che gli è attribuita al giorno d'oggi è la **sicurezza**; infatti, le VM permettono di isolare meglio le applicazioni, in modo da avere una maggiore affidabilità ed una maggiore sicurezza: un'applicazione difettosa/compromessa non vede né la macchina fisica né altre VM e non può compromettere il Sistema Operativo di alcun sistema all'infuori di quello dell'unica VM che vede, quella su cui sta eseguendo (che per l'applicazione è un computer a sé stante, grazie ai principi di encapsulazione ed information hiding). Il VMM è l'unico componente privilegiato che in questo gioco può gestire sia l'hardware fisico che le VM.



Il VMM, come il Sistema Operativo, **fornisce un'astrazione della macchina fisica su cui esegue** secondo i seguenti termini:

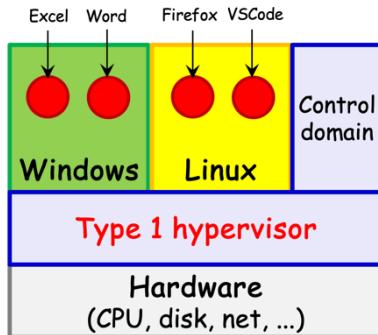
- **Emulazione e scheduling di CPU virtuali;**
- **Allocazione di pagine fisiche alle VM;**
- **Emulazione di dischi ed interfacce di rete.**

L'astrazione di una VMM, in confronto a quella fatta dal Sistema Operativo, rispetto alle risorse fisiche è riassunta nella seguente tabella:

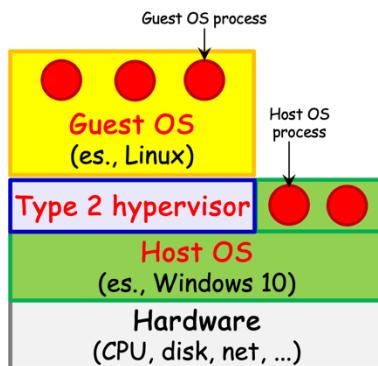
Risorse	Sistema Operativo	VMM
CPU	Processi, Thread	CPU virtuale
Memoria	Memoria virtuale	Memoria virtuale
Disco	File, Directory	Disco virtuale
Rete	Socket	Rete virtuale

Esistono prevalentemente **due architetture di virtualizzazione**:

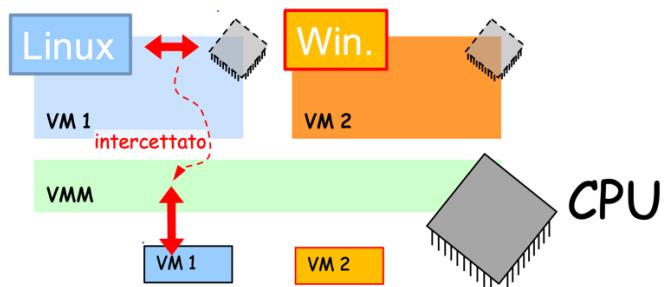
- **Virtualizzazione bare – metal**, per la quale il **VMM** (detto anche hypervisor di tipo 1) esegue sull'hardware "nudo", senza altre architetture che vi si interpongono e offrendo **migliori prestazioni** (è la soluzione più utilizzata in ambito server);



- **Virtualizzazione hosted**, per la quale il VMM (detto anche hypervisor di tipo 2 o hosted hypervisor) esegue su un Sistema Operativo tradizionale (host OS), come Windows, facilitando l'integrazione con i Sistemi Operativi nelle VM (guest OS) ed è la soluzione più usata in ambito desktop.

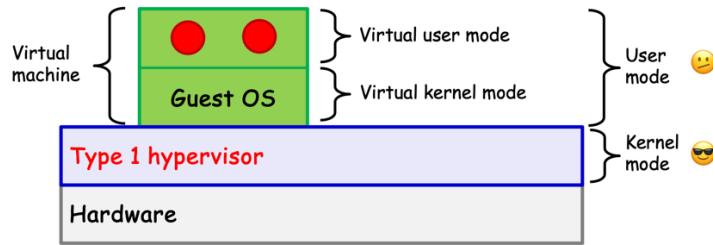


Il Sistema Operativo non è un comune programma, accede in modo esclusivo e privilegiato ad interrupts, eccezioni, MMU, tabelle delle pagine, porte I/O, stati del processore, ecc... e se il file binario del kernel fosse lanciato come un comune programma user – space non riuscirebbe ovviamente ad eseguire; pertanto, il VMM espone al guest OS un'emulazione della CPU, in modo da intercettare l'esecuzione delle istruzioni critiche (sensitive instructions) e leggere/scrivere su una struttura dati tipica per ogni VM. Il tutto, schematizzato come segue:



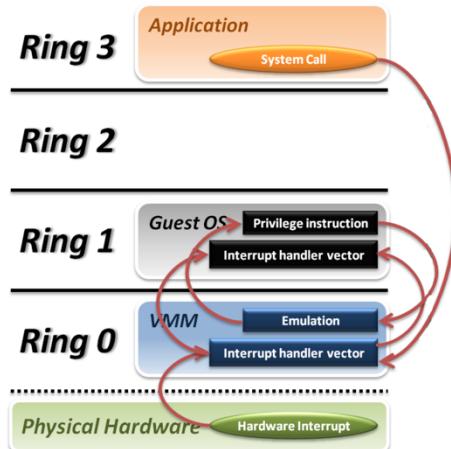
Il VMM e il guest OS coesistono a diversi livelli di privilegio della CPU (pratica detta de – privileging); in particolare:

- Il VMM è in **kernel mode**, ruolo normalmente ricoperto dal Sistema Operativo;
- La VM è in **user mode**, dentro cui:
 - Il guest OS è in **virtual kernel mode**;
 - Le applicazioni utente sono in **user mode**.



Se il guest OS (de – privilegiato) esegue istruzioni privilegiate, la CPU genera una trap, il VMM intercetta tale trap ed emula in software gli effetti dell’istruzione privilegiata; questa pratica, che permette tutte le feature di sicurezza vantate in precedenza, è detta **trap – and – emulate**.

Ad esempio, con quattro livelli di privilegio (Ring):



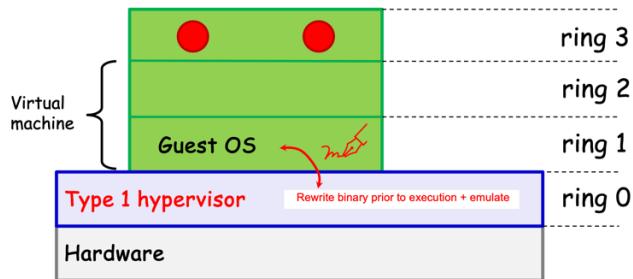
L’architettura x86 tradizionale non è virtualizzabile con solo il meccanismo di trap – and – emulate e molte istruzioni sensibili non generano alcuna trap; in questo caso, se il guest tenta di eseguire un’istruzione sensitive, la CPU ignora l’istruzione, che viene detta **sensibile ma non – privilegiata**. In quest’ottica, si individuano diverse tecniche di virtualizzazione CPU:

- **Full virtualizzazione, senza supporto hardware** e usando tecniche software (dynamic binary translation, shadow page tables, ...);
- **Para – virtualizzazione, il guest SO è sviluppato appositamente** per cooperare con il VMM;
- **Full virtualizzazione, con supporto hardware** per migliori prestazioni e VMM più semplici.

Tra **virtualizzazione Full** e **Para** c’è una differenza sostanziale riassunta dal **confronto tra hypercall e Dynamic Binary Translation (o DBT)**:

- **Hypercall** (para – virtualizzazione)

Nel 1999, il VMM di VMware introdusse **tecniche efficienti di para virtualizzazione** per Intel x86 per le quali **il codice binario del guest OS viene riscritto al volo dal VMM stesso prima di essere eseguito**; in questo modo, **sono sostituite le istruzioni sensibili con codice di emulazione**.



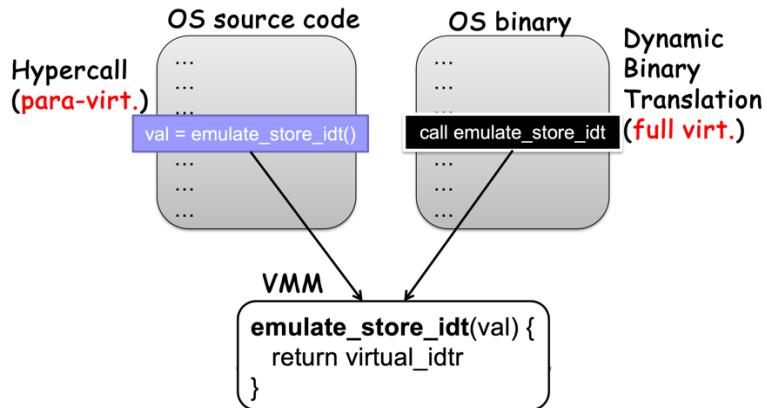
Il guest OS è consapevole di essere eseguito in un ambiente virtuale e per interagire con il VMM utilizza delle chiamate speciali, create appositamente per quel guest OS, dette **hypercall**; le hypercall sono operazioni esplicite per comunicare al VMM e richiedere l'accesso a risorse privilegiate o eseguire operazioni non consentite direttamente al guest (simili alle system call ma eseguite dal guest OS). È un approccio molto efficiente, perché evita di emulare istruzioni complesse e problematiche, ed è adatto per ambienti controllati, dove è possibile modificare il codice del Sistema Operativo; gli svantaggi di questo approccio risiedono proprio nella necessità di dover modificare il codice sorgente del guest OS (la modifica è fatta dal programmatore, non dal VMM), rendendosi inutilizzabile con Sistemi Operativi non modificabili (come Windows standard o versioni legacy) e rendendo il guest OS non eseguibile su hardware fisici (può funzionare solo in combinazione con il VMM per il quale è stato programmato).

- **DBT** (full virtualizzazione)

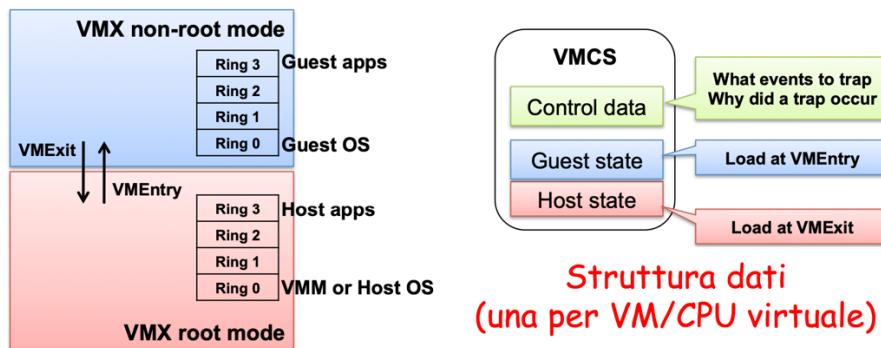
Il codice binario del guest OS, non modificato, viene eseguito direttamente; infatti, quando il guest OS tenta di eseguire un'istruzione non virtualizzabile, il VMM intercetta l'istruzione e la traduce dinamicamente in un'operazione che può essere emulata. Questa traduzione avviene “al volo” e non richiede consapevolezza da parte del guest OS. Il vantaggio di questo approccio consiste nel non dover richiedere modifiche al guest OS (potendo così lavorare con Sistemi Operativi non modificabili), al costo però di un overhead maggiore rispetto all'Hypercall (l'emulazione dinamica è più complessa e meno efficiente).

Sin dall'avvio della VM, il VMM analizza a blocchi il codice eseguito dal guest OS e riscrive le eventuali istruzioni privilegiate con codice di emulazione; inoltre, il salto finale viene sostituito con una chiamata al VMM, in modo che possa avanzare il processo di traduzione al prossimo blocco (per definizione, una breve sequenza di istruzioni sequenziali che termina con un'istruzione di salto).

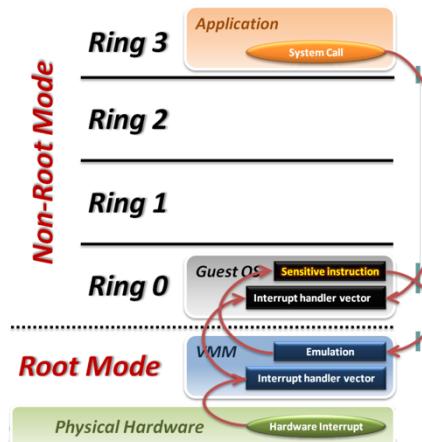
Caratteristica	Hypercall (Paravirtualizzazione)	DBT (Virtualizzazione Completa)
Richiede modifiche al guest OS	Si	No
Efficienza	Alta	Minore (più overhead)
Compatibilità	Limitata (solo OS modificabili)	Totale (funziona con qualsiasi OS)
Implementazione	Utilizza chiamate dirette al VMM	Traduce istruzioni al volo



Per la virtualizzazione, le **CPU Intel VT** introducono due modalità di esecuzione (VMX, o VT – x, root e non – root) e un VMCS (VM Control Source). Questi strumenti semplificano il codice del VMM, eseguendo buona parte della virtualizzazione in hardware, evitando il ring de – privileging del guest OS, dando maggiore efficienza del cambio di contesto tra VM e VMM e garantendo il meccanismo di trap per tutte le istruzioni critiche. La VMX root mode permette l'esecuzione del VMM, mentre la non – root mode quella del guest OS, il cui comportamento delle istruzioni sensibili è configurato dal VMM, e il passaggio tra le due è innescato da eventi detti VM – exit e VM – enter (timer, page fault, ...).



Il ring de – privileging:



La trap, in questo caso, è gestita direttamente dal guest OS, mentre gli eventi hardware sempre dal VMM, anche se con qualche ottimizzazione in più. Non c'è alcun bisogno di riscrittura del guest OS e tutte le istruzioni causano trap, in modo configurabile.

Il VMM usa la VMCS per indicare alla CPU le condizioni e le azioni di VM – entry e VM – exit, in cui è salvato/caricato in hardware anche lo stato della CPU.

VM-execution controls	Determines what operations cause VM exits	CR0, CR3, CR4, Exceptions, IO Ports, Interrupts, Pin Events, etc.
Guest-state area	Saved on VM exits Reloaded on VM entry	EIP, ESP, EFLAGS, IDTR, Segment Regs, Exit info, etc.
Host-state area	Loaded on VM exits	CR3, EIP set to monitor entry point, EFLAGS hardcoded, etc.
VM-exit controls	Determines which state to save, load, how to transition	Example: MSR save -load list
VM-entry controls	Determines which state to load, how to transition	Including injecting events (interrupts, exceptions) on entry

Esempi di cause di VM – exit sono:

- **Istruzioni sensibili:**
 - CPUID, riporta le CPU capabilities;
 - RDMSR/WRMSR, legge/scrive i “model – specific registers”;
 - INVLPG, invalida entry TLB;
 - RDPMC e RDTSC, leggono i registri di performance monitoring e di timestamp;
 - HLT, MWAIT e PAUSE, disattivano il guest OS;
 - VMCALL, nuova istruzione per invocare il VMM;
- **Accessi a stato sensibile:**
 - MOV DRx, accessi ai debug register;
 - MOV CRx, accessi ai control register;
 - Task switch, accessi al CR3 (puntatore alla tabella delle pagine);
- **Eccezioni ed eventi asincroni** (page fault, debug exceptions, interrupts, ...).

Il VMCS consente di iniettare eventi (interrupt, eccezioni, ...) nella VM al momento della VM – entry; tale iniezione è fatta dalla CPU in hardware, semplificando il VMM e migliorando le prestazioni. Ad esempio:

Esempio: Page faults

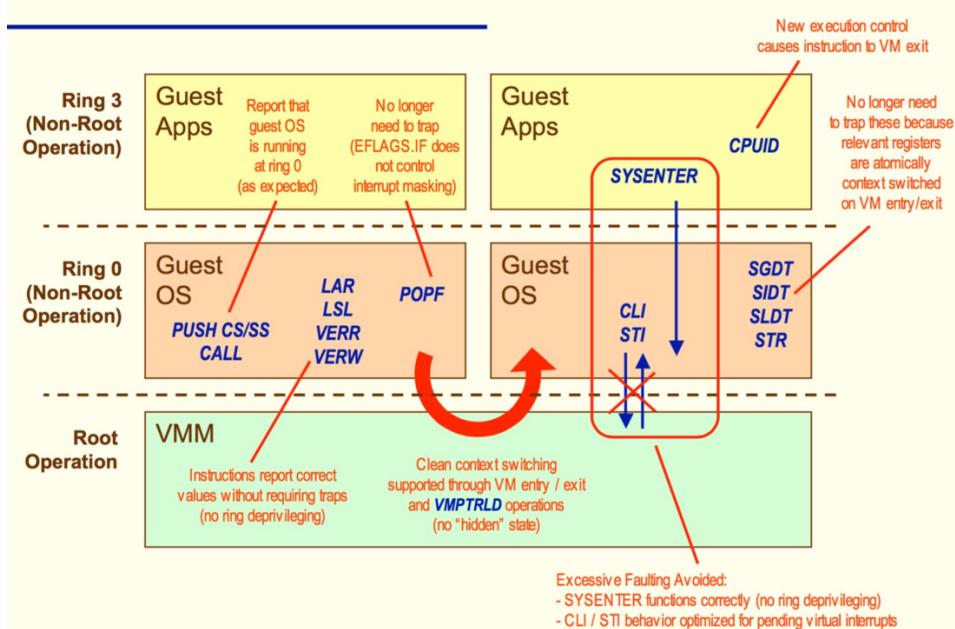
1. La CPU scrive nel VMCS le cause dell'eccezione
2. Il VMM sceglie se **gestire il page fault**, in modo trasparente al guest OS (es., paginazione a domanda), ...
3. ... oppure iniettare l'eccezione nel guest OS, utilizzando il campo VM-entry controls nel VMCS

Esempio: Interrupt masking

1. Il guest OS usa istruzioni CLI/STI (caso frequente)
2. La CPU apre/chiude la “interrupt window” della VM, **senza innescare** il VMM con delle VM-exit
3. Quando si verifica un interrupt, il VMM “accoda” l'interrupt alla VM
4. La CPU inietterà automaticamente l'interrupt alla apertura della interrupt window

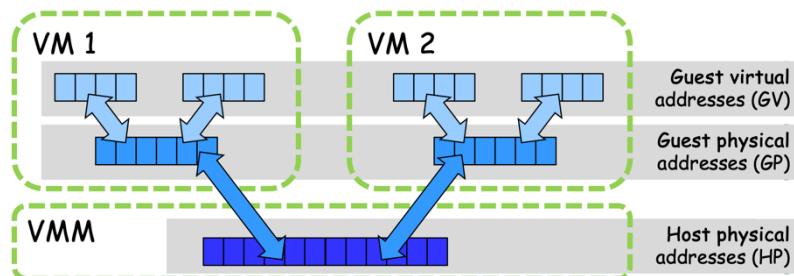
Riassumendo:

How VT-x Closes Virtualization Holes

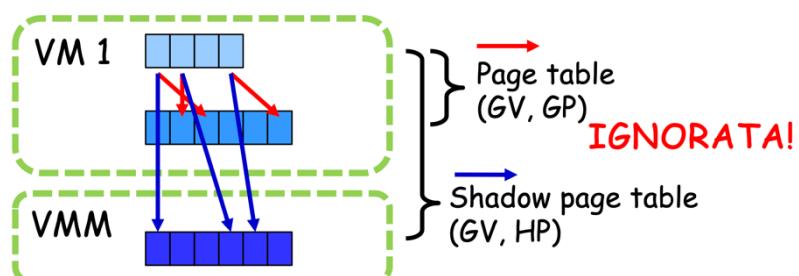


LA VIRTUALIZZAZIONE DELLA MEMORIA E DELL'I/O

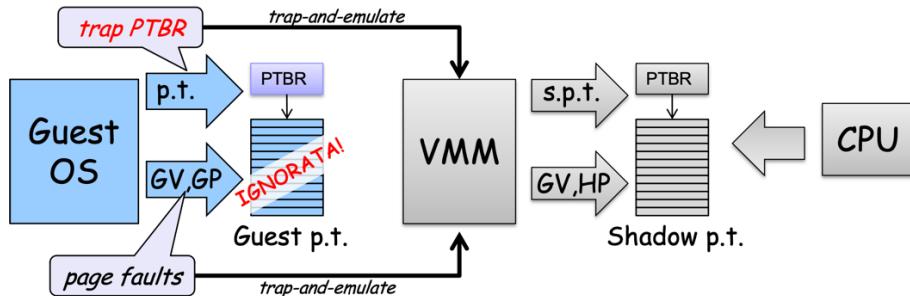
All'interno di una VM, il **guest SO** gestisce una “finta” memoria fisica della VM, mentre il **VMM** amministra quella vera fisica, anch’esso con paginazione a domanda, swapping, algoritmi di prelazione, ecc... Si tenga poi in considerazione che, così come i processi del Sistema Operativo di una VM non erano accessibili alle altre VM, così la memoria di una VM non è accessibile alle altre VM, è isolata. Schematicamente, gli indirizzi virtuali e fisici sono così gestiti:



Per shadow page table si intende una tecnica per la virtualizzazione della memoria per la quale il VMM intercetta le modifiche alle page table nel guest, creando una loro controparte, detta appunto shadow page table. La CPU usa lo shadow page table, ignorando la page table del guest OS:



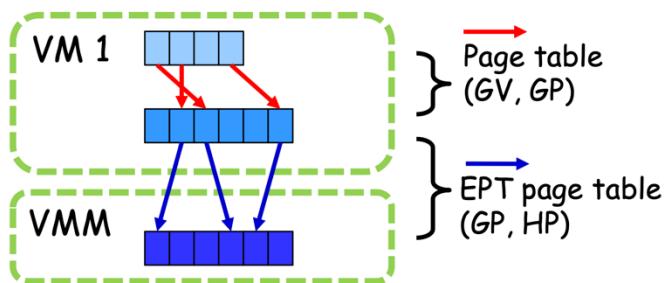
In particolare, il guest OS scrive, con un’istruzione sensitive, nel registro virtuale PTBR l’indirizzo della page table; con trap – and – emulate, poi, il VMM carica nel registro fisico PTBR l’indirizzo della shadow page table ed intercetta le modifiche alla page table (le cui pagine sono read – only, in modo da causare page fault). La CPU usa la shadow page table per tradurre gli indirizzi virtuali della VM, ignorando la guest page table. Schematicamente:



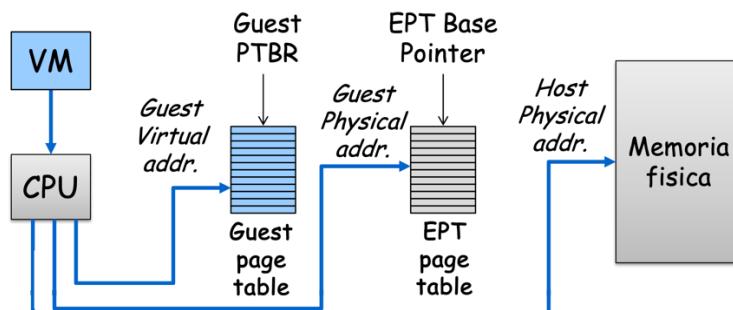
Le shadow page tables sono un meccanismo di virtualizzazione oneroso, visto che aumentano il volume e l’overhead dei page fault. Si distinguono:

- **Hypervisor – induced page fault**, introdotti dall’hypervisor per aggiornare le shadow page table;
- **Guest – induced page faults**, “genuini” e prodotti dalla VM, intercettati poi dal VMM e “re – iniettati” nella VM.

Intel VT introduce un livello aggiuntivo di traduzione degli indirizzi mediante Extended Page Tables (EPT), gestite dal VMM; la MMU accede in hardware ad entrambe le tabelle, evitando hypervisor – induced page faults:

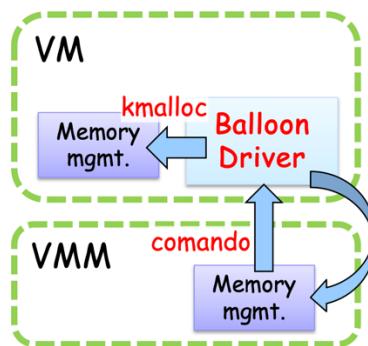


In particolare, la CPU accede le guest page del guest OS (da guest virtual address a guest physical address) e poi accede le extended page table del VMM (da guest physical address a host physical address):

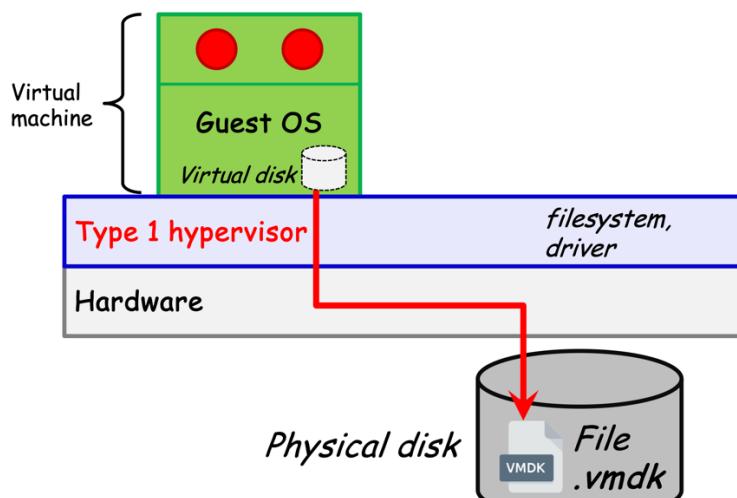


L’algoritmo di sostituzione del VMM può entrare in conflitto con l’algoritmo di sostituzione del guest OS, non conoscendo quali pagine quest’ultimo ritiene “superflue”; la tecnica del **ballooning** permette la cooperazione tra guest OS e VMM:

- Il guest OS esegue un modulo kernel (balloon driver), che riceve comandi dal VMM;
- Quando il VMM ha bisogno di allocare nuove pagine, invia al balloon un comando di espansione;
- Il balloon aloca memoria all’interno della VM, forzando il guest OS a fare swap – out delle pagine superflue;
- Il VMM utilizza le pagine allocate dal balloon senza conflitti.



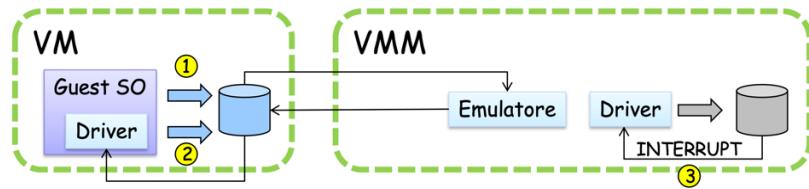
Le operazioni di I/O sono virtualizzabili con l’approccio trap – and – emulate, simulando un dispositivo virtuale; per memory – mapped I/O, si intercettano accessi agli indirizzi del dispositivo tramite page fault, mentre per port – based I/O si intercettano le istruzioni sensibili di I/O. Facendo l’esempio di un disco virtuale, il VMM riceve il comando di lettura/scrittura ed opera su un file dedicato (ad esempio, .VMDK) che conterrà i dati del disco virtuale; invece, per un NIC virtuale, il VMM riceve un pacchetto e lo inoltra alla rete fisica o virtuale (virtual switch).



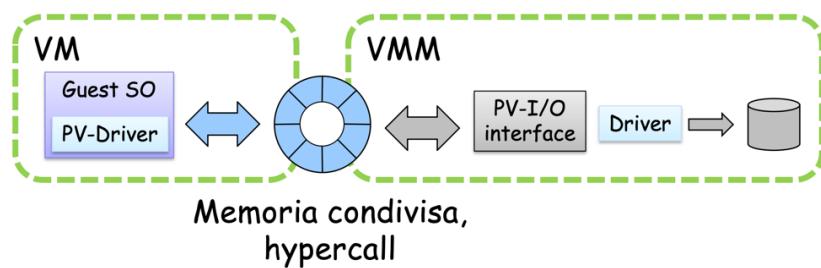
Praticamente, per la virtualizzazione dell’I/O esistono varie tecniche:

- **I/O full virtualization**, simula in software un dispositivo reale (spesso semplici e ben noti) ma le prestazioni sono penalizzate dall’emulazione e dal passaggio attraverso il VMM:
 1. Il guest OS interroga il bus per trovare i dispositivi connessi, il VMM intercetta ed indica il modello di dispositivo virtuale;
 2. Il guest OS tenta di leggere/scrivere i registri del dispositivo virtuale, il VMM intercetta e copia i valori da/verso i registri hardware;

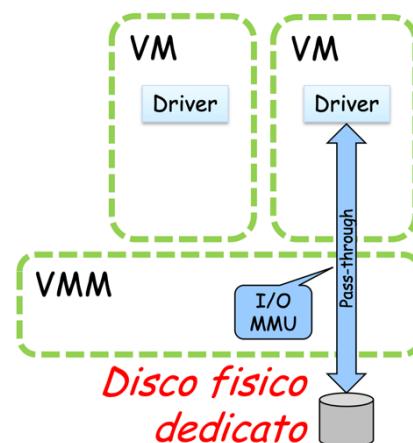
3. Il dispositivo hardware genera un'interruzione, il VMM la serve simulando un'interruzione nel contesto della VM;



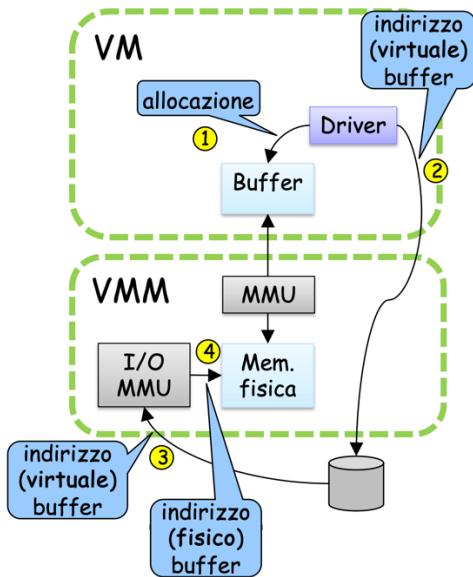
- **I/O para – virtualization**, è l'analogo della para – virtualization della CPU e, pertanto, prevede che **il guest OS dialoghi con il VMM tramite hypercall**, evitando la complessità e il rallentamento dovuti alla simulazione di un dispositivo:
 - Il VMM esporta al guest OS delle aree di memoria condivisa e hypercall per l'I/O, mentre questo è consapevole della virtualizzazione (carica un driver apposito per l'I/O paravirtualizzato);



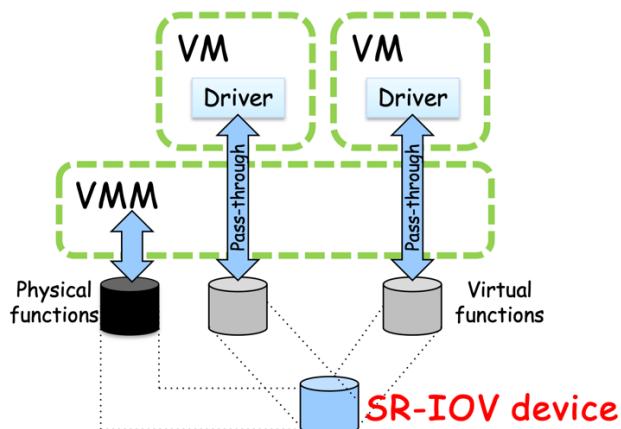
- **I/O passthrough**, assegna un dispositivo di I/O in modo esclusivo ad una VM, senza virtualizzazione, e il guest OS vi accede direttamente tramite memory – mapped I/O;



La I/O – MMU è un componente utilizzato per l'uso sicuro di device in maniera passthrough e in altre forme di virtualizzazione avanzata (senza, i dispositivi possono leggere/scrivere su qualsiasi indirizzo fisico della RAM, con, i dispositivi accedono indirettamente alla memoria, tramite indirizzi di memoria virtuali configurati dal VMM). **Il guest OS effettua operazioni di memory – mapped I/O su indirizzi "guest physical"** (virtuali); **il dispositivo tenta di accedere all'indirizzo di memoria virtuale e la I/O – MMU lo traduce in un indirizzo fisico**, controllato dal VMM, **inoltrando anche le interruzioni del dispositivo al guest OS** (pratica denominata interrupt remapping):



- **SR – IOV (Single Root I/O Virtualization)**, in caso di molte VM non è fattibile assegnare in modo esclusivo un dispositivo ad ogni VM e questa è una funzionalità dei moderni dispositivi PCIe che permette la condivisione dei device fra più VM:
 - Il controller fornisce un insieme di interfacce virtuali (virtual functions), ognuna delle quali ha un insieme separato di registri, DMA e interrupt.

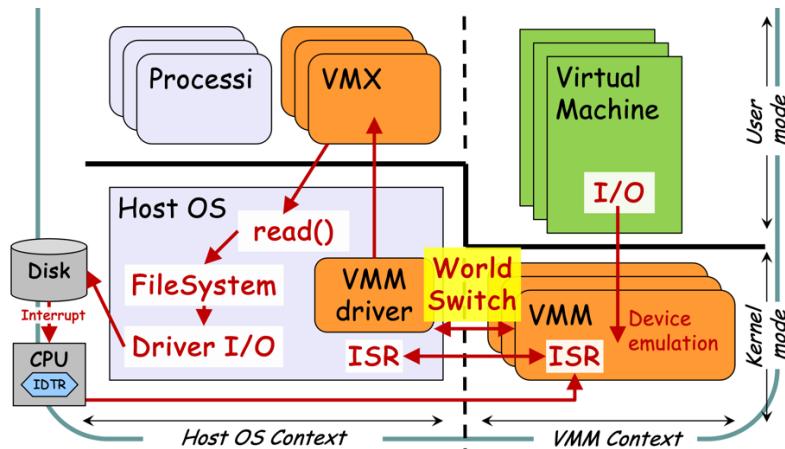


LE TECNOLOGIE VMWARE E CONTAINER – BASED

VMware Workstation è stata la prima soluzione di virtualizzazione per piattaforme Intel x86, con un hypervisor di tipo 2 che esegue come se fosse un'applicazione su host OS Windows o Linux, una full virtualization con dynamic binary translation, full memory virtualization con shadow page tables e full I/O virtualization tramite emulazione di semplici dispositivi legacy. Per convivere con l'host OS, VMware Workstation introduce i seguenti componenti:

- **Processo VMX**, un programma user – space “percepito” dall’utente (un VMX per ogni VM);
- **VMM**, che esegue in modo kernel fuori dal contesto dell’host OS, implementando anche meccanismi di full virtualization (un VMM per ogni VM);
- **VMM driver**, un modulo kernel che esegue nel contesto dell’host OS e che permette la cooperazione con il VMM;

- **World Switch**, un cambio di contesto tra host OS e VMM (tutto lo stato della CPU, inclusi registri di controllo).



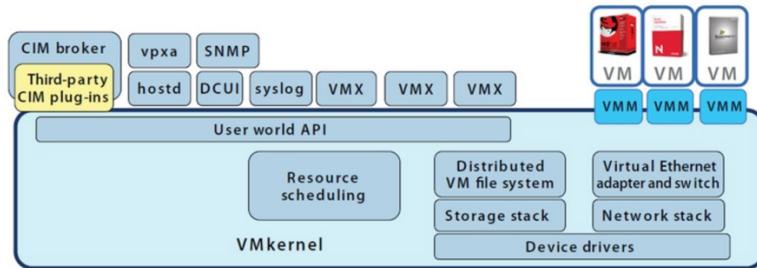
Un esempio di lettura di I/O nella VM:

- Il guest OS nella VM effettua I/O sul dispositivo virtuale;
- Il VMM intercetta l'operazione ed innesca un world switch verso l'host OS;
- Il VMM driver notifica il processo VMX;
- Il processo VMX effettua `read()` sul file .VMDK;
- Il disco genera un'interrupt, intercettata dal VMM;
- Il VMM innesca un altro world switch verso l'host OS;
- Il VMM driver innesca la ISR nell'host OS;
- I dati sono letti dal processo VMX e raggiungono la macchina virtuale, seguendo il percorso opposto.

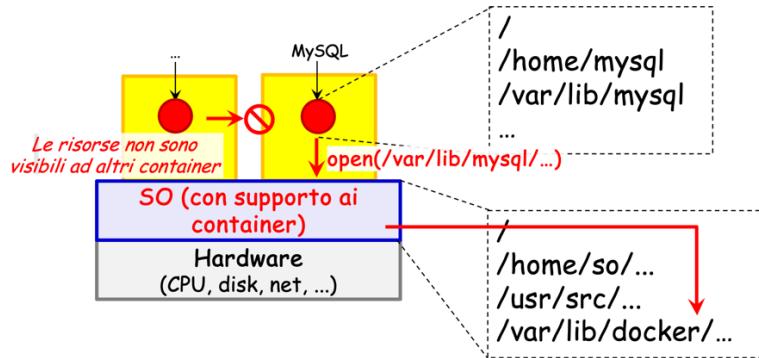
	<i>Virtual Hardware (front end)</i>	<i>Back end</i>
Multiplexed	1 virtual x86 CPU, with the same instruction set extensions as the underlying hardware CPU	Scheduled by the host operating system on either a uniprocessor or multiprocessor host
	Up to 512 MB of contiguous DRAM	Allocated and managed by the host OS (page-by-page)

Emulated	PCI Bus	Fully emulated compliant PCI bus
	4x IDE disks 7x Buslogic SCSI Disks	Virtual disks (stored as files) or direct access to a given raw device
	1x IDE CD-ROM	ISO image or emulated access to the real CD-ROM
	2x 1.44 MB floppy drives	Physical floppy or floppy image
	1x VMware graphics card with VGA and SVGA support	Ran in a window and in full-screen mode. SVGA required VMware SVGA guest driver
	2x serial ports COM1 and COM2	Connect to host serial port or a file
	1x printer (LPT)	Can connect to host LPT port
	1x keyboard (104-key)	Fully emulated; keycode events are generated when they are received by the VMware application
	1x PS-2 mouse	Same as keyboard
	3x AMD Lance Ethernet cards	Bridge mode and host-only modes
	1x Soundblaster	Fully emulated

VMware ESXi è un hypervisor bare metal, che riusa il VMM da Workstation ma l'host OS è sostituito dal VMkernel. CPU e memory management sono ottimizzati per scalabilità e basso overhead, con funzioni avanzate di migrazione, network/storage virtualization, ...

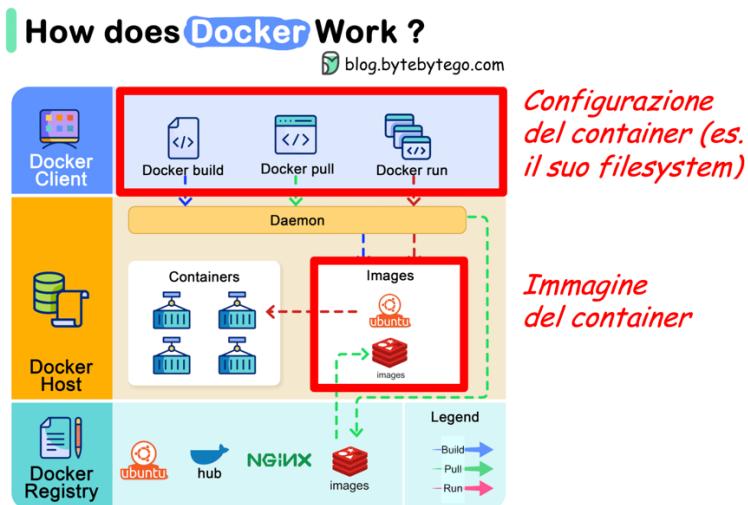


Per container si intende un'estensione del concetto di processo, ovvero di un'entità dotata di un proprio filesystem namespace, propri indirizzi IP, propri processi figli, ecc...



La virtualizzazione si riferisce alle tecnologie basate su hypervisor, mentre la containerization (o OS level virtualization) non richiede alcun hypervisor. Alcune tecnologie di containerization sono:

- Linux Containers (LXC), estensioni del kernel Linux per partizionare le risorse;
- Docker, software di gestione dei container di tipo LXC e altri.

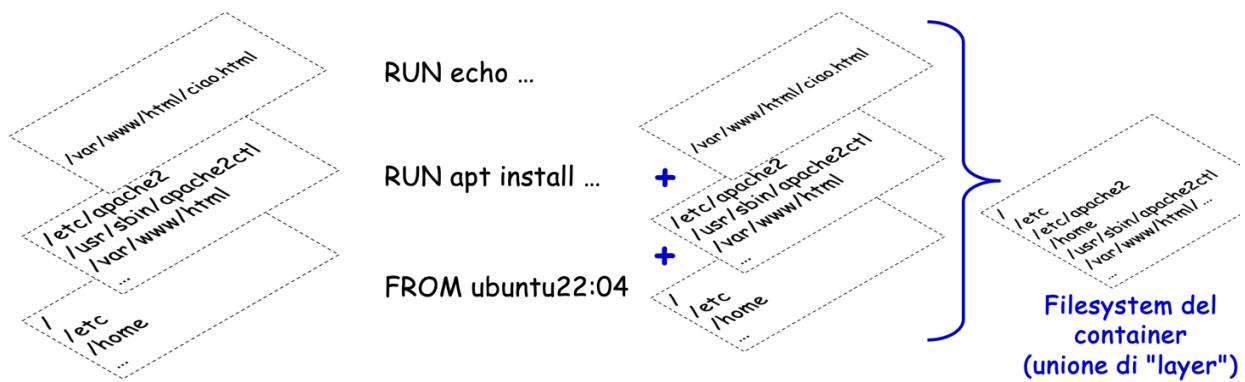


Il Docker Hub consente di condividere immagini Docker con altri. Un docker file si presenta come segue:

```
FROM ubuntu:20.04
RUN apt update
ARG DEBIAN_FRONTEND=noninteractive
RUN apt install -y apache2
RUN apt install -y apache2-utils
RUN apt clean
RUN echo "CIAO MONDO" > /var/www/html/ciao.html
CMD ["apache2ctl", "-D", "FOREGROUND"]
```

In ordine:

- Initial docker image;
- Comandi appts, aggiorna il package database e non solo;
- Crea un file con una semplice pagina web;
- Lancia Apache HTTPd quando il container è in esecuzione.



ANDROID

Android è un popolare Sistema Operativo impiegato in dispositivi mobile, prodotti consumer e sistemi embedded che mira a creare un ecosistema che coinvolga sviluppatori e produttori al di fuori di Google. È basato sul kernel Linux e su altro software open – source ma utilizza i meccanismi tradizionali di UNIX (processi, utenti, IPC, memoria virtuale, ...) in modo innovativo. Nasce per mano della Android Inc., acquisita da Google nel 2005, la quale porterà a compimento lo sviluppo della prima versione del Sistema Operativo, Android 1.0, nel 2007; ad oggi, Google continua a guidare lo sviluppo di Android, rilasciando specifiche e test di compatibilità, contribuendo alla piattaforma open – source e ai kit di sviluppo e sviluppando un app store e i propri servizi commerciali per Android.

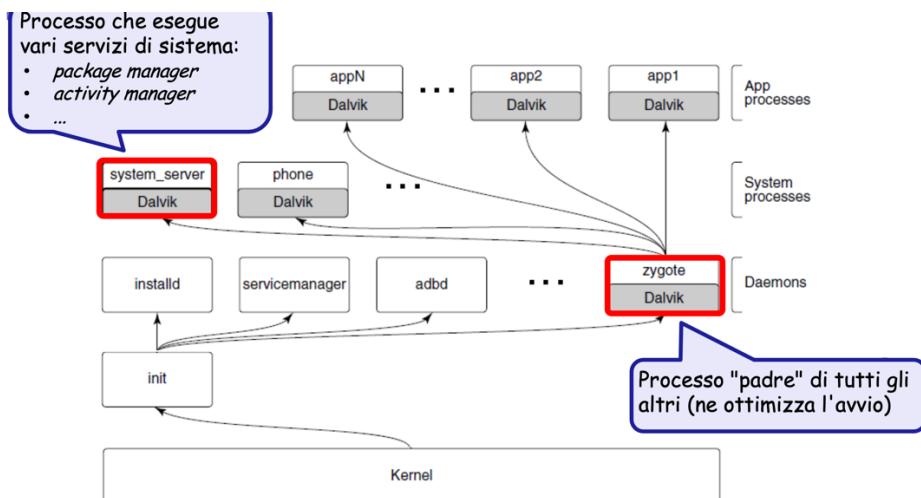
La versione open – source di Android, (AOSP, Android Open Source Project) è una collezione di software e tool che fornisce interfacce di programmazione (API) per applicazioni ed è neutrale rispetto alle applicazioni commerciali, di Google e di altri.



Gli obiettivi di progetto di Android OS sono:

- Ottimizzazione per interazioni brevi, con avvio e switch veloce delle app (in al più 200ms nelle app di base);
- Creazione di applicazioni non più desktop e monolitiche ma piccole e in grado di interagire;
- Gestione automatica della memoria e del ciclo di vita delle app;
- Implementazione di un modello di sicurezza che consente di eseguire app anche “non fidate”.

La gerarchia dei processi in Android si configura come segue:



Mentre i principali componenti di sistema sono:

- **init**, gestisce il boot, avviando demoni di basso livello (tra cui zygote);
- **adb**, crea processi shell per connessioni remote;
- **zygote**, il padre di tutti i processi a più alto livello, basati sul linguaggio Java;
- **Dalvik**, l'ambiente di esecuzione del linguaggio Java (oggi rimpiazzato da ART);
- **System Server**, esegue molti servizi di sistema, tra i quali:
 - Package manager;
 - Activity manager;
 - Power manager;
 - ...
- **Altri processi di sistema**, come Service Manager, RILD, NETD, MediaServer, ...;
- **Android Framework**, un insieme di classi Java usate dalle app come interfaccia API ad Android OS.

È possibile collegare al PC via USB un dispositivo Android ed accedere alla sua shell via ADB, in modo da copiare files, accedere al filesystem, installare app, accedere ai log di sistema, ... L'Android storage è strutturato sulla base delle seguenti directory:

- rootfs (/), partizione in – RAM (initrd) che contiene i file di avvio e configurazione;
- /system, file di Android OS;
- /data, app, contatti, messaggi, config, ...;
- /cache, dati e app acceduti di frequente;
- /sdcard, mountpoint di memorie SD esterne (oppure, partizione “emulata” e salvata in /data).

Un'applicazione in Android è un contenitore, distribuito come **unico archivio zip con estensione .apk** (Android Package, diverso da un tradizionale file eseguibile di Linux), di:

- **Codice**, tipicamente Java o Kotlin;
- **File di configurazione**;
- **Grafica**;
- **Altri dati**.

Un **Android Package** contiene:

- **Manifesto**, un file XML che contiene dichiarazioni su cosa fa l'applicazione e su come eseguirla ed interfacciarsi con essa;
- **Risorse**, file di stringhe, GUI layout in XML, immagini, ...;
- **Codice**, bytecode e librerie native;
- **Firma digitale**, che identifica l'autore in maniera sicura.

Ogni app è identificata da un package Java, com.example.email nell'immagine, mentre i punti di ingresso alle funzionalità dell'app sono descritti dai componenti activity del manifesto. I componenti principali di un app Android sono:

- **Activity**, i punti di ingresso;
- **Receiver**, per gestire i messaggi del sistema o di altre app;
- **Service**, per eseguire operazioni in background;
- **Content provider**, per condividere dati tra app.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.email">
<application>

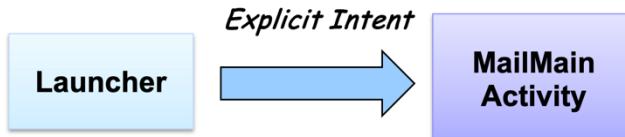
    <activity android:name="com.example.email.MailMainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

    <activity android:name="com.example.email.ComposeActivity">
        <intent-filter>
            <action android:name="android.intent.action.SEND" />
            <category android:name="android.intent.category.DEFAULT" />
            <data android:mimeType="*/*" />
        </intent-filter>
    </activity>
</application>
</manifest>
```

In ogni Activity, quello specificato con l'attributo `android:name` descrive la classe Java che implementa quel componente ed istanziata dal sistema su domanda.

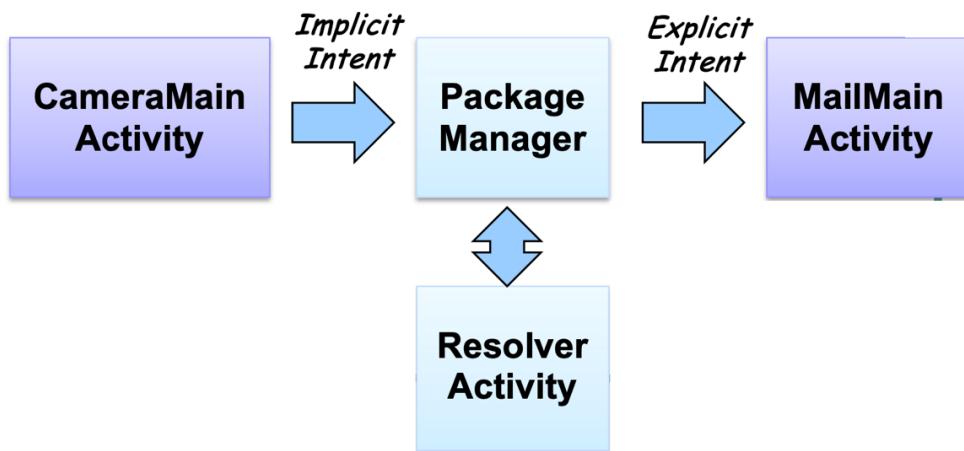
Il package manager è un servizio di sistema che tiene traccia delle applicazioni installate e gestisce i loro permessi, le loro azioni e le relative activity, ed è chiamato da Android quando si verifica un evento da gestire (Intent). Gli Intent possono essere di due tipi:

- **Explicit Intent**, indica il package name dell'app e il class name dell'Activity destinataria in maniera esplicita;



```
$ adb shell "am start -t image/* -d file:///sdcard/Download/UNINA.png  
-a ACTION_VIEW  
-n com.android.gallery3d/.filtershow.crop.CropActivity"
```

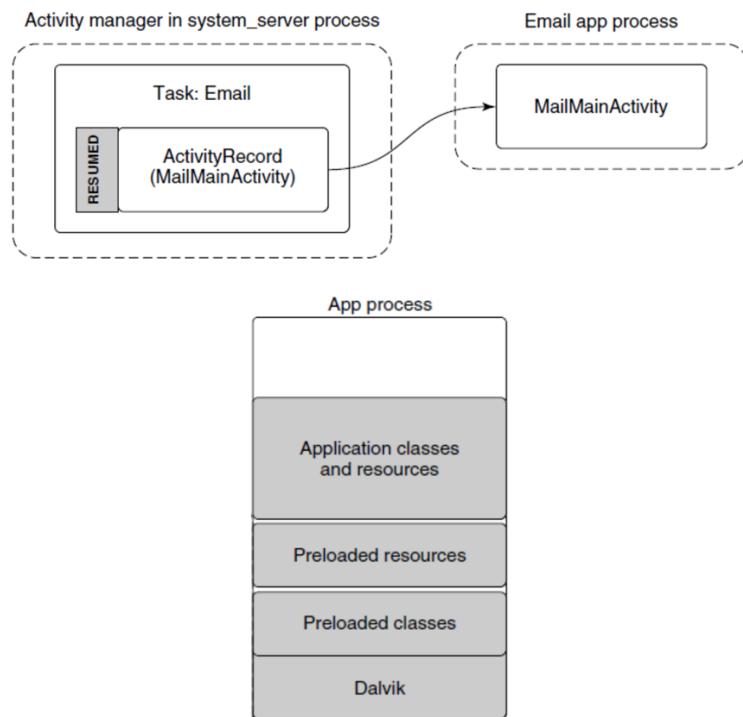
- **Implicit Intent**, descrive un'azione da svolgere (ad esempio, “condividere un’immagine”) e, se l’azione è supportata da più app, mostra un menu di selezione all’utente.



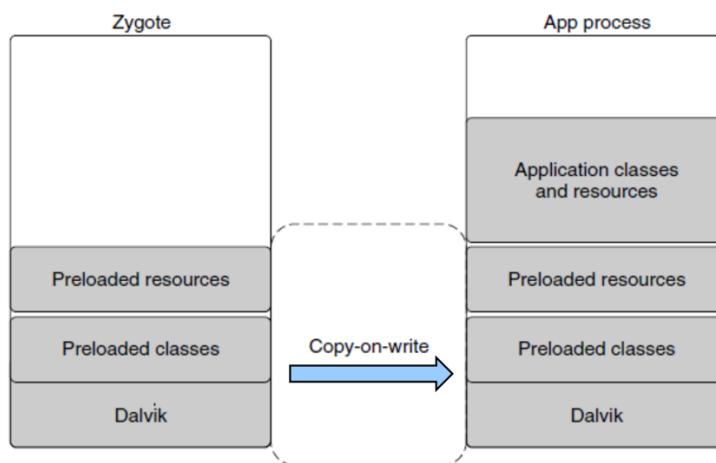
```
$ adb push UNINA.png /sdcard/Download  
$ adb shell "am start -t image/* -d file:///sdcard/Download/UNINA.png"
```

Come già annunciato, una Activity è una schermata grafica che interagisce con l’utente, mentre l’Activity Manager è il servizio che le avvia e gestisce; nell’esempio precedente dell’app Email, MailMainActivity visualizza la lista di messaggi e si avvia al lancio dell’app dalla schermata home, mentre ComposeActivity crea un nuovo messaggio ed è avviata dalla MailMainActivity (o da altre app su richiesta).

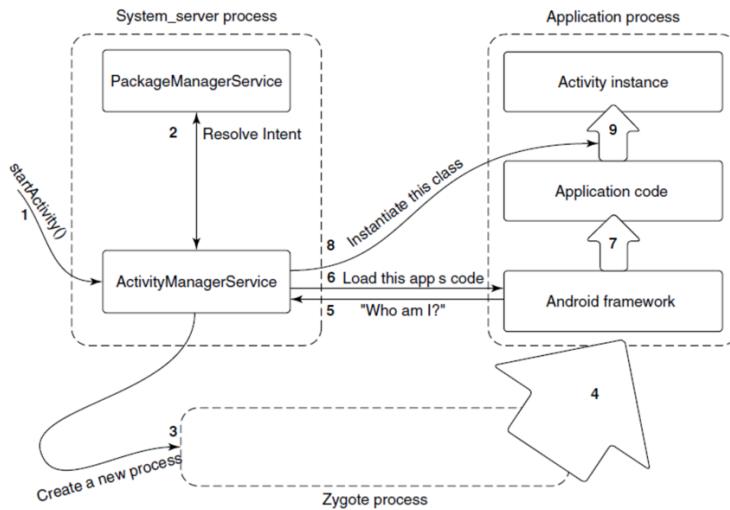
Il ciclo di vita di una Activity prevede che Android avvii e inserisca l’Activity nel processo, facendo inserire all’Activity Manager una riga (record) nella sua lista di Activity:



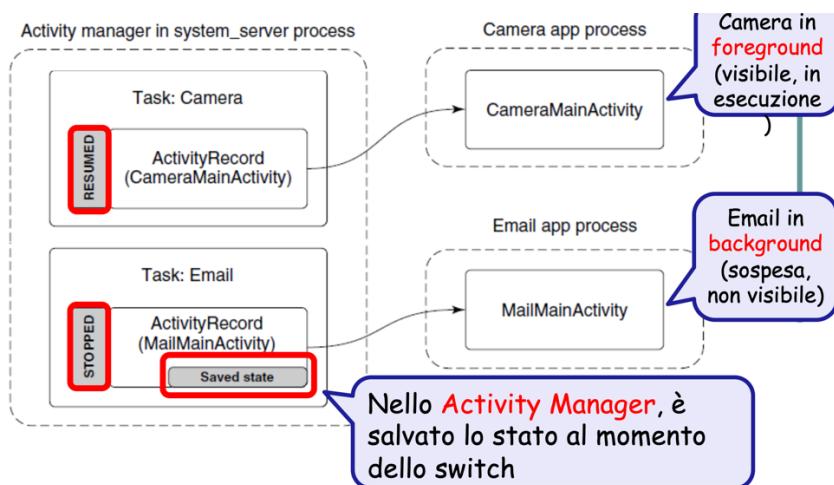
Il nuovo processo avvia un’istanza di Java Virtual Machine (Dalvik nelle prime versioni, Android Run Time ad oggi), che contiene classi e risorse sia di Android OS che dell’applicazione. Per accelerare l’avvio dell’app, **Android clona con `fork()` il processo Zygote, già inizializzato con classi e risorse in un momento precedente:**



La creazione da un processo Zygote passa per diversi step, riassunti dal seguente schema:



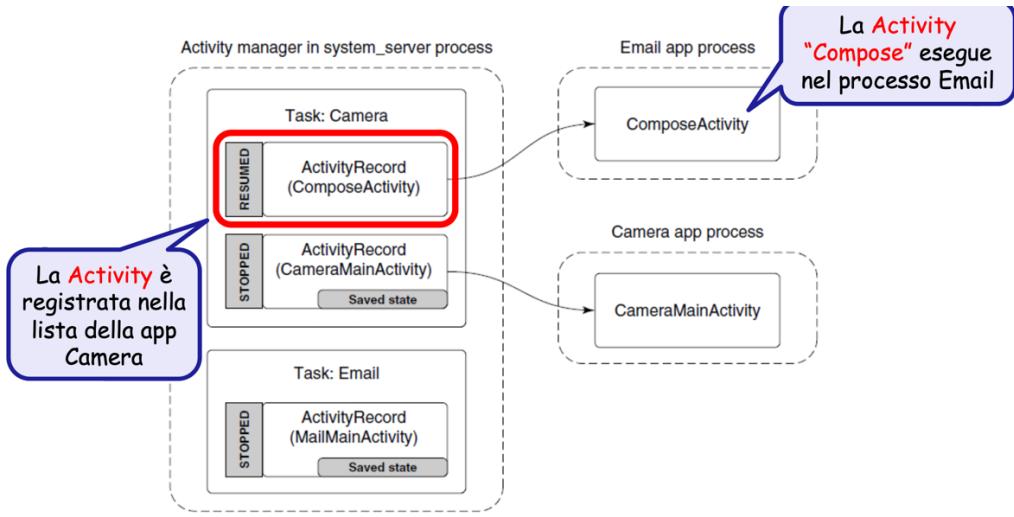
Ora, si supponga che l'utente, dall'applicazione appena aperta, voglia tornare alla schermata home e avviare un'altra app. Il processo in background viene sospeso, ne viene creato un altro per l'app appena aperto e, in seguito al context switch, lo stato del processo precedente viene salvato insieme al record relativo nell'**Activity Manager**:



Infatti, in caso di scarsità di memoria libera, il Sistema Operativo non potrebbe swappare i processi, che vengono invece allocati dopo aver salvato il loro stato. I processi non escono esplicitamente ma sono lasciati cached finché non vengono terminati dal Sistema Operativo; se la RAM scarseggia, lo out – of – memory (OOM) killer uccide i componenti a minor importanza (aka punteggio alto), assegnata dall'Activity Manager:

Category	Description	oom_adj
SYSTEM	The system and daemon processes	-16
PERSISTENT	Always-running application processes	-12
BACKGROUND	Currently interacting with user	0
VISUAL	Visible to user	1
PERCEPTIBLE	Something the user is aware of	2
SERVICE	Running background services	3
HOME	The home/launcher process	4
CACHED	Processes not in use	5

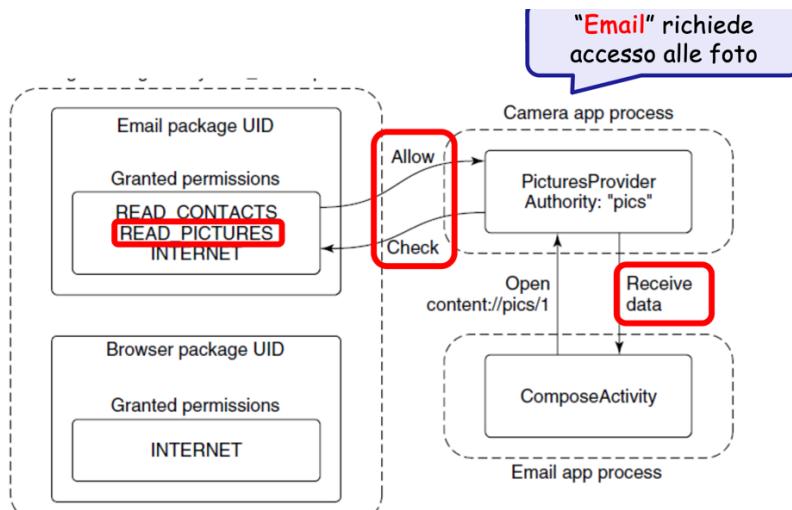
Se una Activity di un'applicazione esegue nel contesto di un'altra, è eseguita nel processo dell'altra applicazione ma nell'Activity Manager sarà sempre registrata nella lista dell'app da cui proviene:



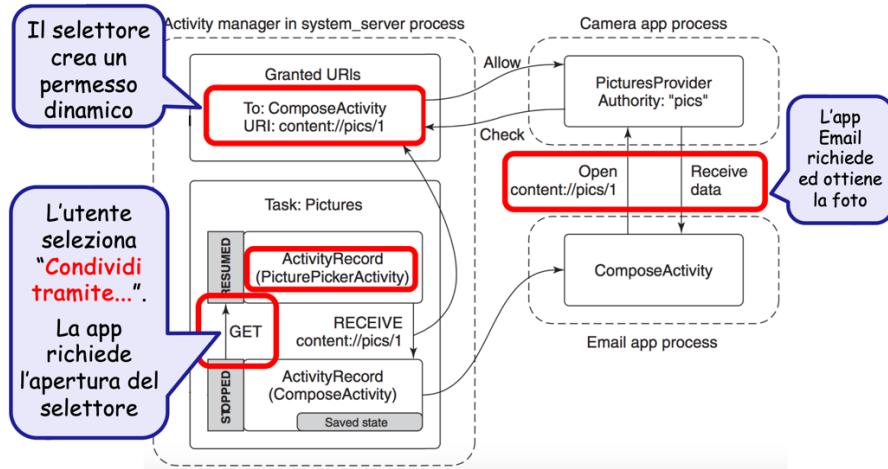
Nei Sistemi Operativi tradizionali, i processi hanno gli stessi permessi dell'utente che li ha avviati e tutti i suoi file e le sue risorse sono accessibili dalle applicazioni eseguite dall'utente stesso; questo modello tradizionale è poco sicuro in un contesto mobile, dal momento in cui l'utente usa spesso app di terze parti, non necessariamente sicure. In Android, l'app è considerata come un ospite del sistema e, quando si installa un'applicazione, si crea un utente Linux univoco e dedicato ad essa, creandogli anche una propria cartella home; questa pratica prende il nome di **sandboxing**.

In Linux/UNIX, tutte le risorse sono rappresentate da file (ad esempio, i file in /dev) ed ogni file ha un solo utente e gruppo proprietario, con relativi permessi di lettura, scrittura ed esecuzione. Questo modello tradizionale è limitato per un contesto mobile, esistono molti tipi di operazioni (ad esempio, avvio, chiamata, scatto foto, lettura della rubrica, ...), diversi per ogni app. Android assegna dei privilegi alle applicazioni, come la possibilità di condividere foto, di accedere ad internet, di accedere alla rubrica, ... in due modi: staticamente, in fase di installazione e garantiti alla nascita, o dinamicamente, in base alle azioni dell'utente e non garantiti finché l'azione stessa non è invocata.

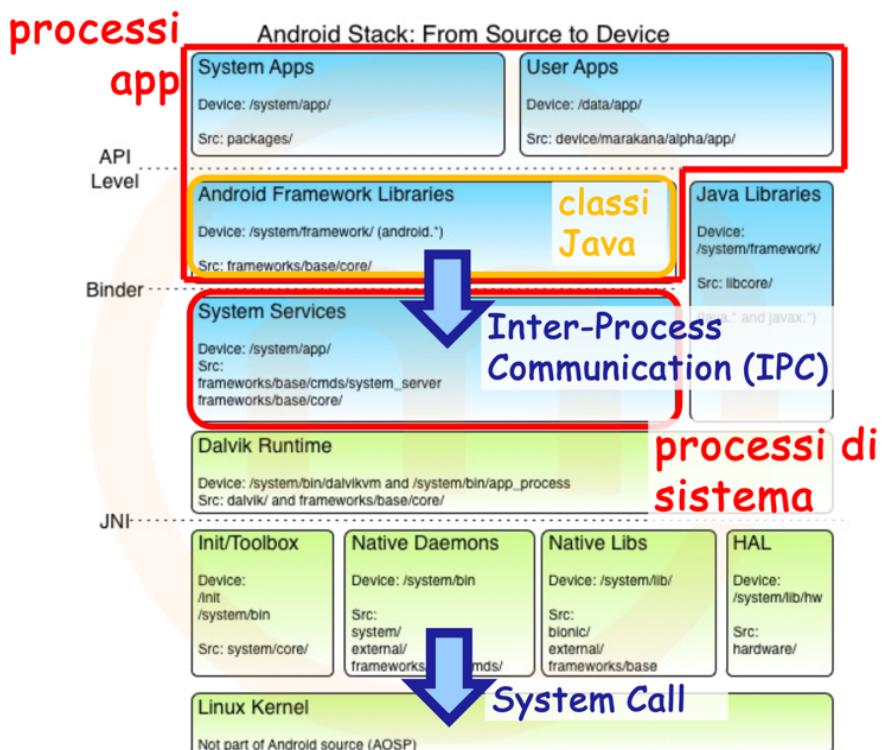
Un permesso statico è schematizzato come segue:

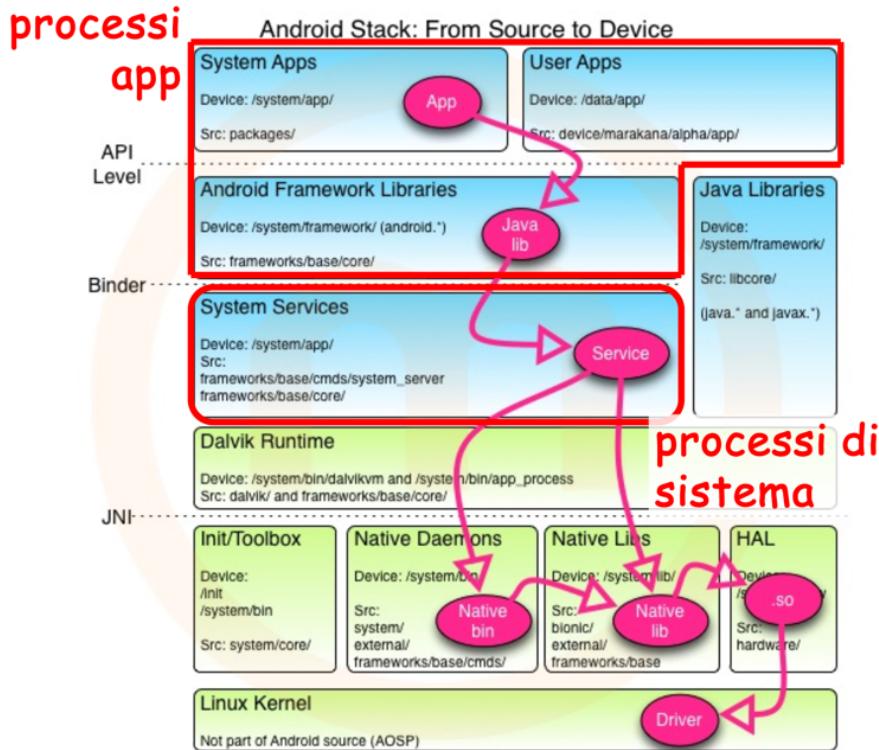


I permessi dinamici danno un'autorizzazione temporanea (come nelle azioni di condivisione) e garantiscono una maggiore flessibilità. Un esempio di permesso dinamico occorre quando l'utente scatta una foto e clicca sul tasto condividi; va, poi, selezionata un'applicazione (ad esempio, Email) con cui effettuare la condivisione e che acquisisce un permesso dinamico. Un permesso dinamico è schematizzato come segue:

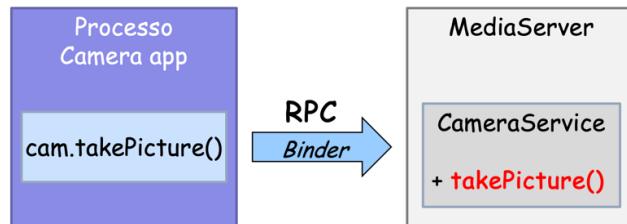


Il Sistema Operativo Android è strutturato in servizi (Fotocamera, Sensoristica, Audio, Connattività, ...) e le app usano i servizi tramite classi Java, dette **Android Frameworks**:



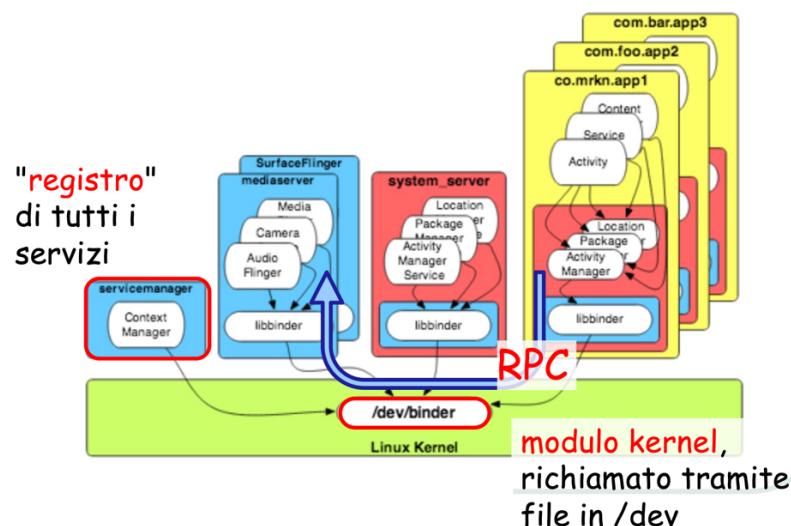


Ogni sistema, poi, lavora in modo differente nel proprio sottosistema. Il Binder è il meccanismo IPC di Android per l'interazione tra app, è basato su Remote Procedure Call (RPC) ed è usato per inviare Intent e chiamare servizi di sistema.

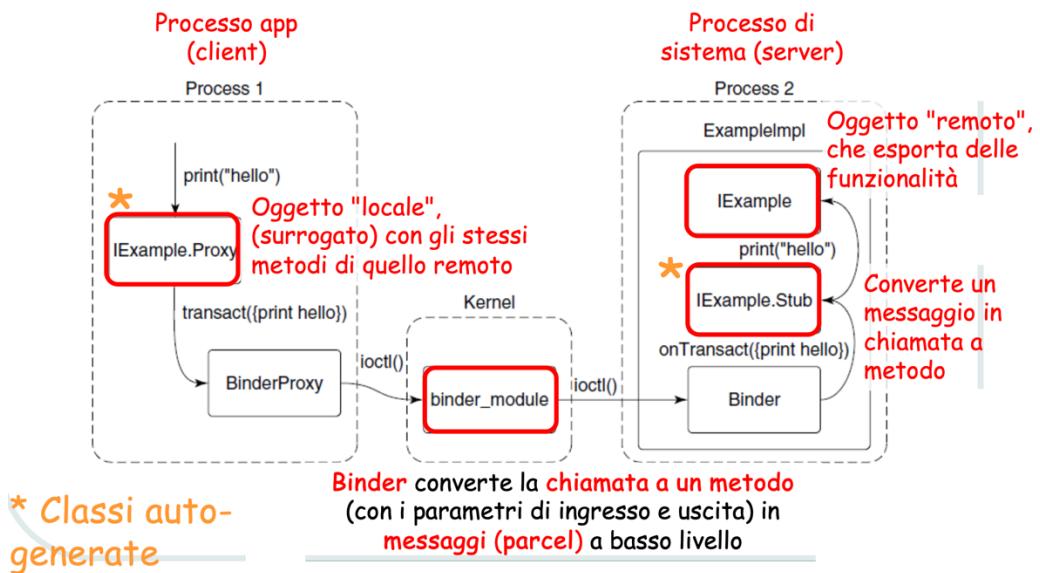


Android non utilizza le IPC UNIX System V (semafori, shared memory e message queue), non forniscono un comportamento robusto per ripulire le risorse di app bugate o malevole.

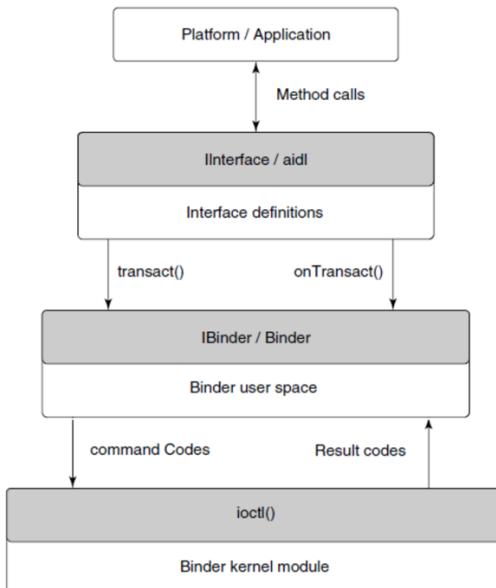
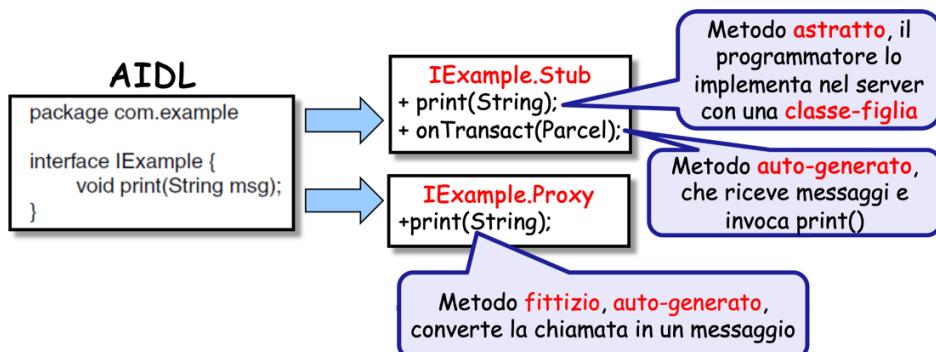
Un IPC Android si configura così come segue:



Mentre il RPC:



Il programmatore definisce in file AIDL (Android Interface Definition Language) l'interfaccia delle remote procedure call (in termini di moduli e parametri), mentre un compilatore elabora questo file producendo una classe Proxy, usata dall'app client, e una classe Stub, usata nel server e classe padre del servizio.



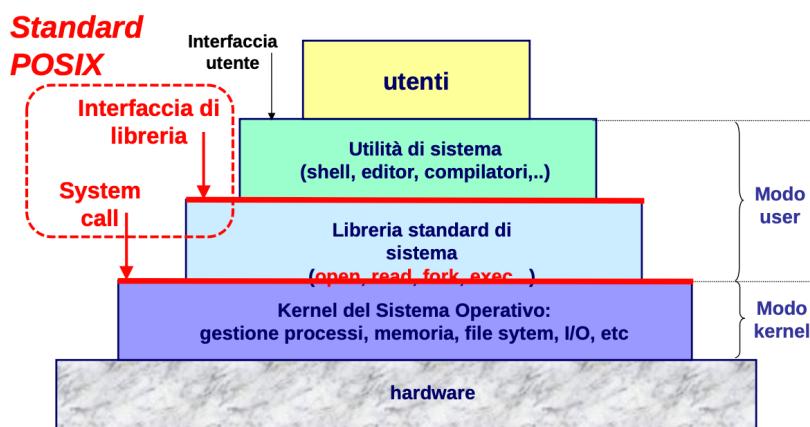
Le applicazioni invocano metodi su un oggetto Proxy, la cui classe, al tempo di compilazione e in linguaggio AIDL, è **generata in automatico** e collegata all'app. In user – space, Binder è una libreria condivisa (`libbinder.so`) con una API object – oriented, mentre in kernel space è un modulo kernel accessibile tramite il file `/dev/binder` e la system call `ioctl()`.

APPROFONDIMENTI LINUX/UNIX

INTRODUZIONE LINUX UNIX

UNIX è una famiglia di Sistemi Operativi multitasking (time sharing in particolare) nato con l'obiettivo di sostituire i grossi e pesanti sistemi per la gestione di mainframe presenti alla fine degli anni 60; è un Sistema Operativo molto robusto, flessibile e portabile, nonché noto per la sua multiutenza, ma ha un processo di configurazione abbastanza complesso, con un'interfaccia shell – oriented non molto intuitiva.

Il MIT, in collaborazione con Bell Labs di AT&T, sviluppa a ridosso degli anni 70 il Sistema Operativo MULTICS (MULTIplexed Information and Computing Service), il precursore degli attuali sistemi time sharing, ma il progetto fallisce a causa della sua eccessiva complessità. Nel decennio successivo, Ken Thompson presso i Bell Labs realizza un sistema più semplice per un minicomputer PDP – 7, denominato UNICS, poi rinominato UNIX. In collaborazione con Dennis Ritchie, il sistema UNIX fu riscritto in linguaggio C e portato su altre piattaforme maggiormente popolari, come DEC e PDP – 11 (piccoli mainframe diffusi nelle università). AT&T, che inizialmente non poteva commercializzare UNIX, concede il codice sorgente in licenza a varie università, che ne svilupparono nuove estensioni. In particolare, BSD (Berkeley Software Distribution) introdusse la memoria virtuale, la paginazione, varie migliorie al filesystem e il supporto ai protocolli implementati nelle reti TCP/IP. Nel 1984 AT&T si scinde e altri vendor commercializzano una versione di UNIX, System V, sviluppata parallelamente a BSD. Per favorire la portabilità delle applicazioni tra le varianti di UNIX, nel 1988 l'IEEE definisce lo standard POSIX, che stabilisce un'interfaccia di libreria standard per le applicazioni a partire dall'intersezione di System V e BSD.

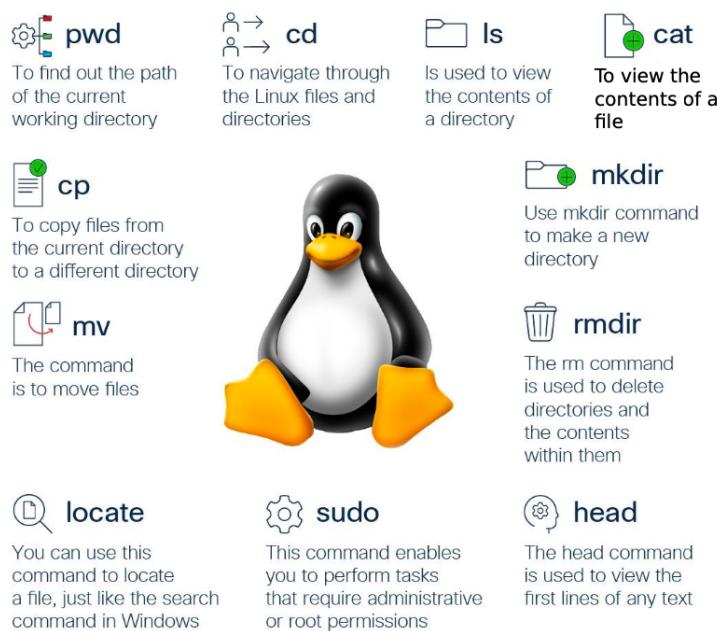


Linus Torvalds, studente dell'Università di Helsinki, sviluppa Linux, un Sistema Operativo POSIX – compatibile per processori Intel 386 che diventa lo “UNIX open source” di riferimento di un'ampia comunità di sviluppatori.

Il kernel Linux è un singolo progetto e negli anni sono proliferate molte distribuzioni (o distro) di questo Sistema Operativo, alcune free e alcune commerciali. Una distribuzione include il kernel, delle applicazioni (come browser, editor, compilatori, ...) e degli strumenti per l'amministrazione di sistema. Ad esempio, GNU – Linux è una distribuzione che permette di associare a Linux un'interfaccia grafica e combina il kernel Linux alle Utilità e Libreria standard di sistema di GNU. Linux, in sé e per sé, è un free software distribuito con licenza GNU General Public License (GPL); non è un software di dominio pubblico o gratis, sono posti dei vincoli di utilizzo:

- Se si modifica il programma, gli autori precedenti della versione originale devono essere menzionati, lasciando intatto il loro copyright;
- Non è consentito re – distribuire il programma senza anche distribuire il codice sorgente, con le eventuali modifiche;
- Non è consentito re – distribuire il programma con una licenza d'uso non libera.

Così come UNIX, Linux è un sistema multiutente, multiprogrammato (time sharing) e interattivo. Il terminale (o console) è un dispositivo combinato display/tastiera e non è considerabile un computer a sé stante, era usato come client nei primi sistemi e, pertanto, ha un'interfaccia a caratteri. La shell, invece, è un programma che interagisce con l'utente tramite comandi testuali e può essere considerato come l'emulatore di un terminale (simula la console tradizionale in una finestra grafica); in una shell è possibile avviare e gestire processi, operare sul filesystem (creazione, copia e spostamento file) o semplicemente amministrare il sistema (aggiornamenti, backup, networking, ...). In una shell è comune trovare il carattere \$, che indica lo stato di attesa del prompt dell'utente; alcuni comandi (tutti minuscoli) di UNIX per la sua shell sono i seguenti:



Molti di questi comandi necessitano di parametri di ingresso, sotto forma di stringhe di caratteri separate da spazi:

cp /percorso/file /percorso/destinazione

cp è il comando, **/percorso/file** e **/percorso/destinazione** i parametri. Alcuni comandi accettano come parametri (opzionali) delle stringhe prefissate, dette flag, che iniziano per -.

In modo simile a Windows, i file in Linux sono identificati da un percorso assoluto composto da una sequenza di nomi di cartelle (a partire dal root) separati da / più il nome del file; in Windows:

C:\User\roberto\ciao.txt

In Linux:

/home/roberto/ciao.txt

Nella shell, **può diventare scomodo e lungo dichiarare il percorso assoluto di un file** (soprattutto se innestato in molte cartelle) e, pertanto, **è possibile usare una loro versione breve, il percorso relativo:**

```
/home/roberto/ciao.txt → ciao.txt
```

La shell riconosce un percorso relativo dall'assenza dello / iniziale e aggancia al percorso del file quello della CWD (Current Working Directory):

```
ciao.txt → /home/roberto/ + ciao.txt
```

Una volta avviata la shell, potrebbe non essere intuitivo capire la directory in cui ci si trova; **il simbolo ~ è un'abbreviazione di /home/lucamariaincarnato/**:

```
~$ → /home/roberto/ + attesa user prompt
```

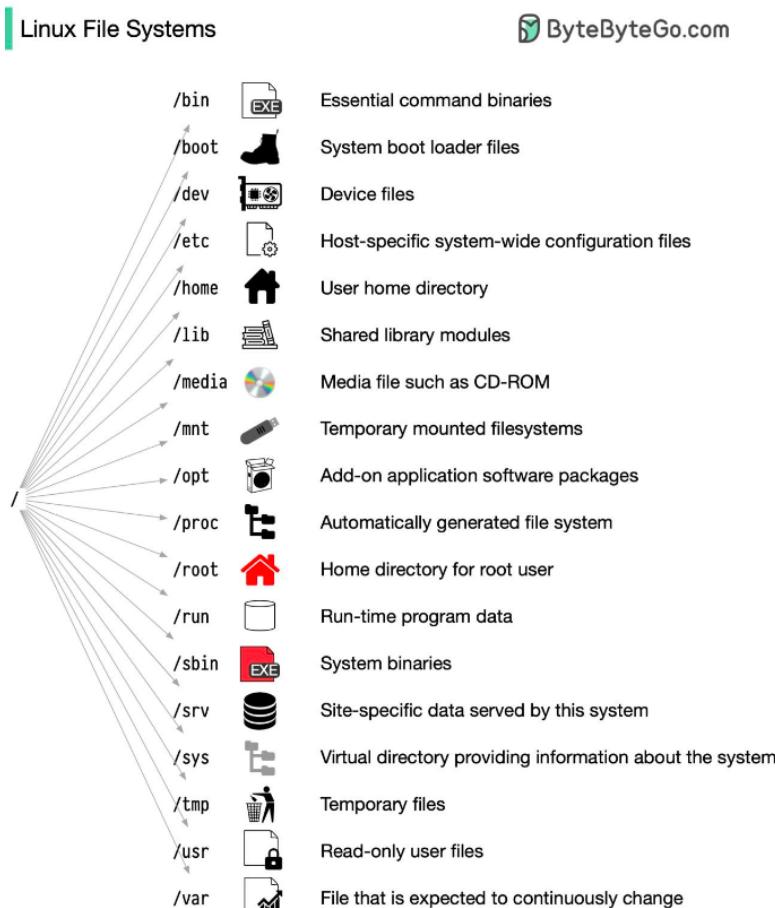
Cambiando directory:

```
~/Desktop$ → /home/roberto/Desktop + attesa user prompt
```

Esistono altri due simboli da comprendere per la navigazione in shell: **.., che punta alla CWD, e . . . , che punta alla directory padre della CWD:**

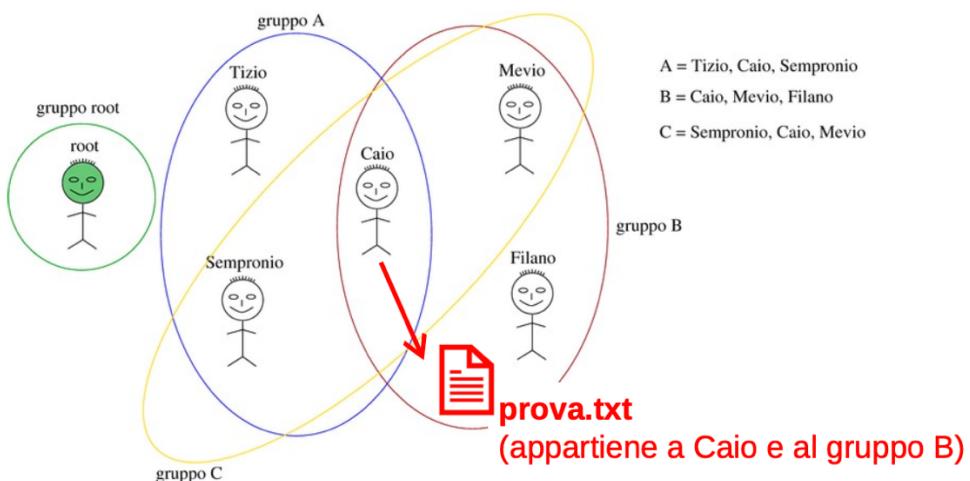
```
~/Desktop$ cd .. /Desktop → /home/roberto
```

Un **filesystem tipico di Linux** è così composto:



Per lanciare un programma non di sistema, va scritto il suo percorso e, in questo caso, usando percorsi relativi è necessario specificare il . della CWD, in modo da comunicare che il programma è in questa directory piuttosto che in un'altra.

In Linux, ogni file ha un utente proprietario che, di default, è colui che ha creato il file stesso; in questo modo, si può anche impedire agli altri utenti del computer di leggere o cambiare il file. Inoltre, è permessa in Linux la divisione degli utenti in gruppi, utile alla condivisione selettiva dei file:



Alla creazione di un utente, viene creato un gruppo omonimo a lui riservato, in modo che i file di "Tizio" appartengano anche al gruppo "Tizio". È possibile cambiare l'utente ed il gruppo proprietari del file con il comando **chown**; ad esempio:

```
chown so:sviluppo miofile.txt
```

Con questo comando, gli utenti del gruppo **sviluppo** sono proprietari del file specificato. Ad un file possono essere attribuiti i seguenti permessi:

r : readable w : writable x : executable	} per	{ proprietario { gruppo altri utenti
--	-------	--

Ad esempio:

```
rwx r-- r--
```

I primi tre specificano per il proprietario, i secondi tre per il gruppo e i terzi tre per gli altri utenti del gruppo. Per poter specificare un comando con dei permessi al suo interno, può essere usata la versione binaria (1 per il permesso concesso e 0 per il permesso non concesso) o ottale (ottenuta dalla traduzione dal binario e più compatta).

Il comando **chmod**, invece, assegna i permessi specificati al file ma può essere eseguito solo dal proprietario del file (o dall'amministratore); ad esempio:

```
chmod 644 myfile.txt
```

Sono assegnati al file i permessi **rwx r-- r--**.

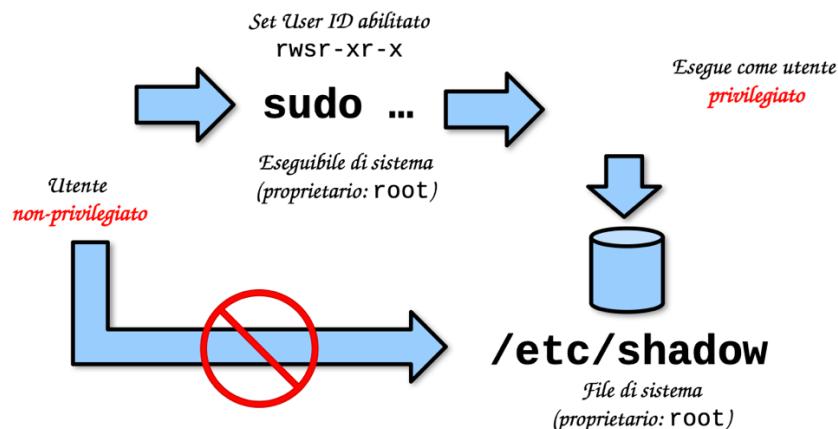
In UNIX, l'utente **root** (o superuser) ha privilegi speciali rispetto agli altri utenti:

- Accesso a tutto il filesystem;
- Accesso a tutte le periferiche;
- Creazione e cancellazione di tutti gli utenti;
- ...

Anche gli utenti non – privilegiati hanno bisogno di accedere a cartelle o file riservati, tipo per cambiare la password, e per farlo hanno due possibilità:

- **Comando su**, avvia una nuova shell come utente privilegiato (**root**);
- **Comando sudo**, esegue un singolo comando come utente privilegiato ma la shell rimane non privilegiata.

La differenza sostanziale tra i due comandi consiste nel fatto che il primo modifica i privilegi della shell in cui viene avviato, il secondo esegue comandi come se si fosse un altro utente. A prescindere da quale si utilizza, quando un programma lancia con un comando **su/sudo**, viene elevato a programma amministratore; ciò è dovuto al fatto che entrambi i comandi sono eseguibili setuid (Set User ID).



Le distribuzioni Linux danno anche la possibilità di installare pacchetti pre – compilati di software open source tramite un comando, specifico per la distribuzione (apt per Debian, Ubuntu e derivate e yum per Red Hat, Fedora e CentOS), scaricando automaticamente anche eventuale altro software aggiuntivo di cui il programma necessita (dipendenze). I comandi più utilizzati sono:

- Aggiornare il database dei pacchetti:

```
sudo apt update
```

- Installare un pacchetto:

```
sudo apt install <nomepacchetto>
```

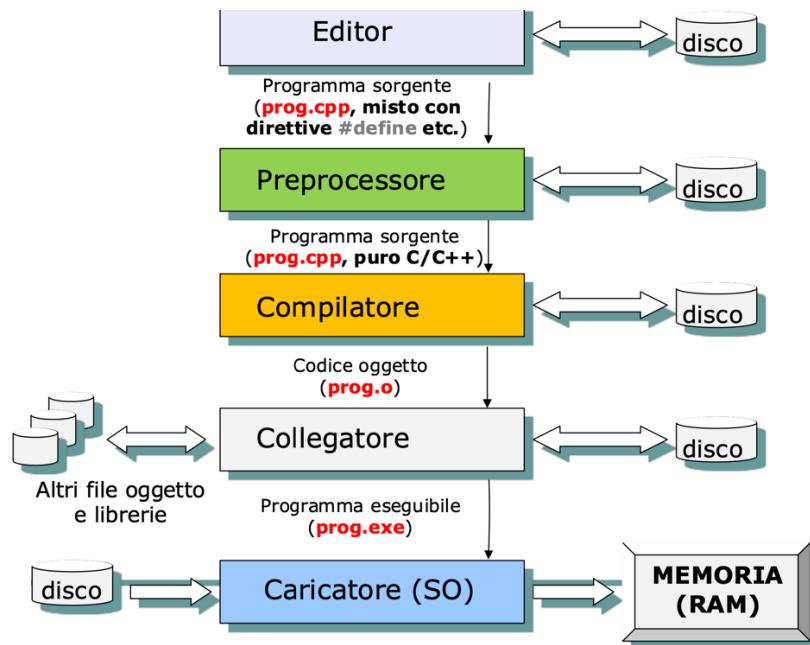
- Cercare un pacchetto, il cui nome o descrizione contenga la stringa selezionata:

```
apt search <stringa>
```

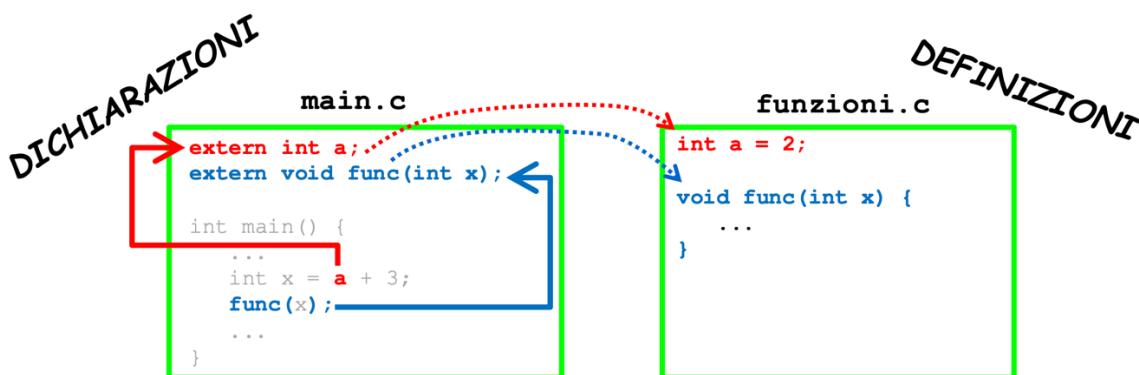
La documentazione di un comando è accessibile tramite il comando **man** e consiste di più sezioni, tra cui Commands, System Calls e Library Functions.

IL CICLO DI SVILUPPO MODULARE

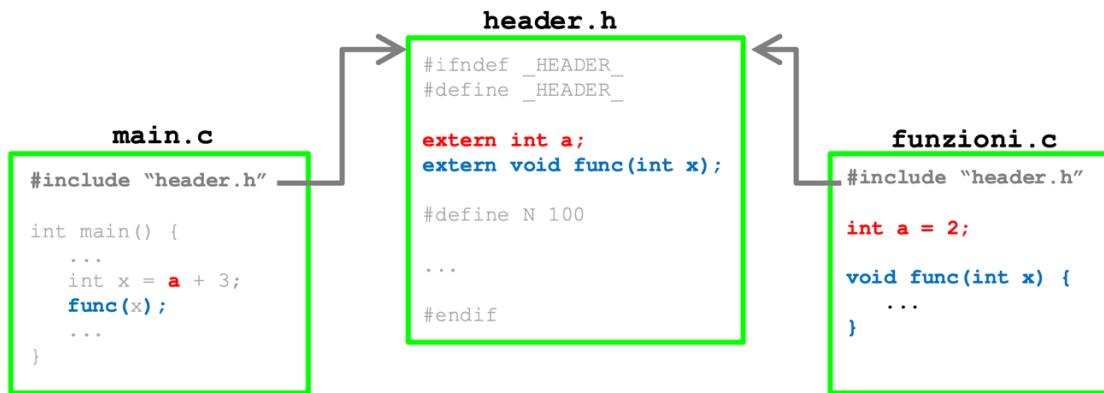
Per lo sviluppo di un software modulare, si parte dai singoli file sorgenti (detti unità di compilazione) tramite un editor che accede ai file in questione sul disco. Una volta creato il file sorgente, parte il processo di compilazione, passando per il preprocessore in modo che un file **.cpp** possa produrre un file **.o** (un file oggetto è la traduzione del codice sorgente in linguaggio macchina, codice binario). Dopo la produzione del file oggetto, si passa al linking, una pratica che rileva gli altri file oggetto e le librerie necessarie alla corretta esecuzione del software e le unisce in un unico eseguibile, caricato (**loading**) poi in memoria RAM dal loader con i relativi dati necessari. Il tutto è schematizzato come segue:



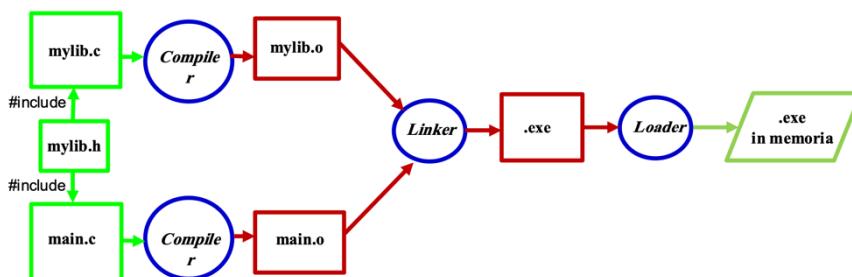
I linguaggi C e C++ hanno diversi meccanismi per garantire lo sviluppo modulare: la compilazione separata, l'inclusione testuale e l'uso dei prototipi di funzioni. Per sviluppo modulare si intende la pratica con la quale sono separati specifica e implementazione di una porzione del software, detta modulo, in modo da poter garantire l'immutabilità della prima e la possibilità di modificare la seconda in ogni fase dello sviluppo del software. La modularizzazione, nella pratica, avviene tramite la distinzione tra dichiarazione e definizione: una struttura logica è dichiarata quando è fornita al compilatore, mentre è definita nel momento in cui le è assegnato un valore o una struttura fisica.



Un file può usare variabili e funzioni definite in un altro file. Con i **file header** è anche possibile separare le istruzioni dichiarative da quelle computazionali, riservando queste ultime ai file .cpp e le prime ai file .h:



In entrambi i casi, **il linking controlla**, a partire dal file oggetto prodotto da ogni file del progetto, se ci sono delle dichiarazioni e delle definizioni in comune e le collega fisicamente in uno stesso eseguibile.



Linux fornisce automaticamente i compilatori open – source gcc e g++, rispettivamente per i linguaggi C e C++. La compilazione, il linking e il loading avvengono tramite i seguenti comandi:

```

// Compilazione del file oggetto (aggiungendo il flag -c)
$ gcc -c main.c -o main.o
$ gcc -c mylib.c -o mylib.o
// Linking dell'eseguibile (rimuovendo il flag -c)
$ gcc main.o mylib.o -o programma
$ ./programma           // Loading ed esecuzione
  
```

Il codice oggetto in Linux è in un formato standard, chiamato ELF (Executable and Linkable Format), ed integra un tool di analisi dei file oggetto, objdump e nm. Con il comando **\$ objdump -d main.o**, si può osservare nel dettaglio il formato ELF del file e il suo disassemblamento, con una rappresentazione in linguaggio assembler (sintassi AT&T) affiancata al codice in linguaggio macchina:

```
UjDHuo:      file format elf64-x86-64

Disassembly of section .init:
00000000004000e8 <.init>:
4000e8:    50                      push   %rax
4000e9:    58                      pop    %rax
4000ea:    c3                      ret

Disassembly of section .text:
00000000004000f0 <.text>:
4000f0:    51                      push   %rcx
4000f1:    31 d2                  xor    %edx,%edx
4000f3:    31 f6                  xor    %esi,%esi
4000f5:    41 b8 38 00 00 00      mov    $0x38,%r8d
4000fb:    b9 01 00 00 00      mov    $0x1,%ecx
```

Le serie di 0 che si notano in un file di disassemblaggio sono puntatori vuoti alle funzioni e alle variabili definite in altri file e che non sono state ancora linkate. Se si volesse eseguire lo stesso comando ma su un file eseguibile, sarebbero mostrati a schermo i disassemblaggi di tutti i componenti del file e sarebbe possibile notare come la sostituzione dei puntatori, prima vuoti, con i link corretti.

Il comando make è una utility usata specialmente per semplificare la compilazione separata dei programmi ed agisce come segue:

```
$ ls
Makefile mylib.c main.c
$ make
↑ legge Makefile
gcc -c main.c -o main.o
gcc -c mylib.c -o mylib.o
gcc main.o mylib.o -o programma

$ ls
Makefile mylib.c mylib.o
main.c main.o programma
```

Il Makefile è un file di configurazione, con una lista di regole contenenti:

- **Nome della regola** (detto target);
- **File necessari per l'esecuzione** della regola (dette dipendenze);
- **Comandi da eseguire.**

Tipicamente, un Makefile contiene più regole per eseguire più operazioni con un solo comando (compilazione, linking, cancellazione del file oggetto, ...). Un esempio di Makefile è il seguente:

```
file.o: file.cpp file.h
g++ -c file.cpp -o file.o
```

Con file.o target, file.cpp e file.h dipendenze e g++ -c file.cpp -o file.o comando da eseguire. **Il comando make controlla se le dipendenze sono più recenti del target:**

- **Se è vero, significa che i sorgenti sono stati modificati** dall'ultima volta che il target è stato compilato ed il comando viene eseguito;
- **Altrimenti, non ci sono state modifiche** e il comando non viene eseguito.

È anche possibile specificare dipendenze che sono a loro volta target nello stesso makefile. Quando si esegue il comando make, si eseguono i tre target ricorsivamente:

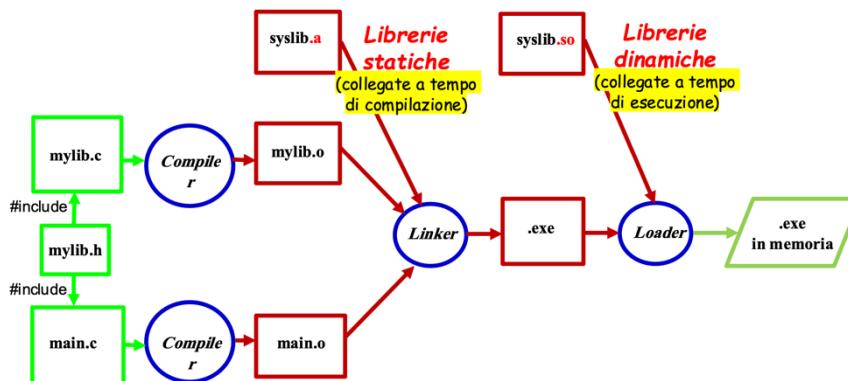
- Si esegue il primo comando del Makefile;
- Si verifica se le dipendenze sono state aggiornate:
 - Se sì, si esegue il relativo comando;
- Si ritorna ad eseguire la prima regola.

In corrispondenza di questo tipo di Makefile, è possibile anche far eseguire solo un target, specificandolo dopo il comando **make**. In un Makefile, è utile specificare anche una regola di **clean**, senza dipendenze, che cancella file oggetto ed eseguibili e che viene eseguita tramite **make clean**:

```
clean:
    rm -f *.o
    rm -f nomeEseguibile
```

I Sistemi Operativi semplificano il riuso del codice mediante le omonime librerie; ad esempio, la C standard library (**libc**) fornisce le funzioni **printf()**, **strlen()**, ecc. Una libreria entra in gioco essenzialmente in due momenti, sulla base del tipo di libreria:

- Libreria statica, è collegata a tempo di compilazione e interviene in fase di linking;
- Libreria dinamica, è collegata a tempo di esecuzione e interviene in fase di loading.



Una libreria statica è un archivio di più file oggetto, ha il prefisso **lib** e l'estensione **.a**. Generalmente, in Linux le librerie sono conservate in sotto – directory della cartella **/usr**, con i file header nella directory **include** e i file oggetto, già compilati, in **user**. Per collegare un eseguibile ad una libreria si può usare il seguente comando:

```
$ gcc -o prog obj1.o ... objN.o -L/path/ -lprova
```

Con:

- **-L** il percorso da cui prelevare la libreria (. se è nella directory corrente);
- **-l** il nome della libreria da collegare (senza prefisso **lib** e senza estensione **.a**).

Le librerie statiche hanno un maggior consumo di spazio su disco degli eseguibili ed un maggior consumo di spazio in RAM (se due programmi usano la stessa libreria, essa sarà copiata in entrambi i programmi); inoltre, se una libreria viene aggiornata, è necessario ricompilare anche le applicazioni che la usano. A differenza di quelle statiche, le librerie dinamiche sono collegate in fase di esecuzione e, sebbene mantengano il prefisso **lib**, l'estensione cambia in **.so**, hanno un minor consumo di spazio su disco degli eseguibili e in memoria RAM, visto che essa è caricata una volta solo per tutti i programmi, ed è possibile aggiornarle senza dover ricompilare tutti i

programmi che le usano. La libreria dinamica viene indicata in fase di linking, in modo da poter essere accoppiata in fase di loading, con il comando:

```
$ gcc -o prog obj1.o ... objN.o -L/path/ -lprova
```

Quando viene eseguito il comando \$./prog, il Sistema Operativo carica anche libprova.so, dalle cartelle di sistema in memoria RAM. Con il comando ldd è possibile visualizzare le librerie dinamiche usate dal programma specificato come parametro.

Se la libreria dinamica non è in una cartella di sistema, occorre indicarne il percorso tramite la shell, con la variabile di sistema LD_LIBRARY_PATH contenente una lista di percorsi (separati da :):

```
$export LD_LIBRARY_PATH+=":/path"  
$ ./prog
```

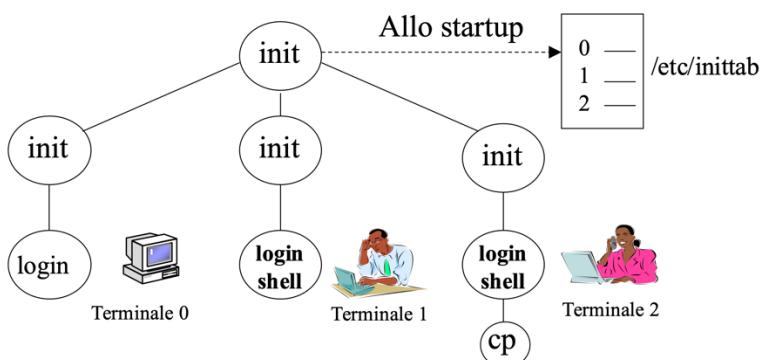
Git è un VCS (Version Control System), un sistema di gestione del codice sorgente con il quale è possibile salvare in remoto i codici sorgenti e la loro cronologia per dare la possibilità di condividere un progetto tra sviluppatori; GitHub, invece, è la piattaforma più famosa per l'utilizzo online di Git. Il flusso di lavoro tra un server GitHub e un utente locale è basato su tre comandi:

- git clone, salva in locale una copia dei file di un progetto in remoto;
- git add e git commit, salva delle modifiche in una cronologia locale;
- git push, invia le modifiche al server remoto.

Una buona pratica consiste nel fare commit ogni volta si aggiunge una nuova funzionalità o si corregge un bug, evitando micro commit e commit giganti e inserendo messaggi di commit esplicativi.

I PROCESSI IN UNIX

In UNIX i processi sono organizzati gerarchicamente e uno di essi può essere **creato solo da suoi simili**, tramite la chiamata di sistema **fork()**, ad eccezione del processo **init**, che viene **creato all'avvio del kernel**.



init legge il file /etc/inittab per determinare il numero di terminali del sistema da attivare e, per ogni terminale, genera un processo figlio che esegue il programma /bin/login.

Quando l'utente inserisce il suo nome e la sua password, il programma **login** li verifica consultando il file **/etc/passwd**; in caso affermativo, esegue il programma **shell**, ossia l'interprete dei comandi.

Ogni processo ha un identificatore univoco, il **PID** (Process IDentifier), un numero intero compreso tra 0 e 32768 (in sistemi a 32 bit) ed è assegnato dal kernel al momento della sua creazione; un processo può consultare il proprio PID con la chiamata di sistema:

```
int getpid();
```

Nella documentazione e negli header di sistema di UNIX, il tipo **int** è sostituito da **typedef pid_t, size_t, ...** per una maggior portabilità tra diverse CPU. Ogni processo, inoltre, ha un processo padre (eccetto **init**) con relativo PID, ottenuto attraverso la chiamata di sistema:

```
int getppid();
```

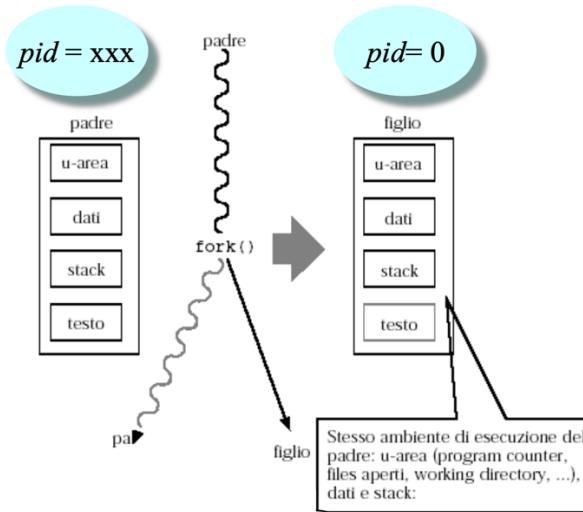
La chiamata a **sleep** sospende un processo, facendo avvenire la transizione in stato **blocked**, per un certo numero di secondi:

```
unsigned int sleep(int sec);
```

Come preannunciato, la creazione di nuovi processi avviene tramite la chiamata a **fork**:

```
int fork(void);
```

Crea una **copia esatta (figlio)** del processo chiamante (**padre**), con lo stesso codice e gli stessi dati, inclusi registri, stack, heap e dati globali. I processi padre e figlio sono identici ma le loro risorse sono delle copie distinte, non sono condivise, e, pertanto, le modifiche ai dati di uno dei due non sono visibili all'altro.



Tipicamente, in un programma concorrente si vuole che il processo figlio esegua un'attività diversa dal padre e per differenziare i due, nonostante il figlio sia clone del padre, il Sistema Operativo predispone una variabile che ha valore diverso nei due processi e lo pone come valore di ritorno della chiamata a **fork**. Il Program Counter (PC) è uguale per padre e figlio, entrambi eseguono dallo stesso punto in cui è stata chiamata la **fork** ma questa ritorna un valore intero che può differenziare i due processi:

```

pid = fork();

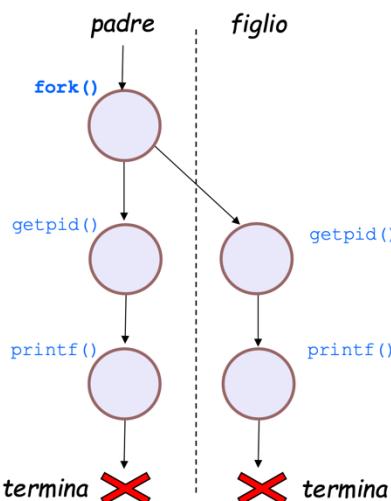
if (pid > 0) {
    /* codice eseguito
       dal padre */

    wait();
}

else if (pid == 0) {
    /* codice eseguito
       dal figlio */
}

```

Tuttavia, non si può verificare che il padre stampi prima del figlio o viceversa, questo dipende dallo scheduler del Sistema Operativo, neppure schematicamente:



fork è una system call potenzialmente molto onerosa, si occupa di allocare memoria per il processo figlio copiando memoria e registri del padre su di lui.

In caso di errori, le chiamate di sistema possono terminare con un insuccesso, come memoria esaurita, limiti di sistema raggiunti (come numero massimo di processi), mancanza di permessi, risorsa non trovata, ... Per gestire gli errori, si controlla il valore di ritorno e, se è negativo è indicato un fallimento, ma si può anche stampare un messaggio di errore con la chiamata a perror, aggiungendo informazioni sull'errore in base alla variabile di sistema errno:

```

pid = fork();

if (pid > 0) {
    /* codice eseguito
       dal padre */

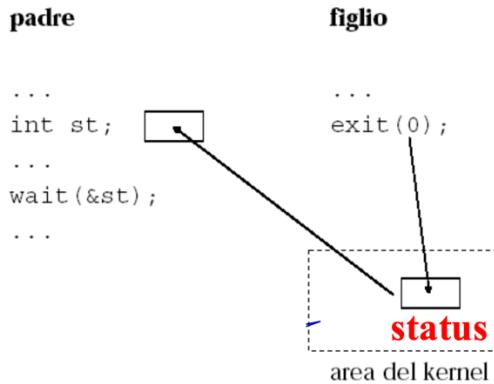
    wait();
}

else if (pid == 0) {
    /* codice eseguito
       dal figlio */
}

else if (pid < 0) {
    /* chiamata fallita */
    perror("FORK FALLITA");
    exit(1);
}

```

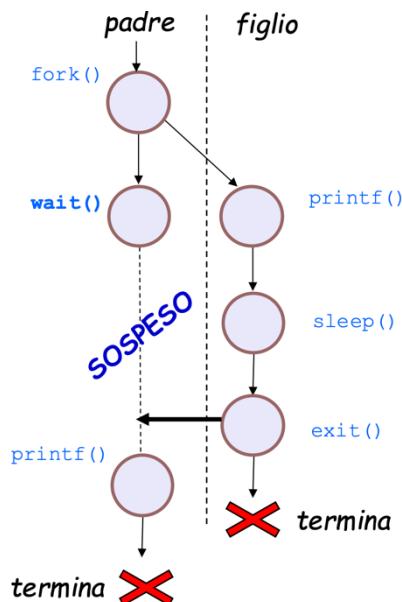
Il processo padre attende la terminazione del processo figlio, detta operazione di **join**, tramite la system call **wait**, mentre il figlio, tramite la chiamata ad **exit**, deposita un valore numerico (status) che indica l'esito della sua esecuzione (status = 0, l'esecuzione è avvenuta correttamente, status != 0, l'esecuzione ha presentato un'anomalia).



Quella appena introdotta, **wait**, è una system call che consente ad un processo padre di raccogliere lo stato di terminazione di un suo processo figlio (se specifica NULL come parametro ignora questa operazione); nel caso in cui nessun processo figlio fosse ancora terminato, il kernel sospende il processo padre. La variabile di stato in uscita conterrà il valore passato dal processo figlio alla system call **exit**. Se ci sono più processi figli per uno stesso padre, il processo si sblocca al primo figlio che termina (qualunque esso sia).

Un processo termina con la chiamata di sistema **exit** che, una volta invocata, passa uno stato di uscita numerico intero dal processo al kernel; tale valore è, poi, reso disponibile al processo padre tramite la chiamata di sistema **wait**. Normalmente, un processo che termina senza anomalia restituisce uno stato di uscita 0.

L'operazione di **wait** è importante perché garantisce che il processo padre termini sempre dopo il processo figlio, indipendentemente da cosa organizza lo scheduler del Sistema Operativo:



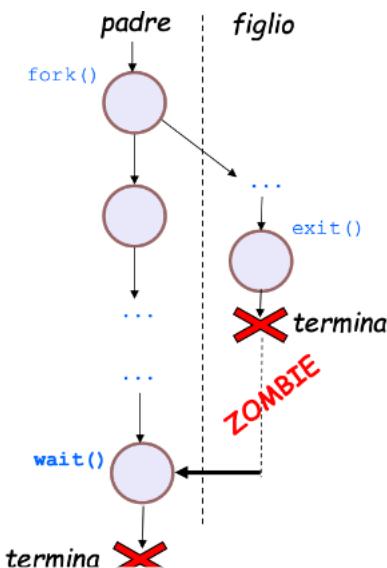
Il processo padre può anche utilizzare le seguenti macro:

- `WIFEXITED(status)`, restituisce **vero se il figlio è terminato volontariamente e ne ritorna lo stato di terminazione**;
- `WIFSIGNALED(status)`, restituisce **vero se il figlio è terminato involontariamente e ne ritorna il numero del segnale UNIX che ha causato la terminazione**.

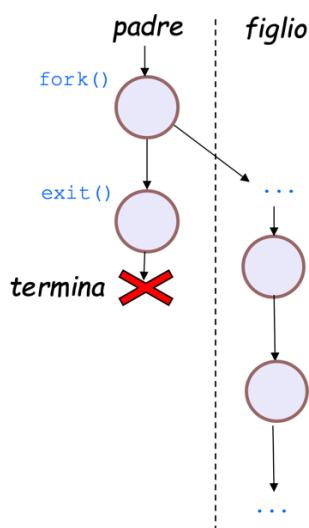
La system call `waitpid` è una variante della `wait` che consente al programmatore di indicare **il PID di uno specifico processo figlio di cui attendere la terminazione**:

```
int waitpid(pid_t pid, int *state, int options);
```

Si supponga che **il processo figlio termini prima che il padre si metta in attesa con la `wait`**; il processo figlio in questione viene detto **processo zombie** perché il kernel rilascia tutte le risorse **tranne il suo stato di terminazione**, in modo da dare la possibilità di ricongiungersi con il padre **in un secondo luogo**:



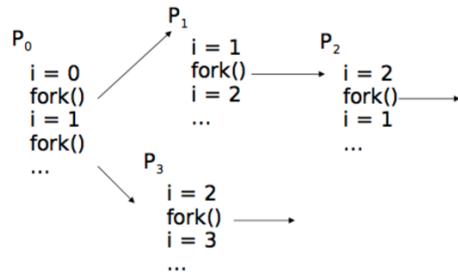
Un ulteriore caso notevole è quello dei **processi orfani**, che occorrono quando **il processo genitore termina prima dei suoi processi figli** (attivi o zombie); il kernel, in questo caso, assegna a tali processi **il valore 1 come parent ID**, cioè li fa diventare figli del processo `init` (che non termina mai) ma li lascia inconsapevoli della terminazione del padre.



Si supponga di voler **creare cinque processi**, il codice potrebbe essere qualcosa del tipo:

```
for (int i = 0; i < 5; i++) {
    pid = fork();
    if (pid == 0) {
        // Codice del processo figlio
    }
}
```

Ma osservando uno schema grafico che rappresenta la **biforcazione dei processi nel tempo**:



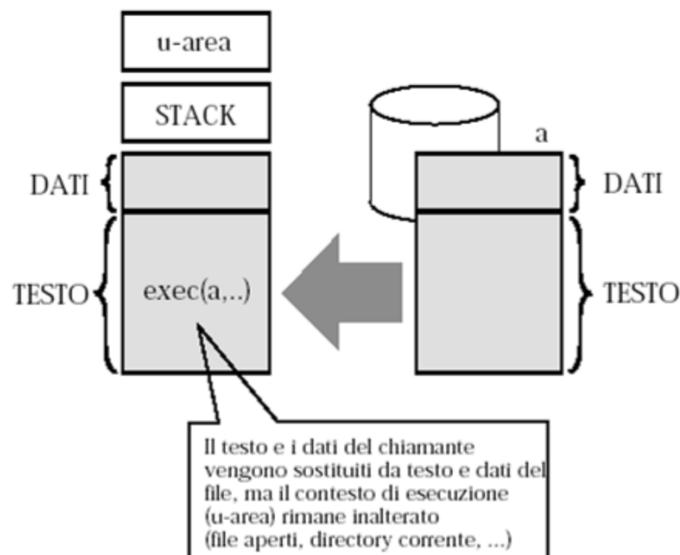
Si nota che **il numero totale di processi è**:

$$\sum_{i=0}^4 2^i = 31$$

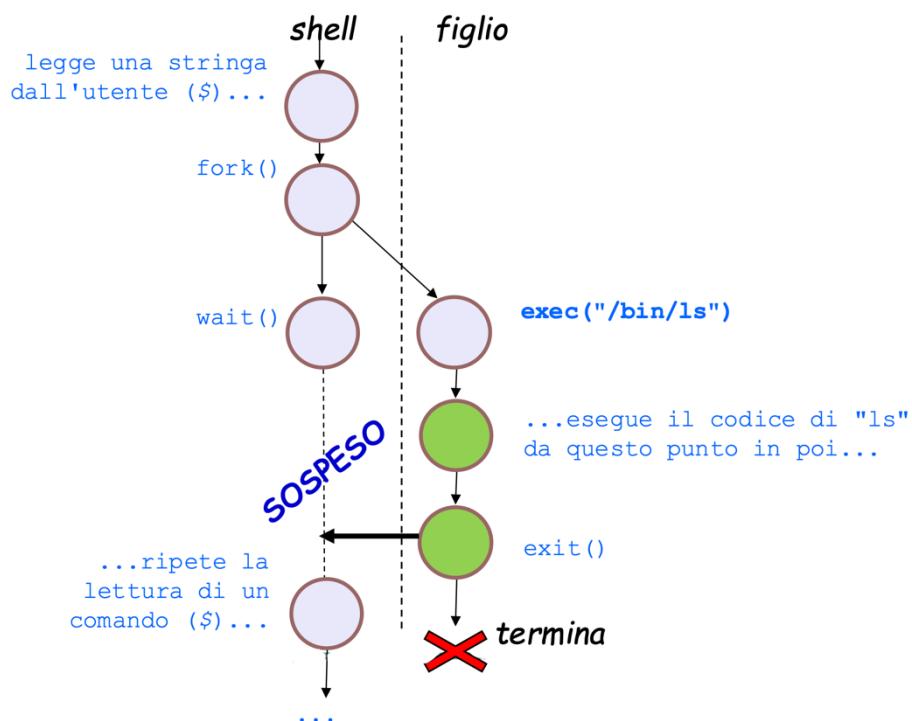
Che non è il numero di processi da cui si voleva partire. Il motivo di questo errore risiede nell'assenza di istruzioni di terminazione che lasciano i processi attivi quando inizia una nuova iterazione. La soluzione corretta è:

```
for (int i = 0; i < 5; i++) {
    pid = fork();
    if (pid == 0) {
        // Codice del processo figlio
        exit(0);
    }
}
```

fork permette di creare nuovi processi che eseguono lo stesso programma del chiamante ma, come è facile intuire, **da sola questa system call non consente di eseguire altri programmi**; per farlo, **occorre che un processo (ad esempio, la shell) crei un nuovo processo tramite fork e che questo esegua il programma desiderato chiamando la system call exec**, per la quale **il controllo passa al nuovo programma senza mai tornare a quello chiamante** (eccetto in casi di errore). Si annoti che **il nuovo programma viene eseguito nel contesto del processo che chiama exec**, cioè **il PID non cambia**.



Dopo aver chiamato `exec`, **il processo mantiene lo stesso process control block, la stessa user area** (a parte PC e informazioni legate al programma) **e lo stack nel kernel** ma ha codice, dati globali, **stack e heap nuovi**, nonché riferisce una **nuova area codice (text)**.



```
pid = fork();  
  
if (pid == 0) {  
  
    // codice figlio  
    ...  
    if (execvp("program", ...) < 0){  
        perror("exec fallita");  
        exit(1);  
    }  
  
    // il processo figlio non  
    // esegue mai in questo punto  
}  
// Il padre continua da questo  
// punto in poi...
```

Si annoti che, **quando si avvia un comando, la shell ne attende la terminazione con wait e fino ad allora non è possibile digitare altri comandi.**

Come è stato possibile intuire, **in Linux non esiste un solo comando exec:**

- Percorso completo dell'eseguibile, parametri tramite lista:

```
int execl(char *path, char *arg0, char *arg1, ..., (char*)0);
```

- Nome dell'eseguibile (da cercare nelle cartelle di sistema), parametri tramite lista:

```
int execlp(char *nomefile, char *arg0, char *arg1, ..., (char*)0);
```

- Percorso completo dell'eseguibile, parametri tramite array:

```
int exev(const char *path, char *const argv[]);
```

- Nome dell'eseguibile (da cercare nelle cartelle di sistema), parametri tramite array:

```
int execvp(const char *nomefile, char *const argv[]);
```

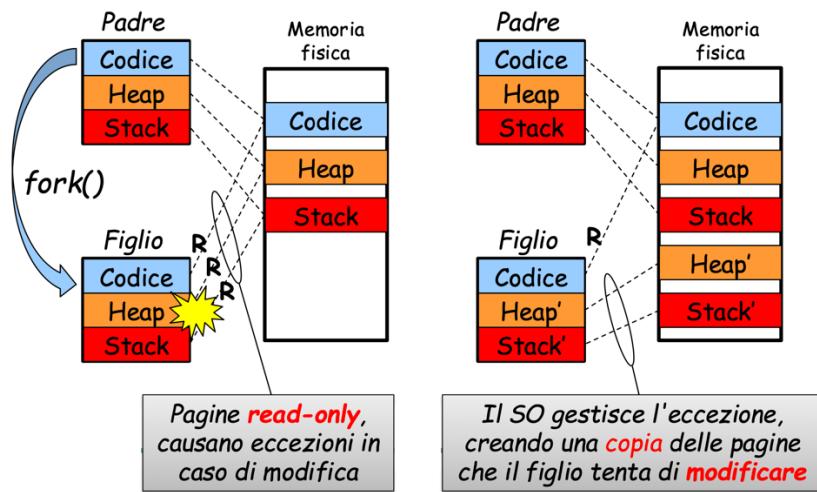
In UNIX, poi, è convenzione che il primo parametro coincida con il nome del programma. Un programma riceve i valori di ingresso dalla exec tramite argc e argv nel main: il primo consiste nel numero di parametri di ingresso e il secondo in un vettore di stringhe. Nella shell:

```
gcc sorgente.c -o eseguibile
```

Il primo token è il nome del programma da eseguire. La shell, interprete dei comandi, estrae i token dalla linea di comando (un token è una parola separata da spazi), crea un nuovo processo (con fork), esegue il programma e gli passa i parametri (exec); attende, poi, che il figlio appena creato termini e mostra di nuovo il prompt dei comandi per attendere un nuovo user input.

Nel 99% dei casi, **dopo una fork viene eseguita una exec e l'operazione di copia della memoria tra padre e figlio è nella maggior parte dei casi sprecata**, con un overhead consistente. Alcuni Sistemi Operativi (come Windows), per risolvere questo problema, combinano fork ed exec in un'unica system call, chiamata **copy – on – write**: il figlio inizialmente condivide

la memoria con il padre in modalità read – only, finché non modifica una porzione di questa memoria; a tal punto, il kernel crea una nuova copia di quella sola porzione di memoria nel processo figlio.



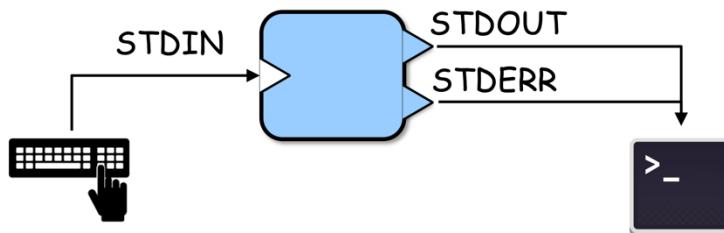
Il fatto che **non sia diventato uno standard l'unione di `fork` ed `exec`** è dovuto al fatto che **la loro separazione permette di configurare il processo figlio prima di eseguire un programma**, potendo eseguire **operazioni di gestione** (come la riduzione dei permessi o la chiusura di risorse).

LA SHELL LINUX

Il principio fondante di UNIX è di collegare tanti piccoli programmi per creare workflow complessi, sulla base dell'assunto per cui **ogni comando ha in ingresso un flusso di caratteri e produce in uscita un altro flusso di caratteri**.

Di default, **ogni processo ha accesso ad almeno tre canali di I/O**:

- STDIN (Standard Input, 0), per la lettura dalla tastiera;
- STDOUT (Standard Output, 1) per la scrittura a video;
- STDERR (Standard Error, 2) per la scrittura a video.



La tabella dei file aperti abbina un numero, detto handle o descrittore, **ad ogni canale/file acceduto dal processo**; ad esempio:

File Handle Table del processo (nel kernel)

Handle	File
0	<tastiera>
1	<video terminale>
2	<video terminale>
3	/home/user/hello.txt
4	...

Un processo può leggere/scrivere su tastiera e video come se scrivesse un normale file su disco, utilizzando le stesse system call e non dovendosi preoccupare del luogo da/in cui questi dati verranno presi/messi.

A primo impatto, **STDOUT e STDERR possono sembrare uguali**; tuttavia, **hanno utilizzi diversi e i messaggi prodotti possono essere rediretti verso due destinazioni diverse**:

- **STDOUT** è usato per leggere i normali messaggi del programma;
- **STDERR** è usato per eventuali messaggi di errore.

In linguaggio C, i canali in questione sono acceduti, rispettivamente, da:

- `scanf`, per STDIN;
- `printf`, per STDOUT;
- `perror o exit`, per STDERR.

Si annoti che **in UNIX, per convenzione, in caso di errori un comando esca con un codice numerico diverso da zero**.

Nel caso in cui si volesse **redirigere il flusso di dati da un programma verso un file**, è possibile **usare il simbolo > dopo il comando**:

```
ls > cartelle.txt
```

In questo esempio, l'output non appare a schermo ma è salvato sul file `cartelle.txt`. È anche possibile **silenziare un comando tramite la redirezione**, semplicemente **puntando al file /dev/null**, un file fittizio messo a disposizione da UNIX:

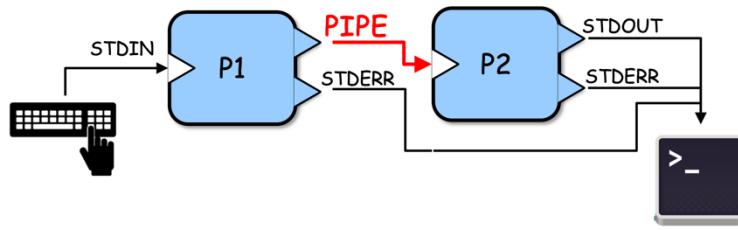
```
ls > /dev/null
```

Al contrario, il simbolo **< redirige i dati in ingresso**, ovvero da file piuttosto che da tastiera:

```
./programma < input.txt
```

Specificando, invece, **un numero prima del simbolo >** è indicato **il canale verso cui redirigere**; ad esempio **1>** è **STDOUT** (di default), mentre **2>** è **STDERR**. La shell effettua la redirezione dopo la **fork e prima di exec**.

Il meccanismo di UNIX con cui è **possibile collegare più programmi** prende il nome di **pipe** e permette di **redirigere i canali di I/O verso un altro processo**, potendo quindi **combinare più comandi**:



Praticamente, la concatenazione in pipe avviene con il simbolo | ; ad esempio:

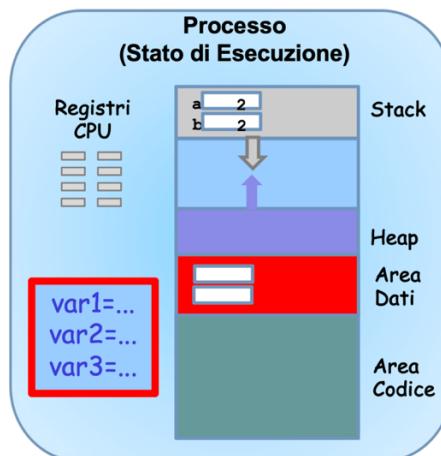
`ls /bin | grep "yes"`

Il comando grep si occupa di leggere le righe dallo STDIN e di stampare sullo STDOUT le sole righe contenenti la sottocorrispondenza specificata; nell'esempio, la sottocorrispondenza specificata è “yes” ma lo STDIN è lo STDOUT di ls. Questo comando è solo uno di quelli che vengono chiamati comandi di Log Parsing:

 GREP	GREP allows you to search patterns in files. ZGREP for GZIP files. \$grep <pattern> file.log	-n: Number of lines that matches -i: Case insensitive -v: Invert matches -E: Extended regex -C: Count number of matches -f: Find filenames that matches the pattern	 SORT	SORT is used to sort a file. \$sort foot.txt	-o: Output to file -r: Reverse order -n: Numerical sort -k: Sort by column -C: Check if ordered -U: Sort and remove -f: Ignore case -h: Human sort
 NGREP	NGREP is used for analyzing network packets. \$ngrep -I filepcap	-d: Specify network interface -i: Case insensitive -x: Print in alternate hexdump -t: Print timestamp -I: Read pcap file	 UNIQ	UNIQ is used to extract uniq occurrences. \$uniq foot.txt	-c: Count the number of duplicates -d: Print duplicates -i: Case insensitive
 CUT	The CUT command is used to parse fields from delimited logs. \$cut -d ":" -f 2 filelog	-d: Use the field delimiter -f: The field numbers -C: Specifies characters position	 DIFF	DIFF is used to display differences in files by comparing line by line. \$diff foolog barlog	How to read output? a: Add #: Line numbers C: Change <: File 1 d: Delete >: File 2
 SED	SED (Stream Editor) is used to replace strings in a file. \$sed s/regex/replace/g	S: Search -e: Execute command g: Replace -n: Suppress output d: Delete W: Append to file	 AWK	AWK is a programming language use to manipulate data. \$awk '{print \$2}' foolog	Print first column with separator ":" \$awk -F ":"{print \$1}' /etc/passwd Extract uniq value from two files: awk 'FNR==NR {a[\$0]++; next} K40 in a' f1.txt f2.txt

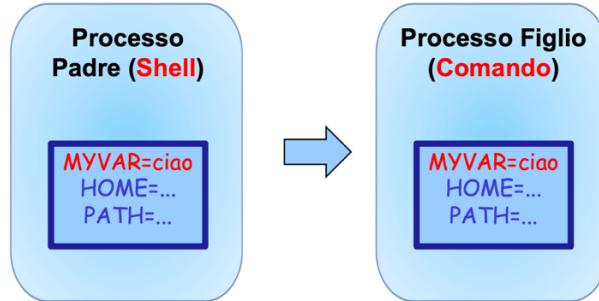
Senza entrare nel dettaglio, **tutti i comandi UNIX hanno una pagina di manuale**, con la lista di tutti i parametri di ingresso, e può essere **ottenuta digitando sulla shell man nomeComando**; in realtà, la maggior parte dei comandi, se richiamati con la sola opzione **-h** o **--help**, permettono di mostrare una sintesi di quali sono e cosa fanno i parametri possibili. Altre risorse per l'apprendimento dei comandi per la shell UNIX sono: cheat.sh, explainshell.com, tldr pages e The UNIX Game.

UNIX consente di memorizzare in un processo delle variabili d'ambiente (come variabili di configurazione):



Ad esempio, la variabile **PATH** indica alla shell le cartelle di sistema in cui cercare i programmi eseguibili (come elenco separato con :). Tipicamente, le variabili d'ambiente sono impostate dall'utente nella shell e poi copiate nei processi avviati dalla shell stessa:

```
export MYVAR=ciao
```



È possibile usare la shell anche per creare dei piccoli programmi, detti script, intesi come un file di testo con una lista di comandi che viene interpretato dalla shell stessa riga per riga come se fosse un insieme di comandi digitati a mano. Uno script di questo tipo può essere eseguito tramite il comando **bash**, seguito dal nome del file:

```
bash mioscript.sh
```

Ma in realtà può essere anche avviato con le stesse modalità con cui si avvia un classico programma (`./mioscript.sh` in shell); tuttavia, in questo caso sarebbero necessari i permessi di esecuzione ed iniziare lo script con la seguente riga:

```
#!/bin/bash
```

La riga in questione, in fase di esecuzione, viene interpretata come **/bin/bash mioscript.sh**, permettendo l'esecuzione convenzionale dello script mascherata da programma eseguibile. La shell fornisce anche un linguaggio completo, con variabili, condizioni e cicli; ad esempio, il carattere \$ è usato per leggere variabili:

```
VAR=abc
echo "Il valore è"; echo $VAR
```

In shell:

```
Il valore è
abc
```

Il modo con cui le variabili sono dichiarate è del tutto simile a quello con cui si gestiscono le variabili d'ambiente, con la sola differenza della keyword `export` esplicitata perché si vuole inserire la variabile in argomento tra le variabili d'ambiente del processo e trasmetterla ai figli. È anche possibile interpolare:

```
VAR=abc
echo "Il valore è $VAR"
```

In shell:

Il valore è abc

Il linguaggio di scripting fornito dalla shell è opportuno solo per programmi molto semplici e deve essere sostituito con linguaggi più solidi e strutturati, come Python, per esigenze più articolate; il motivo risiede nell'attenzione che ci vuole per maneggiare diversi costrutti in linguaggio shell, come:

- Spazi assenti prima e dopo il simbolo =:

```
VAR="Ciao"
```

- Spazi necessari prima e dopo le parentesi quadre:

```
if [ "$VAR" == "" ]
then
    echo "La variabile è vuota"
else
    echo "La variabile non è vuota"
fi
```

- Quoting necessario delle variabili e delle stringhe se c'è la possibilità che siano vuote o che contengano spazi, altrimenti possono incorrere errori sintattici:

```
"$VAR"
```

La shell aggiorna automaticamente una variabile speciale, \$?, ogni volta che viene eseguito un comando; pertanto, se \$? è diversa da 0, allora si è verificato un qualche errore nell'esecuzione del comando (il confronto fra interi si effettua con gli operatori -ne, -eq, -gt):

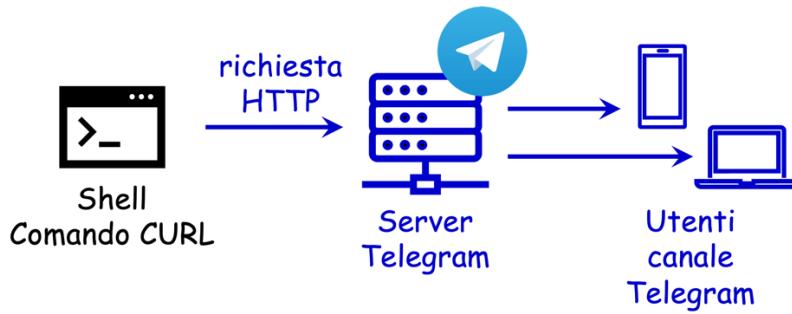
```
if [ $? -ne 0 ]
then
    echo "Download fallito"
else
    echo "Download riuscito"
fi
```

Mentre, invece, la sintassi \$ (...), con un comando all'interno delle parentesi, permette di lanciare un comando esterno il cui output può essere registrato in una variabile:

```
NAME=$(whoami)
echo "Il mio username è $NAME"
```

Mostrando l'username dell'utente che ha eseguito il comando.

È anche permesso di inviare richieste http verso servizi remoti, come il cloud, potendo potenzialmente anche mandare un messaggio con un bot Telegram:

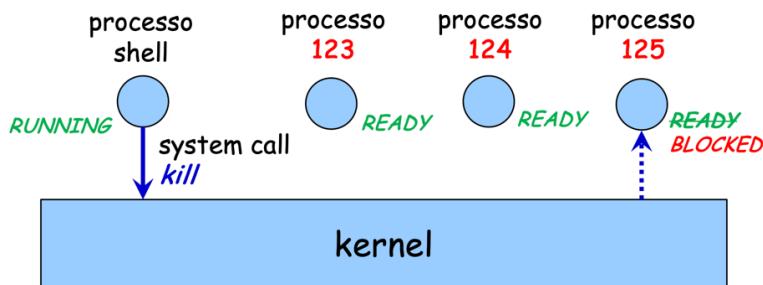


Tipo di richiesta	Percorso da accedere (sendMessage)
HEADER	POST /bot123456:xxxxxxxxxxxxxxxxxxxx/sendMessage HTTP/2 Host: api.telegram.org User-Agent: curl/8.1.2 Server da accedere Accept: */* Content-Type: multipart/form-data; ... Content-Length: name="chat_id" -123456789 ... name="text" il messaggio

Anche se ci sono spesso script open – source per command – line client (CLI) verso servizi cloud che svolgono le stesse operazioni.

La shell attende la terminazione dei comandi usando `wait()`, durante la quale il cursore dei comandi non è disponibile per esecuzione di task in foreground; infatti, come già detto in precedenza, la shell può eseguire un comando alla volta e, per eseguire una seconda task, è necessario attendere la fine dell'esecuzione di quelle precedenti. In realtà, è possibile evitare l'attesa dell'esecuzione specificando il carattere `&`, rendendo il cursore subito disponibile ed eseguendo in background il task precedentemente avviato.

Per controllare lo stato di un processo, la shell si affida al meccanismo dei segnali:



Ci sono due shortcut da tastiera che permettono di interagire con i processi in foreground in maniera immediata:

- CTRL + C, termina il processo in foreground nella shell;
- CTRL + Z, sospende (può essere riattivato) il processo in foreground nella shell.

Per una comunicazione generica con i processi di sistema, nella shell è possibile usare il comando `kill`, accompagnato dal tipo di segnale da inviare (acronimo o codice numerico) e dal PID del processo da segnalare:

```
kill -INT 123
```

È stato mandato un segnale di interruzione (equivalente di **CTRL + C**) **al processo con PID 1234**. Esistono **numerosi tipi di segnali**, ognuno dei quali **innesca uno spettro ampio di azioni** (configurate nel programma) da parte del processo destinatario o dal Sistema Operativo; ad esempio:



Signal name	Signal num.	Description
SIGHUP	1	Hang-up detected on controlling terminal, or death of controlling process
SIGINT	2	Issued if the user sends an interrupt signal (Ctrl+C)
SIGQUIT	3	Issues if the user sends a quit signal (Ctrl+D), core dump
SIGKILL	9	If a process gets this signal, it must quit immediately and will not perform any clean-up operations
SIGSEGV	11	Issued when a memory access violation occurs
SIGALRM	14	Alarm clock signal (used for timers)
SIGTERM	15	Software termination signal (sent by kill by default)

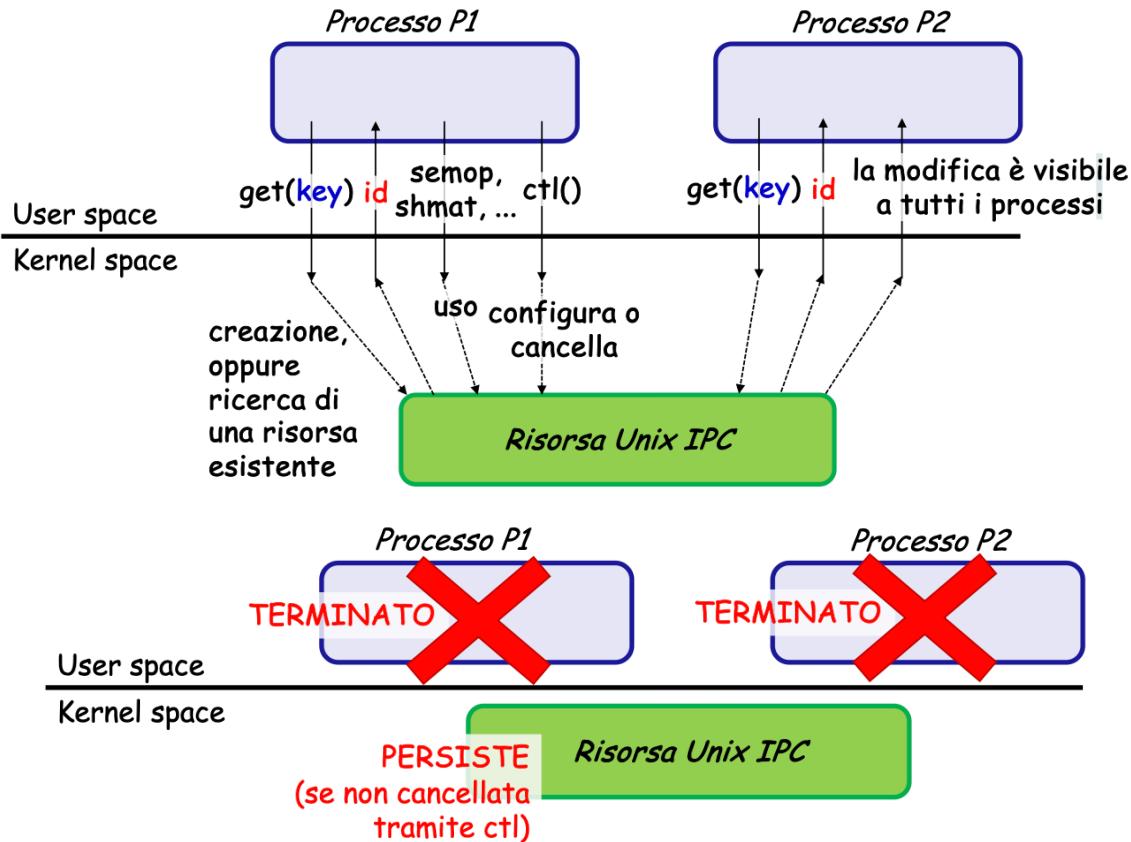
Si annoti che i **signal handler vanno usati solo per operazioni semplici** (come clean – up di risorse, riconfigurazione, riportare lo stato, abilitare il debugging, ...), potendo **soffrire di race condition e non essendo indicati per la sincronizzazione**.

SHARED MEMORY E SEMAFORI

Linux/UNIX permette la comunicazione tra processi mediante primitive e strutture dati fornite dal kernel stesso; per oggetto (o risorsa) IPC in UNIX si intende una struttura dati contenente:

- **Memoria condivisa** (SHM, shared memory segments);
- **Semafori** (SEM, semaphore arrays);
- **Code di messaggi** (MSG, queues).

Ogni IPC è gestita da due primitive, **get** e **ctl**: la prima utilizza una chiave (detta IPC key) ed opportuni parametri per restituire al processo un descrittore della risorsa, mentre il secondo permette, dato un descrittore, di verificare lo stato di una risorsa, cambiare lo stato di una risorsa o rimuovere una risorsa.



Le risorse IPC sono permanenti, se un processo esce **non sono eliminate automaticamente** ma è necessaria una **chiamata esplicita alla chiamata `ctl`**.

Per quanto riguarda l'utilizzo, la primitiva `get`:

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

Con:

- `key`, chiave dell'oggetto:
 - Valore intero arbitrario “cablato”;
 - Valore generato da `ftok()`;
 - Costante `IPC_PRIVATE`;
- `flag`, indica la modalità di acquisizione della risorsa:
 - Una o più costanti passate insieme in OR;
 - 0, utilizza la risorsa se già esistente altrimenti da errore;
 - `IPC_CREAT`, crea una nuova risorsa se non ne esiste già una con la chiave indicata (se già esiste, viene riutilizzata, rendendo ininfluente il flag);
 - `IPC_EXCL`, da usare in combinazione con `IPC_CREAT` e permette di gestire i due casi di risorsa già esistente e risorsa appena creata (utile per gestire il valore iniziale della risorsa);
 - Permessi di accesso (ad esempio, 0644).

Per quanto riguarda l'utilizzo, la primitiva `ctl`:

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

Con:

- desc, il descrittore della risorsa;
- cmd, il comando da eseguire:
 - IPC_RMID, rimozione della risorsa;
 - IPC_STAT, richiede informazioni sulla risorsa;
 - IPC_SET, richiede la modifica di attributi della risorsa (ad esempio, i permessi di accesso).

Ogni risorsa IPC è identificata da un valore univoco nel sistema, denominato **chiave** o **IPC key**; il modo più semplice di scegliere una chiave consiste nell'uso di **un valore a piacere del programmatore, “cablato” nel programma**. Il problema di questo approccio risiede nel **rischio che più sviluppatori di programmi diversi possano scegliere**, per caso, **lo stesso valore**; un approccio sicuramente più sicuro può essere **l'impiego della funzione ftok**, che genera automaticamente **una chiave sulla base** dei parametri di ingresso, cioè **del percorso di un file/cartella appartenente al programma e di un carattere scelto a piacere**:

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

Questa funzione, però, è **deterministica**; infatti, se **due processi** (ad esempio, di uno stesso programma) **usano gli stessi parametri**, ottengono le stesse chiavi. Tuttavia, ciò rappresenta un **problema di cardinalità certamente inferiore rispetto al caso precedente** perché **programmi diversi useranno certamente percorsi diversi**, evitando il problema di una possibile collisione tra chiavi almeno tra processi di programmi diversi.

IPC_PRIVATE (equivale a 0) è **un valore costante che può essere usato per creare una risorsa senza chiave**, semplificandone la gestione ma introducendo il problema dell'impossibilità di accesso da parte di altri processi e in altre esecuzioni. L'unico modo per condividere questo tipo di risorsa è tramite il meccanismo di **fork**, ovvero solo con processi correlati:

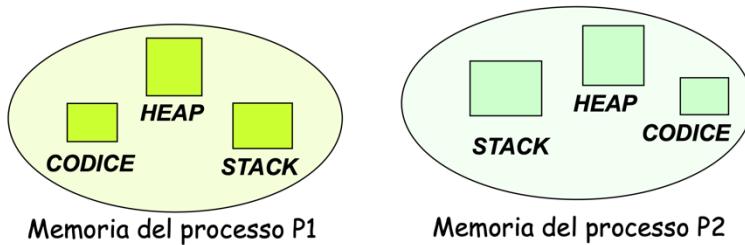
```
key_t mykey = IPC_PRIVATE;
int id = ....get(mykey, ...);

pid = fork();
if(pid == 0) {
    /* figlio, utilizza il descrittore "id" */
}
else if(pid > 0) {
    /* padre, utilizza anche lui "id" */
}
```

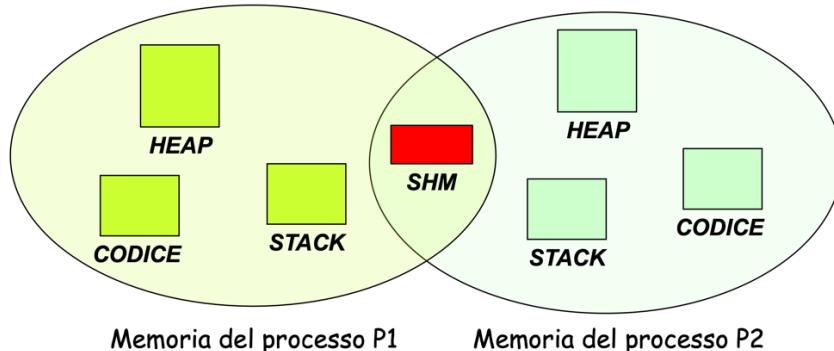
Nella shell, è possibile usare i seguenti comandi:

- ipcs <-m|-s|-q>, per visualizzare le risorse IPC allocate nel sistema, mostrandone l'identificatore e l'utente proprietario;
- ipcrm <shm|sem|msg> <IPC_ID>, per rimuovere una risorsa IPC ed utilizzabile qualora i processi non abbiano rimosso le risorse (ad esempio, in caso di terminazione anomala).

I processi UNIX, a differenza dei threads, **non condividono alcuna area di memoria**; ciò vale anche per i processi parenti: **il figlio ottiene una copia dei dati del padre ma ogni processo modifica la propria copia**.



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



L'utilizzo di una SHM passa per le seguenti operazioni:

- Creazione

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

Con **size** la dimensione in byte della memoria condivisa. Questa primitiva restituisce un identificatore numerico (il descrittore) in caso di successo, oppure -1 in caso di fallimento. Un esempio concreto è il seguente, con il quale si crea una nuova shared memory (nel caso in cui non esista già, altrimenti si riutilizzerebbe il segmento esistente) con solo utente e gruppo proprietari abilitati all'accesso RW:

```
int ds_shm = shmget(40, 1024, IPC_CREAT | 0664);
if (ds_shm < 0) {
// Qualcosa è andato storto
    perror("Errore shmget");
    exit(1);
}
// Usa la shared memory
...
```

Utilizzando sia **IPC_CREAT** che **IPC_EXCL**, si consente di gestire separatamente i due casi di risorsa già esistente ($ds_shm < 0$) e di risorsa appena creata ($ds_shm \geq 0$); in caso di risorsa già esistente, e quindi $ds_shm < 0$ negativo, occorre chiamare una seconda volta **shmget** ma senza usare **IPC_CREAT** e **IPC_EXCL**.

- Collegamento verso

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

Collega il segmento di memoria allo spazio di indirizzamento del chiamante, con shmid identificatore del segmento di memoria, e restituendo un puntatore all'area di memoria collegata (o -1 in caso di fallimento). **Gli altri sono parametri opzionali**, possono essere o **NULL** o **0** e sono:

- shmidaddr, indirizzo al quale collegare il segmento di memoria condivisa (se NULL, l'indirizzo viene automaticamente scelto dal Sistema Operativo);
- flag, 0 per lettura/scrittura, **IPC_RONLY** per collegare in sola lettura.
- Uso

Per leggere e scrivere una memoria condivisa non è richiesto di usare chiamate di sistema, l'area può essere acceduta come una qualsiasi variabile dello spazio di indirizzamento del processo; ad esempio:

```
mystruct * p = shmat(ds_shm, NULL, 0);
p->variabile = 123; // la modifica è visibile
                     // anche ad altri processi
```

Per quanto riguarda le operazioni di controllo, invece, la primitiva da usare è la seguente:

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

Esegue un comando su una SHM, con ds_shm il descrittore della memoria condivisa su cui si vuole operare, cmd la specifica del comando da eseguire e buff un puntatore ad una struttura di tipo shmid_ds (parametro di ingresso nel caso di comando **IPC_SET** e parametro di uscita nel caso di comando **IPC_STAT**), definita nell'header <sys/shm.h>. Ad esempio:

```
shmctl(ds_shm, IPC_RMID, NULL);
```

I comandi possono essere:

- **IPC_STAT**, ottiene le informazioni sul segmento di memoria condivisa;
- **IPC_SET**, modifica i campi del segmento di memoria condivisa (come i permessi di accesso);
- **IPC_RMID**, marca da eliminare e rimuove solo quando non vi sono più processi attaccati;
- **SHM_LOC**, impedisce che il segmento venga swappato o paginato.

shmctl restituisce -1 in caso di errore.

- Scollegamento
- Eliminazione

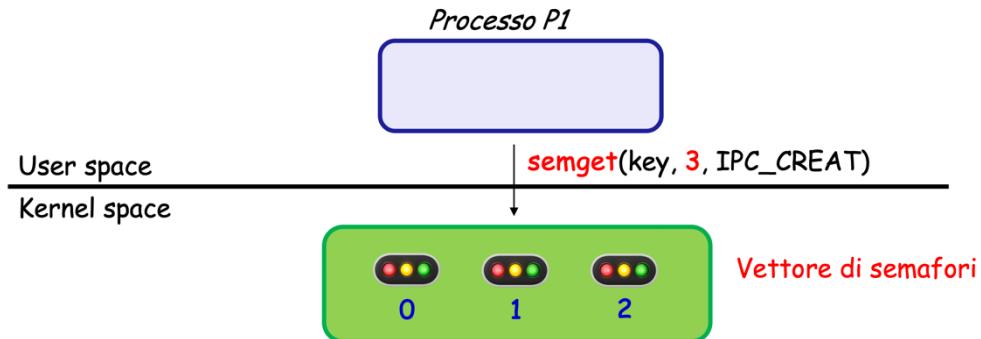
L'eliminazione della memoria condivisa non avviene subito ma solo quando nessun processo vi è più collegato.

Tutto quanto detto finora è definito negli header file seguenti:

```
<sys/types.h>
<sys/ipc.h>
<sys/shm.h>
```

In UNIX, la creazione e l'inizializzazione di semafori avviene tramite le seguenti primitive:

```
int semget(key_t key, int nsems, int semflg);
int semctl(int semid, int semnum, int cmd, ...);
```



Con la prima, si va a **creare un vettore di *nsems* semafori**, mentre con la seconda è possibile **effettuare operazioni sui singoli semafori di tale vettore**:

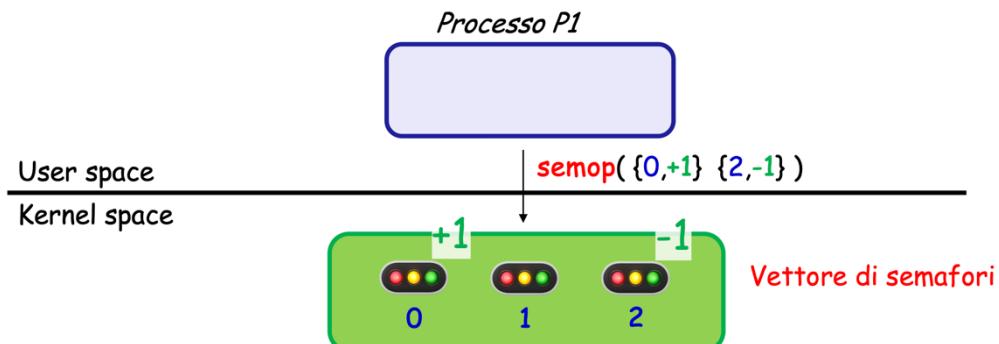
```
sem = semget(IPC_PRIVATE, 2, IPC_CREAT|0664);
semctl(sem, 0, SETVAL, 1); // Accede al semaforo in posizione 0
semctl(sem, 1, SETVAL, 5); // Accede al semaforo in posizione 1
```

Ogni semaforo del vettore è rappresentato internamente nel kernel da una struttura dati, che comprende tra i suoi campi i seguenti:

```
unsigned short semval; /* valore semaforo*/
unsigned short semzcnt; /* # proc che aspettano 0 */
unsigned short semncnt; /* # proc che aspettano incr.*/
```

La primitiva ***semop*** effettua un'operazione (o un gruppo di operazioni) su un vettore di semafori:

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



La primitiva in questione prende in ingresso un vettore di operazioni **sops* di dimensione *nsops*, ognuna delle quali rappresentata da una struttura *sembuf*:

```
struct sembuf {
    unsigned short sem_num; /* numero di semaforo */
    short sem_op;           /* operazione da compiere */
    short sem_flg;          /* flags (IPC_NOWAIT, SEM_UNDO) */
};
```

Un'implementazione di una wait e di una signal sono, rispettivamente:

```
void Wait_Sem (int id_sem, int numsem)  {
    struct sembuf sem_buf;
    sem_buf.sem_num = numsem;
    sem_buf.sem_op = -1; // Decrementa di -1
    sem_buf.sem_flg = 0;
    semop(id_sem, &sem_buf, 1);
}
void Signal_Sem (int id_sem, int numsem)  {
    struct sembuf sem_buf;
    sem_buf.sem_num = numsem;
    sem_buf.sem_op = +1; // Incrementa di +1
    sem_buf.sem_flg = 0;
    semop(id_sem, &sem_buf, 1);
}
```

Si noti come il terzo parametro di `semop` è sempre 1, eseguendo una sola operazione alla volta. Le operazioni nel vettore in questione sono eseguite in maniera atomica (senza interruzioni, in modo da esser eseguite consistentemente e che nessun'altro processo interferisca durante l'esecuzione).

La primitiva si sospende (o ritorna un errore) se qualcuna delle operazioni non può essere svolta (ad esempio, il decremento di un semaforo già negativo); quando tutte le operazioni sono possibili, invece, avvengono simultaneamente (in mutua esclusione con altre `semop`), mentre ogni operazione è eseguita sul semaforo indicato dalla variabile `sem_num`.

Tramite `sem_op`, è possibile indicare tre tipi di operazioni su un semaforo:

- $\text{sem_op} < 0 \Rightarrow \text{wait}$

L'operazione si articola in due modi:

- $\text{semval} \geq \text{abs}(\text{sem_op})$, allora l'operazione procede immediatamente, senza sospendere il processo, e il valore del semaforo sarà modificato come:

```
semval -= abs(sem_op)
```

- $\text{semval} < \text{abs}(\text{sem_op})$, allora il processo si sospende finché non si verifica la prima condizione;

- $\text{sem_op} > 0 \Rightarrow \text{signal}$

L'operazione consiste nell'addizionare il valore di `sem_op` al valore del semaforo (`semval`):

```
semval += sem_op
```

Questa operazione non causa in alcun modo il blocco del processo.

- `sem_op == 0` \Rightarrow `wait_for_zero`

L'operazione specificata è articolata nei seguenti passi:

- `semval == 0`, allora l'operazione procede immediatamente e il processo non si sospende;
- `semval != 0`, allora il processo si sospende finché non si verifica la prima condizione.

Per la rimozione di una struttura dati semaforo si ricorre alla seguente primitiva:

```
semctl(id_sem, num_sem, IPC_RMID);
```

Con `num_sem` che, in questo caso, viene ignorata (ad esempio, può essere posta a zero).

Tutto quanto detto finora è definito negli header file seguenti:

```
<sys/types.h>
<sys/ipc.h>
<sys/sem.h>
```

POSIX THREADS

Diverse librerie proprietarie per la gestione dei threads sono state implementate da diversi produttori di hardware, causando una forte eterogeneità tra le varie librerie e rendendo necessaria un'interfaccia standard. Per i sistemi UNIX, tale interfaccia fu specificata nel 1995 con lo standard IEEE POSIX 1003.1c, detto anche PThreads. I PThreads sono stati definiti come una serie di tipi e procedure implementati in C e composti da un file `pthread.h` e una libreria dinamica (parametro `-pthread`):

```
#include <pthread.h> // Per includere la libreria
gcc -c main -o main.o
gcc -c procedure.c -o procedure.o
gcc -pthread main.o procedure.o -o eseguibile
```

Ma perché hanno riscosso così tanto successo? Principalmente, per il **guadagno in termini di prestazioni e la comunicazione inter – thread**, molto più efficiente e semplice da usare rispetto al caso inter – processo.

Platform	<code>fork()</code>	<code>pthread_create()</code>
IBM 375 MHz POWER3	61.94	7.46
IBM 1.5 GHz POWER4	44.08	1.49
IBM 1.9 GHz POWER5 p5-575	50.66	1.13
INTEL 2.4 GHz Xeon	23.81	1.70
INTEL 1.4 GHz Itanium 2	23.61	2.10

50000 creazioni di processi/thread;
i tempi sono riportati in secondi

Le API per l'utilizzo di PThread si dividono in:

- **Gestione dei Thread**, in particolare **creazione, distruzione e join** di threads

Per la creazione di un thread:

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

Crea un nuovo thread e lo mette in esecuzione, con:

- **id** (output), tipo `pthread_t *`, è un identificatore del thread creato;
- **attr** (input), tipo `pthread_attr_t`, imposta gli attributi del thread;
- **start_routine** (input), è un puntatore alla funzione che verrà eseguita dal nuovo thread, da definire come `void * nome_funzione (void *)` il cui nome andrà passato senza parentesi nel parametro;
- **arg** (input), è un puntatore passato come parametro di ingresso alla starting routine (ne sarà fatto casting a `void *`).

Per la terminazione di un thread, essa può avvenire per diversi motivi:

- La starting routine termina la sua esecuzione;
- Il thread chiama la `ptrhead_exit()`;
- Il thread è cancellato da un altro thread con `pthread_cancel()`;
- L'intero processo termina.

Con:

```
pthread_exit(status)
```

Usata per **terminare esplicitamente un thread** ed è **buona norma usarla in tutti i thread**, con **status** variabile di input che indica lo stato di uscita del thread. Se usata nel programma principale (che potrebbe terminare prima di tutti i thread) gli altri thread continuerebbero ad eseguire.

La `pthread_create` può passare un singolo argomento di tipo `void *`; per passarne più di uno, occorre definire una struct con gli argomenti in questione e, nel thread padre, allocare sullo heap (`malloc`) un'istanza di questa struct, passandone il puntatore al thread figlio.

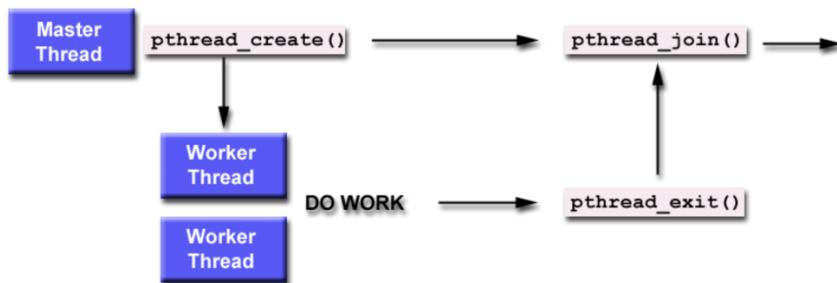
```
struct dati {  
    int dato1;  
    char dato2;  
};  
  
struct dati *d = (struct dati *)malloc(sizeof(struct dati));  
d->dato1=10;  
d->dato2='c';  
pthread_create(&id, NULL, start_func, (void *) d);
```

Il puntatore `struct dati *` viene convertito in puntatore `void *`, è un tipo di puntatore generico, non deferenziabile. Per usare correttamente la struttura dati nel thread figlio, poi, occorre fare il casting inverso del puntatore `void *`:

```
void * start_func( void * p ) {
    struct dati* dati = (struct dati *) p;
    ...
    printf("dato1=%d, dato2=%c", dati->dato1, dati->dato2);
}
```

È importante **utilizzare l'area heap per condividere dati fra threads**; sarebbe **scorretto utilizzare l'area stack**, invece, visto che è dedicata a variabili automatiche e private del thread.

L'operazione di **join** permette di sincronizzare un thread padre con uno o più thread figli. La chiamata `pthread_join(threadId, status)` blocca il chiamante finché il thread `threadId` specificato non termina:



`pthread_join` pone il thread padre in attesa della terminazione del figlio e con `status NULL` viene rinunciata dal padre la ricezione dei dati di uscita dal figlio; tuttavia, affinché si possa effettuare l'operazione di join su un thread, questo deve essere dichiarato come **joinable** attraverso gli attributi passati per parametro alla `pthread_create`. Inoltre, la `pthread_join` consente anche di ricavare lo stato di uscita del thread, `status`, passato dalla `pthread_exit` tramite un puntatore.

Il **thread figlio** può passare dei dati in uscita al **thread padre**, usando la stessa tecnica dei **parametri di ingresso** (ovvero, tramite una struct sull'area heap, non sulla stack, visto che questa viene distrutta all'uscita del thread).

```
struct dati_uscita {
    int risultato;
};

void* figlio(void*) {
    struct dati_uscita * status;
    pthread_join(..., &status);
    // puntatore-di-puntatore
    int ris = status->risultato;

    struct dati_uscita * status = malloc(...);
    status->risultato=...;

    pthread_exit(status);
}
```

- **Gestione dei Mutex**, in particolare creazione, distruzione, lock e unlock di variabili di mutua esclusione (mutex) per la gestione di sezioni critiche

`pthread_mutex_t` è una struttura dati che rappresenta un oggetto – mutex; non è dato sapere il contenuto grazie al principio di encapsulazione. `pthread_mutex_init(mutex, attr)` crea un nuovo mutex e lo inizializza come sbloccato, con:

- `mutex` (output), puntatore di tipo `pthread_mutex_t`;
- `attr` (input), per impostare gli attributi del mutex (può essere `NULL`).

`pthread_mutex_destroy(mutex)` disattiva il mutex specificato. Per le operazioni di lock e unlock:

- `pthread_mutex_lock(mutex)`, un thread invoca la lock su un mutex per acquisire l'accesso in mutua esclusione alla sezione critica relativa e, se è già acquisito da un altro thread, il chiamante si blocca in attesa di un unlock;
- `pthread_mutex_unlock(mutex)`, un thread invoca la unlock su un mutex per rilasciare la sezione critica e per consentire, quindi, l'accesso ad un altro thread precedentemente bloccato;
- `pthread_mutex_trylock(mutex)`, analoga alla lock ma non bloccante, cioè se il mutex è già acquisito ritorna immediatamente con un codice di errore `EBUSY`;
- **Gestione delle Condition Variables**, in particolare creazione, distruzione, wait e signal su variabili condition definite dal programmatore

La cooperazione tra thread avviene mediante condition variable, CV, sempre usate in abbinamento con un mutex, realizzando un **costrutto monitor di tipo signal and continue**. Per la creazione e la distruzione:

- `pthread_cond_t`, una struttura dati opaca per il principio di encapsulazione e rappresentante di un oggetto – varcondition;
- `pthread_cond_init(condition, attr)`, inizializza la CV per l'uso, con `condition` (output) un puntatore di tipo `pthread_cond_t` e `attr` (input) per settare gli attributi della CV (può essere `NULL`);
- `pthread_cond_destroy(condition)`, disattiva la CV che non serve più.

Per la wait e la signal:

- `pthread_cond_wait(condition, mutex)`, blocca il thread chiamante finché non si invoca la `pthread_cond_signal` e richiede in ingresso anche il mutex del monitor, automaticamente rilasciato al momento della chiamata e riacquisito alla riattivazione;
- `pthread_cond_signal(condition)`, serve a risvegliare un thread precedentemente bloccato sulla CV e semplifica il meccanismo di signal and continue;
- `pthread_cond_broadcast(condition, mutex)`, riattiva tutti i thread bloccati su una CV (`signal_all`) ma comunque il monitor è acceduto uno alla volta.

In PThreads, **un monitor può essere ottenuto combinando un mutex e una o più variabili condition**, tipicamente tutto in una struct allocata nello heap e condivisa fra thread:

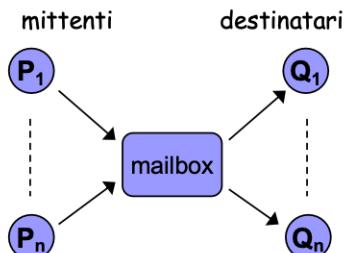
```
struct MyMonitor {
    ... // es., un buffer
    ...
    pthread_mutex_t mutex;
    pthread_cond_t cv1;
    pthread_cond_t cv2;
};
```

Riassumendo, schematicamente, le primitive multi – processo e multi – thread:

	Multi-processo	Multi-thread
Creazione di un flusso	fork()	pthread_create()
Condivisione di dati	shmget() + shmat()	malloc(), dati globali
Attesa dei figli	wait()	pthread_join()
Ingresso monitor	enter_monitor()	pthread_mutex_lock()
Sospensione su var. condition	wait_condition()	pthread_cond_wait()
Attivazione su var. condition	signal_condition()	pthread_cond_signal()
Uscita monitor	leave_monitor()	pthread_mutex_unlock()

LO SCAMBIO DEI MESSAGGI

UNIX supporta la comunicazione indiretta fra processi tramite **mailbox**, dette **code di messaggi**, con le primitive **send asincrona** e **receive** (bloccante e non); sia in ingresso che in uscita, comunque, viene inviato un messaggio, inteso tecnicamente come un vettore di byte.



Per la **creazione di una coda di messaggi** si fa riferimento alla seguente istruzione:

```
int msgget(chiave, flag);
```

Con **chiave** e **flag** che ricoprono lo stesso ruolo di **semget()** e **shmget()**. Si noti che **non occorre indicare la dimensione della coda ed il numero massimo di messaggi**, essendo informazioni decise e gestite autonomamente dal Sistema Operativo.

msgctl() è una primitiva per la **gestione della mailbox**; in particolare, **per la deallocazione** (immediata, senza attendere l'uscita di tutti i processi):

```
msgctl(msqid, IPC_RMID, 0);
```

Per l'**invio**, con accodamento, **di un messaggio alla mailbox**:

```
int msgsnd(int msqid, void *msgp, size_t msgsz, int msgflg);
```

Con msgp puntatore a una struttura – messaggio e msgsz la sua dimensione. Il sistema fornisce una struttura – messaggio, la struct msgbuf, definita in msg.h come:

```
struct msgbuf {  
    long message_type;  
    char message_text[MAX_SIZE];  
};
```

Non è obbligatorio inviare un vettore di caratteri, il programmatore può **definire una propria struttura** con qualsiasi altro tipo di dato, **ma è sempre obbligatorio un primo campo di tipo long strettamente positivo** (type>0, i valori negativi e lo zero sono riservati ad altri usi) **che specifichi il “tipo” di messaggio**; tuttavia, **nella specifica della dimensione del messaggio va esclusa la dimensione del campo long in questione**. Questa flessibilità con il tipo di struct – messaggi è dovuta alla **possibilità di fare casting tra puntatori quando viene fatta una chiamata a msgsnd**.

Quindi, per l'invio di un messaggio tramite la primitiva **msgsnd**, sono da rispettare alcune convenzioni:

- Il primo campo della struct, il type, deve essere sempre di tipo long;
- Il valore del long deve essere sempre strettamente positivo, cioè >0;
- Il terzo parametro di msgsnd, di tipo size_t, deve indicare la dimensione in byte della struct alla quale è sottratta la dimensione del parametro long.

Per la ricezione di un messaggio da una mailbox, invece:

```
int msgrcv(int msqid, void *msgp, size_t, long msqtyp, int msgflg);
```

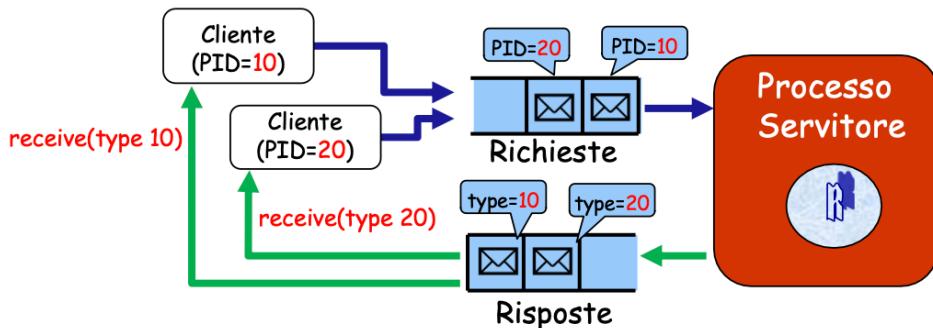
Con msgp puntatore a una struttura – messaggio (parametro di uscita), msgsz la sua dimensione e msqtyp il tipo di messaggio da ricevere (ricezione selettiva).

Il campo type, obbligatorio, permette al programmatore di distinguere tra più tipi di messaggi ed agire di conseguenza, pratica detta **ricezione selettiva**:

```
if(msg.type == ACCENDI) {  
    R.stato = ACCESO  
}  
  
else if(msg.type == SPEGNI) {  
    R.stato == SPENTO  
}  
  
else if(msg.type == LEGGI) {  
    risposta = R.temperatura  
    send(risposta)  
}
```

Il campo tipo può essere anche usato per abilitare la comunicazione indiretta simmetrica: sono usate due code condivise fra server e client, per richieste e risposte, **con ogni client che invia una**

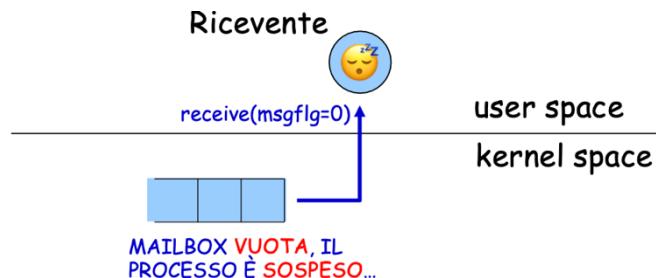
richiesta includendo il suo PID e con il server che invia le risposte usando il PID nel campo tipo; in questo modo, i client ricevono selettivamente solo le risposte con i propri PID. Schematicamente:



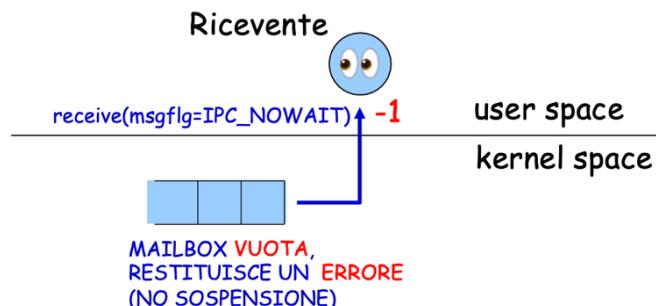
In `msgrecv`, il terzo parametro, `int msgtyp`, indica il tipo di messaggi da ricevere; in genere:

- `msgtyp == 0`, il prelievo avviene in ordine FIFO;
- `msgtyp > 0`, si preleva il primo messaggio tale che `type == msgtyp`;
- `msgtyp < 0`, si preleva il primo messaggio tale che `type >= abs(msgtyp)`;

Infine, il parametro `msgflg` determina se la `msgrecv` effettua una ricezione bloccante o meno, sulla base del suo valore, nullo o meno. Se `msgflg = 0`, il processo si sospende sulla `msgrecv` fino al giungere del messaggio:



Se `msgflg = IPC_NOWAIT`, invece, la `msgrecv` effettua una ricezione non bloccante ed il processo non si sospende se non ci sono messaggi da ricevere, restituendo immediatamente un valore di ritorno positivo (se è disponibile un messaggio) o negativo (se non ci sono messaggi).



Oltre all'assenza di messaggi, possono verificarsi altre anomalie (mancanza di permessi, ...) e, pertanto, è importante controllare anche la variabile globale `errno`; ad esempio:

```

mymessage msg;
// receive non-bloccante
result = msgrcv(msqid, &msg, sizeof(mymessage)-sizeof(long), 0,
                 IPC_NOWAIT);
if(result >= 0) {

    printf("Messaggio disponibile: tipo=%d, val=%d, text=%s\n",
           msg.type, msg.val, msg.text);
}
else if(result < 0 && errno == ENOMSG) {

    printf("Nessun messaggio disponibile\n");
    // ... effettua altro lavoro ...
}
else {
    perror("Errore msgrcv");
}

```

Alcuni tipi di messaggi sono:

ERRORS [top](#)

The `msgrcv()` function shall fail if:

E2BIG The value of `mtext` is greater than `msgsz` and `(msgflg & MSG_NOERROR)` is 0.

EACCES Operation permission is denied to the calling process; see [Section 2.7, XSI Interprocess Communication](#).

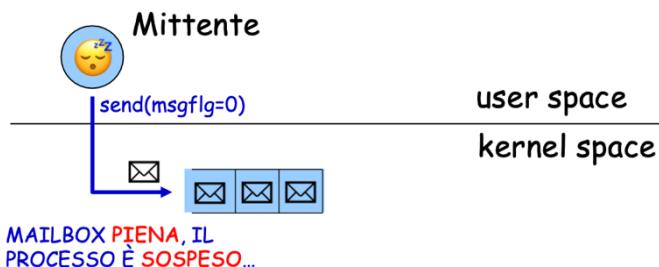
EIDRM The message queue identifier `msqid` is removed from the system.

EINTR The `msgrcv()` function was interrupted by a signal.

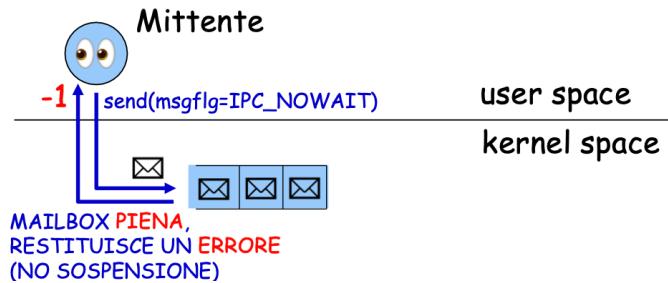
EINVAL `msqid` is not a valid message queue identifier.

ENOMSG The queue does not contain a message of the desired type and `(msgflg & IPC_NOWAIT)` is non-zero.

Il parametro `msgflg`, inoltre, permette di cambiare il comportamento di `msgsnd` in caso di mailbox piena; infatti, se `msgflg = 0`, si sospende il processo solo nel caso in cui la mailbox sia già piena, ovvero se si è raggiunti il limite di messaggi accodabili dal Sistema Operativo:



Invece, se `msgflg = IPC_NOWAIT`, il processo non si sospende mai, neanche se la mailbox è piena (in tal caso ritornerebbe il valore -1):



Va fatta una annotazione molto importante: la `msgsnd` in UNIX è sempre asincrona, indipendentemente da `IPC_NOWAIT`; per ottenere una `send` sincrona è necessario applicare lo schema del rendezvous.

Le primitive per lo scambio di messaggi sono inclini ad errori di utilizzo, a causa di alcune convenzioni da rispettare ed il mancato rispetto di queste regole spesso non è rilevato dal compilatore, portando a malfunzionamenti difficili da individuare; di seguito sono elencati alcuni degli errori più comuni:

- **Corruzione del contenuto dei messaggi:**
 - La lettura di campi numerici nel messaggio produce valori inattesi e negativi, come -1324929;
 - È indicativo che l'area di memoria che il programma sta leggendo non è valida (ad esempio, non è mai stata riempita con dei dati);
- **Stallo**, o terminazione prematura, della primitiva `msgrecv` nonostante ci siano vi siano dei messaggi;
- **Valori di ritorno negativi** dalle chiamate a `msgsnd` e `msgrecv`;
- **Mancanza del campo type** nella definizione del messaggio:
 - Omettere il campo `type` fa interpretare al Sistema Operativo i primi byte della struttura – messaggio sempre come `type`;
- **Posizione non corretta del campo type** nel messaggio:
 - È richiesto che il campo `type`, oltre ad esserci, sia sempre il primo campo della struttura – messaggio, dal momento in cui il Sistema Operativo si aspetta di trovare nei primi byte il tipo del messaggio;
- **Campo type int invece che long:**
 - È richiesto `long` invece che `int` perché alcune architetture trattano i due tipi in maniera completamente diversa;
 - Ad esempio, su CPU a 32 bit `long` e `int` hanno stessa dimensione (4 byte) ma su CPU a 64 bit il primo ha dimensione doppia rispetto a quella del secondo;
- **Campo type nel messaggio impostato a zero:**
 - Il valore 0 è da riservare ad usi speciali (come la possibilità di ricevere messaggi di qualsiasi tipo in `msgrecv`) e non va riservato a casi comuni;
- **Campo type non inizializzato:**
 - Se non inizializzato, il campo `type` è imprevedibile ed arbitrario (dipenderebbe dal Sistema Operativo, dall'hardware e dal compilatore);
- **Errato calcolo della dimensione del messaggio:**
 - La dimensione di una struttura non è necessariamente uguale alla somma dei singoli campi, spesso il compilatore fa delle ottimizzazioni;
 - Usare sempre `sizeof(messaggio) - sizeof(long)`;
- **Errato uso di puntatori alla struttura – messaggio:**

- È un indirizzo ad un'area di memoria che non contiene una struttura messaggio ma dati arbitrari e, pertanto, non validi;
- Il Sistema Operativo interpreta l'area puntata come se fosse una struttura – messaggio;
- Può causare l'invio/ricezione di messaggi corrotti o il blocco di msgsnd/msgrcv;
- Ad esempio, passaggio erroneo di un puntatore a puntatore in ingresso a msgsnd/msgrcv;

```
void myfunction(mymessage * msg) {
    ...
    // ERRORE! "msg" è già un puntatore, non serve "&"
    msgsnd(msqid, &msg, sizeof(mymessage)-sizeof(long), 0);
    ...
}
```

- **Errato uso degli identificatori delle code:**

- Quando si usano molte risorse UNIX (ad esempio, molte code), può capitare di riutilizzare la stessa chiave su chiamate diverse;
- Oppure, può capitare di usare un msgid diverso da quello che si intendeva usare.

```
key_t prima_chiave = ftok("./", 'a');

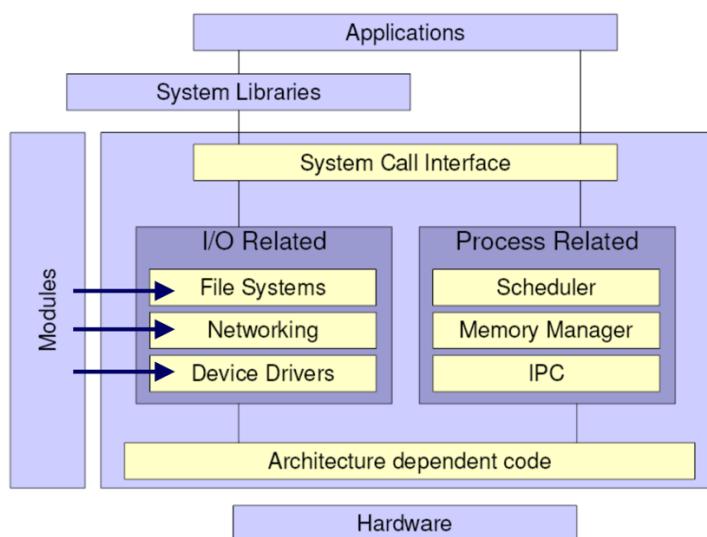
int richieste_id = msgget(prima_chiave, IPC_CREAT|0644);

key_t seconda_chiave = ftok("./", 'b');

// ERRORE! Intendevamo usare "seconda_chiave" qui
int risposte_id = msgget(prima_chiave, IPC_CREAT|0644);
```

SVILUPPO DI MODULI PER IL KERNEL LINUX

Linux è un Sistema Operativo monolitico e modulare, ciò significa che il codice del kernel è contenuto in un unico file eseguibile, utilizzando un unico spazio di indirizzamento e permettendo di caricare ulteriore codice da moduli a tempo di esecuzione.



Ci si chiede, **perché compilare il kernel Linux?** Essenzialmente per diversi motivi:

- **Minore occupazione di memoria e storage** (ad esempio, per sistemi embedded);
- **Abilitare driver e funzionalità specifiche;**
- **Installare modifiche di terze parti;**
- **Ottimizzazione** (ad esempio, selezione di CPU family, SMP, layout di memoria, ...);
- **Utilizzo del sistema di build per sviluppare un nuovo kernel;**
- **Studiare il kernel.**

Generalmente, è **raccomandabile installare la stessa versione, o almeno simile, del kernel già utilizzata dalla propria distribuzione**. Prima di iniziare la compilazione, però, **occorre copiare i sorgenti, installare le librerie e i tool necessari alla compilazione**, dopodiché si può passare alla fase di compilazione. In fase di configurazione è permesso di **selezionare le funzionalità e i moduli da compilare**, escludendo ciò che non serve; ad esempio:

```
make xconfig  
make menuconfig  
make config
```

Il primo comando seleziona e compila i moduli grafici (consigliati), mentre gli altri due moduli relativi alla linea di comando. Per **creare una configurazione di partenza**, usare il comando **make localmodconfig**, inizializzando la configurazione **in base ai moduli in uso nel sistema**.

Ciascuna opzione di configurazione può essere:

- **Compilata all'interno del kernel**, Y simbolo ✓;
- **Compilata in un modulo separato**, M simbolo •;
- **Esclusa dalla compilazione**, N casella vuota.

Alcune opzioni possono richiedere valori numerici o stringhe. Tra le opzioni di configurazione, **LOCALVERSION** permette di modificare il nome assegnato al kernel; ad esempio:

```
LOCALVERSION="-CorsoSO" → 5.4.0-CorsoSO
```

Altri esempi di opzioni di configurazione sono le **modalità di preemption del kernel**, la **frequenza del timer**, le **ottimizzazioni per la CPU** (come SMP, HyperThreading, famiglia di CPU, ...). Molte opzioni riguardano il supporto hardware della macchina su cui il kernel sarà eseguito; per eventuali informazioni sul proprio hardware:

```
lspci -vvv  
cat /proc/cpuinfo  
hal-device-manager  
lsmod
```

Una configurazione viene salvata nel file **.config** nella cartella dei sorgenti; un file che inizia con il punto è detto **file invisibile** e non può essere evidenziato con il comando **ls** se non con il flag **-a**. Per rimuovere un file di configurazione, è possibile **utilizzare il comando make clean**, mentre per compilare sia kernel che moduli **make** e per compilare in maniera parallela (utile per sfruttare la SMP) **make -j 9** (dove 9 rappresenta il numero di jobs se si hanno 8 core). Per installare l'immagine del kernel nella cartella **/boot**:

```
sudo cp arch/x86_64/boot/bzImage /boot/vmlinuz-VERSIONE
```

Mentre per copiare i moduli kernel nella cartella **/lib/modules/**:

```
sudo make modules_install
```

E per salvare una copia del file di configurazione:

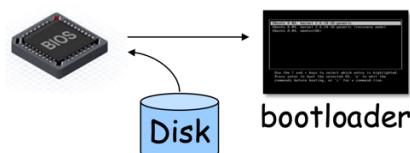
```
sudo cp .config /boot/config-VERSIONE
```

Come si è potuto intuire, la cartella **/boot** raccoglie i file del kernel, come:

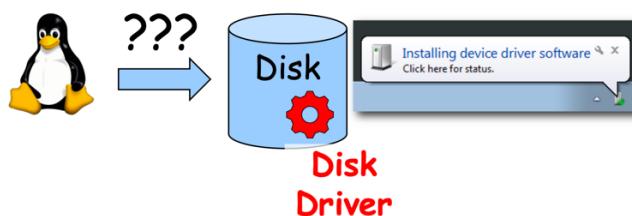
- config;
- initrd.img;
- vmlinuz;
- System.map (opzionale).

Mentre la cartella **/lib/modules** raccoglie i moduli del kernel (in particolare, una sottocartella per ogni versione).

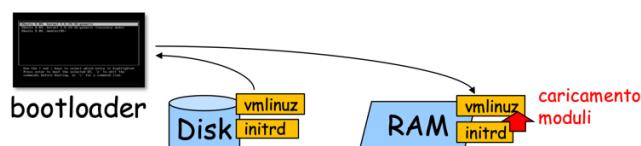
Il bootloader è un piccolo programma che viene eseguito all'avvio del computer: viene prelevato dal BIOS dai primi settori del disco e può interagire con l'utente per selezionare quale Sistema Operativo avviare.



Il kernel ha anche bisogno dei driver di I/O (ad esempio, disco, rete, filesystem, ...) per caricare i moduli al boot; ma che succede se a loro volta anche i driver di I/O sono moduli? È il caso tipico delle distribuzioni Linux, che includono il maggior numero possibile di driver di I/O per una maggiore compatibilità, compilati come moduli ed evitando che il file vmlinuz diventi eccessivamente grande.



I moduli strettamente necessari sono copiati in un file detto **initial RAM disk**, caricato dal bootloader su RAM, da cui in un secondo momento il kernel preleva i driver.



Il comando per la creazione di un initial RAM Disk va eseguito dopo aver installato i moduli con `make modules_install`, e si presenta come:

```
sudo mkinitramfs -o /boot/initrd.img-VERSIONE VERSIONE
```

Anche se alcune distribuzioni usano il programma `mkinitrd`.

Occorre, inoltre, configurare il bootloader per utilizzare il kernel appena compilato; tra le distribuzioni Linux, **il bootloader più usato e comune è GRUB.** Per far apparire il menu di GRUB all'avvio, **va modificato il file `/etc/default/grub`, aggiungendo # all'inizio della riga `GRUB_TIMEOUT_STYLE=hidden` e impostando `GRUB_TIMEOUT` ad un valore maggiore di 0.** Per aggiornare GRUB, `sudo update-grub` e, al riavvio successivo, **si potrà scegliere di avviare il nuovo kernel invece di quello di default.** Si annoti che **il comando `make install` copia automaticamente i file `vmlinuz` e `initrd` in `/boot` ed aggiorna il bootloader.**

Riassumendo:

- Scaricare i sorgenti;
- Effettuare la configurazione:
 - `sudo make localmodconfig` (seleziona automaticamente i moduli kernel in uso nel sistema);
 - Modificare la configurazione, settando il parametro `LOCALVERSION`;
- Lanciare la compilazione:
 - `sudo make`;
- Installare i moduli:
 - `sudo make modules_install`;
- Installare `vmlinuz` e aggiornare il bootloader:
 - `sudo make install`.

Volendo utilizzare il nuovo kernel, per verificarne la versione:

```
uname -a
```

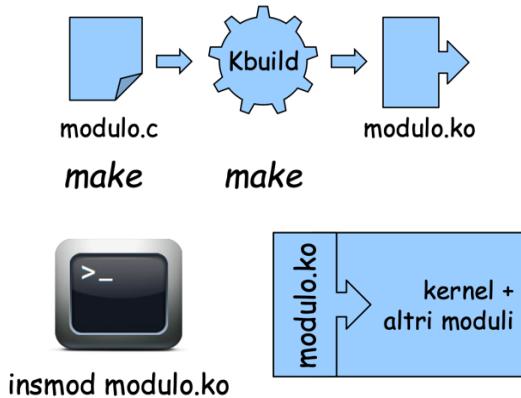
Ovviamente, **per usare un nuovo kernel in una VM può occorrere installare nuovamente i `vmware-tools`** (includono dei moduli kernel che vanno ricompilati per poter essere eseguiti nel nuovo kernel). Di seguito sono elencati alcuni errori comuni:

- La configurazione del kernel non include alcuni moduli necessari all'avvio (ad esempio, i driver del disco o del filesystem);
- La creazione dello initial RAM Disk è fallita (non bisogna ignorare eventuali warning durante la creazione);
- Errori nella conversione dei nomi (ad esempio, l'uso della stringa `LOCALVERSION` all'interno dei nomi dei file).

Prima di creare lo init RAM Disk, cancellare il file `/etc/default/grub.d/50-cloudimg-settings.cfg` e, prima di lanciare `update-grub`, modificare il file `/etc/initramfs-tools/initramfs.conf` configurando `COMPRESS=bzip2`.

I moduli del kernel permettono di rendere il sistema più flessibile; in particolare, permettono **l'installazione di software di terze parti, il caricamento di funzionalità on – demand e lo sviluppo e il collaudo di nuovo codice senza dover riavviare il sistema.** I sorgenti del modulo

vengono **compilati sfruttando i tool** (come Kbuild) e **gli header file inclusi nei sorgenti del kernel**; dopodiché, **il modulo viene caricato dinamicamente, su richiesta dell'utente, oppure dai processi di sistema**.



Nel dettaglio, **il modulo è una collezione di funzioni** (entry point) eseguite quando si verificano:

- Caricamento e rimozione del modulo;
- Chiamate di sistema (o altre interazioni tra user e kernel mode);
- Interruzioni.

Un esempio di modulo è il seguente:

```
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

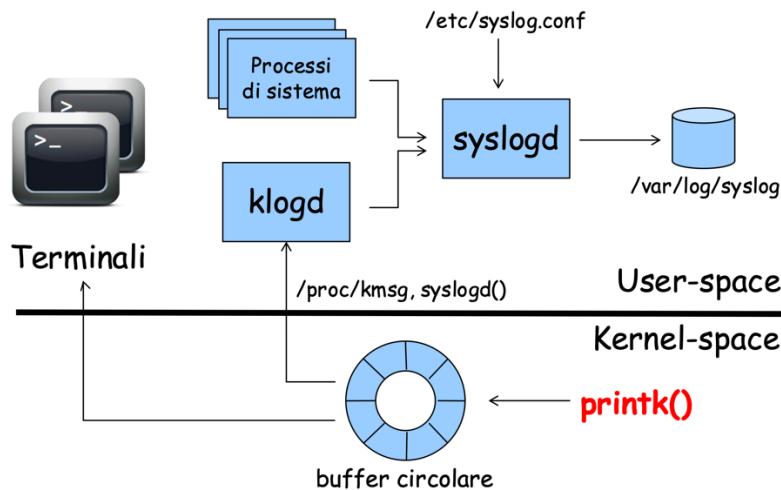
MODULE_LICENSE("GPL");

int __init__module(void) {
    printk(KERN_INFO "Hello world 1.\n");

    return 0;
}
void __exit__module(void) {
    printk(KERN_INFO "Goodbye world 1.\n");
}
```

Il codice del kernel non ha associato un terminale su cui far apparire i messaggi; `printk()` scrive su un buffer circolare, i cui messaggi sono rediretti verso un processo di sistema (`syslogd`), consultabili via:

- Comando `dmesg`;
- Il file di log `/var/log/syslogd`;
- In tutti i terminali (nei casi da alata severità).



Mentre i livelli di severità dei log:

Loglevel	Description
KERN_EMERG	An emergency condition; the system is probably dead.
KERN_ALERT	A problem that requires immediate attention.
KERN_CRIT	A critical condition.
KERN_ERR	An error.
KERN_WARNING	A warning.
KERN_NOTICE	A normal, but perhaps noteworthy, condition.
KERN_INFO	An informational message.
KERN_DEBUG	A debug message—typically superfluous.

La compilazione si basa sul sistema Kbuild di compilazione del kernel (ovvero Makefile e script per la shell), con sorgenti e Makefile del nuovo modulo che possono risiedere in qualunque directory. Ad esempio:

```
obj-m += hello-1.o
VERSIONE_KERNEL=$(shell uname -r)
all:
    make -C /lib/modules/${VERSIONE_KERNEL} /build M=$PWD modules
clean:
    make -C /lib/modules/${VERSIONE_KERNEL} /build M=$PWD clean
```

Directory corrente

È un link simbolico verso la cartella
dei sorgenti del kernel (creato da
"make modules_install")

Il processo di compilazione ed utilizzo, invece, prende la seguente forma:

- **Compilazione**

make

- **Caricamento**

```
sudo insmod ./hello-1.ko
```

- **Visualizzazione elenco moduli**

```
lsmod
```

- **Rimozione**

```
sudo rmmod hello-1
```

- **Informazioni sul modulo**

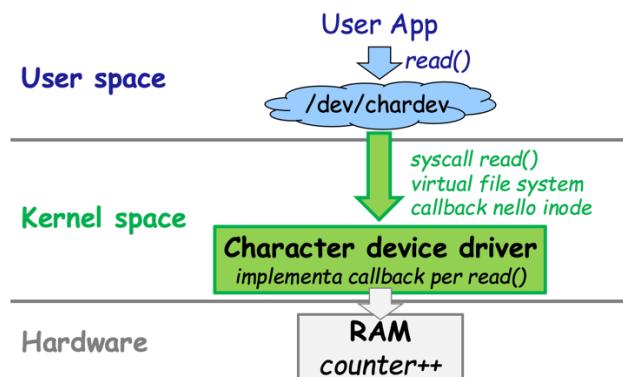
```
modinfo ./hello-1.ko
```

A differenza dei processi, un **bug nella gestione della memoria nel kernel** (ad esempio, un indirizzo invalido) **rende instabile l'intero sistema**; un **oops** è un **messaggio diagnostico prodotto a seguito di eccezioni occorse in kernel mode**: il kernel prova ad uccidere il processo corrente e a continuare l'esecuzione. Sotto alcune determinate circostanze (ad esempio, durante il boot oppure in una ISR) il kernel si ferma, entrando in stato di kernel panic.

Si annoti che **il kernel non può utilizzare le funzioni di libreria C**; impiega, invece, i seguenti sostituti:

- Per il logging, `printk()`;
- Per l'allocazione della memoria, `kmem_cache_alloc()`, `kmalloc()`;
- Per la sincronizzazione, `spin_lock()`/`unlock()`, `wake_up_interruptible()`;
- ...

L'**aritmetica floating point**, inoltre, **non è di semplice utilizzo** (a causa della necessaria gestione delle eccezioni) e **gli stack del kernel sono limitati** (1 o 2 pagine); **occorre, poi, gestire la sincronizzazione con altre parti del kernel** (system call, ISR, kernel threads, ...) **che accedono a dati comuni** (come, ad esempio, il PCB). Ad esempio, un device driver per dispositivo a caratteri (fittizio):



Un processo può leggere/scrivere il file virtuale `/dev/chardev`; il Sistema Operativo, poi, richiama le funzioni del device driver:

- Lettura (system call `open + read`), il device driver copia una stringa con il numero di volte che il file è stato acceduto;

- Scrittura (system call `write`), ignorate, scrive un log e ritorna un errore al processo chiamante.

Il comando `mknod` crea un device file e lo abbina al modulo appena caricato in base ad un major number. Il **major number** è un **numero identificativo del driver che deve gestire un device file** e può essere **scelto dallo sviluppatore o scelto dinamicamente dal kernel** e, essenzialmente, **identificano una classe di dispositivi**; il **minor number**, invece, è un **numero secondario, nel caso un driver gestisca più device file**.

Il **modulo del device driver** deve **registrarsi come tale presso il kernel e registrare le funzioni del modulo atte a gestire le operazioni sul device file** (apertura, lettura, scrittura, ...):

```
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);
```

Esistono anche **funzioni analoghe per altre classi di dispositivi**, come USB, SATA, Ethernet, ...
Per quanto riguarda le **operazioni su device a caratteri**:

```
struct file_operations {  
    struct module *owner;  
    ...  
    int (*open) (struct inode *, struct file *);  
  
    ssize_t(*read) ( struct file *, char __user *,  
                    size_t, loff_t * );  
  
    ssize_t(*write) ( struct file *, const char __user *,  
                     size_t, loff_t * );  
    ...  
};
```

Mentre la **struttura generale del modulo**:

```
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

static int Major;           /* Major number assigned to our device driver */
static int Device_Open = 0; /* Is device open?
                           * Used to prevent multiple access to device */
static char msg[BUF_LEN]; /* The msg the device will give when asked */
static char *msg_Ptr;

static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
```

L'inizializzazione:

```
int init_module(void)
{
    Major = register_chrdev(0, DEVICE_NAME, &fops);

    if (Major < 0) {
        printk(KERN_ALERT "Registering char device failed with %d\n", Major);
        return Major;
    }

    printk(KERN_INFO "I was assigned major number %d. To talk to\n", Major);
    printk(KERN_INFO "the driver, create a dev file with\n");
    printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);
    printk(KERN_INFO "Try various minor numbers. Try to cat and echo to\n");
    printk(KERN_INFO "the device file.\n");
    printk(KERN_INFO "Remove the device file and module when done.\n");

    return SUCCESS;
}
```

L'operazione di apertura:

```
/*
 * Called when a process tries to open the device file, like
 * "cat /dev/mycharfile"
 */
static int device_open(struct inode *inode, struct file *file)
{
    static int counter = 0;

    if (Device_Open)
        return -EBUSY;

    Device_Open++;
    sprintf(msg, "I already told you %d times Hello world!\n", counter++);
    msg_Ptr = msg;

    try_module_get(THIS_MODULE);

    return SUCCESS;
}
```

E, infine, l'operazione di lettura:

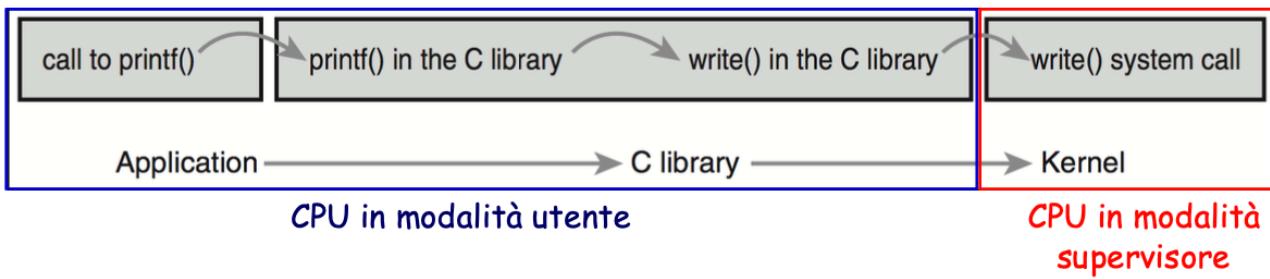
```
static ssize_t device_read(
    struct file *filp,          /* see include/linux/fs.h */
    char *buffer,                /* buffer to fill with data */
    size_t length,               /* length of the buffer */
    loff_t *offset ) {

    int bytes_read = 0; // Number of bytes actually written to the buffer

    if (*msg_Ptr == 0)           // If we're at the end of the message,
        return 0;                // return 0 signifying end of file

    while (length && *msg_Ptr) {
        /* The buffer is in the user data segment, not the kernel segment, so
         * "*" assignment won't work. We have to use put_user which copies
         * data from the kernel data segment to the user data segment.*/
        put_user(*msg_Ptr++, buffer++);
        length--;
        bytes_read++;
    }
    return bytes_read;
}
```

Le chiamate di sistema sono l'interfaccia tra i processi ed il Sistema Operativo e occorrono quando la CPU passa da modalità utente (esecuzione dei processi) a modalità kernel (esecuzione del codice del kernel). Linux fornisce oltre 300 chiamate di sistema, perlopiù basate sullo standard IEEE POSIX ed encapsulate (wrapped) dalla libreria C:

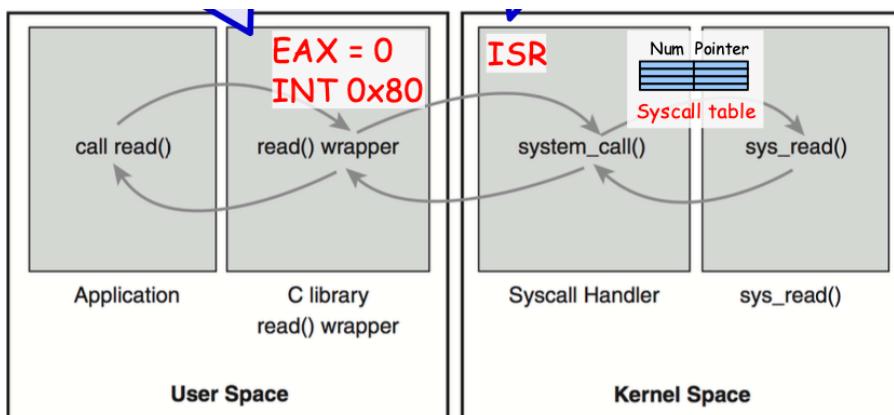


Le funzioni nel kernel che implementano **syscall** hanno, convenzionalmente, il prefisso **sys_**:

```
asmlinkage long sys_getpid(void) {
    return current->tgid;
}
```

Con **currrent** puntatore globale al PCB del processo in esecuzione. In Linux, **il PID di un processo** è anche detto **tgid**, o **Thread Group ID**.

La chiamata ad una **syscall** avviene scrivendo il **syscall number** su un registro e invocando la **supervisor call**; a quel punto, la **ISR** (syscall handler) cerca il **syscall number** nella **syscall table** e trova il puntatore alla funzione da eseguire:



Per **Application Binary Interface (ABI)** si intende **un insieme di convenzioni tra programmi utente e Sistema Operativo**; ad esempio, in x86 il passaggio di parametri avviene tramite:

- Syscall number, EAX;
- Parametri di ingresso, EBX, ECX, EDX, ESI, EDI, EBP;
- Parametro di uscita, EAX.

Il kernel, poi, si occupa di garantire la stabilità ad ogni versione.

Per implementare una **system call**, va prima inserita in un nuovo file all'interno dei sorgenti:

<sorgenti>/kernel/mysyscall.c

Va, poi, **modificato il Makefile nella cartella del nuovo file**; ad esempio, nel file <sorgenti>/kernel/Makefile:

obj-y += mysyscall.o

Infine, va aggiunta una nuova riga alla system call table e ricompilato il kernel; per x86:

```
<sorgenti>/arch/x86/entry/syscall/syscall_64.tbl
```

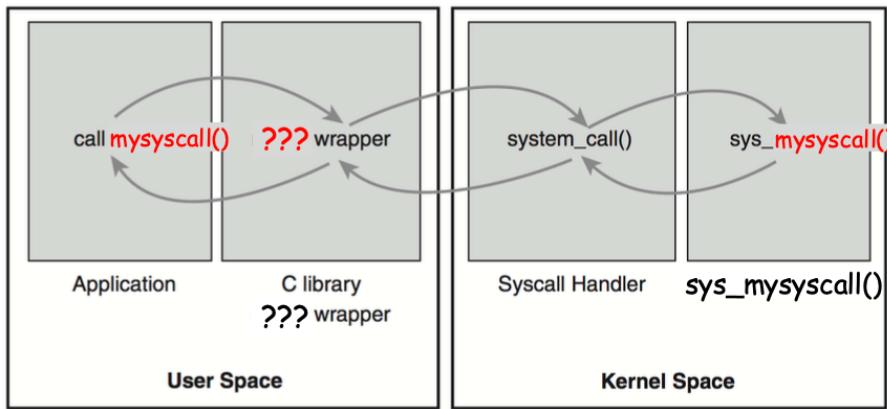
Lato kernel, è possibile anche usare la macro SYSCALL_DEFINEn, invece della classica sintassi del linguaggio C, per poter esportare la definizione anche ad altri tool esterni:

```
SYSCALL_DEFINEn(mysyscall, int, PARAMETRO1, char *, PARAMETRO2, ....)
{
    // .... implementazione della system call ....
    return <VALORE LONG>;
}
```

La system call table sarà un file la cui struttura sarà simile a quella seguente:

```
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0      common  read           __x64_sys_read
1      common  write          __x64_sys_write
2      common  open           __x64_sys_open
...
332    common  statx         __x64_sys_statx
333    common  io_pgetevents __x64_sys_io_pgetevents
334    common  rseq           __x64_sys_rseq
#
# La nuova system call
335    common  mysyscall     __x64_sys_mysyscall
...
```

Per la nuova system call non esiste alcuna funzione nelle librerie di sistema; è, quindi, necessario scrivere una funzione wrapper apposita per la nuova system call.



La libreria C fornisce un wrapper generico, denominato `syscall()`:

```

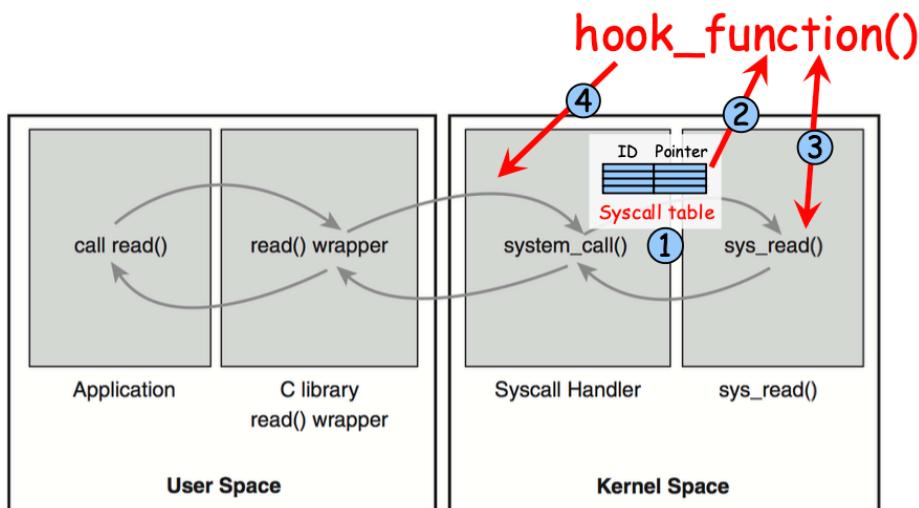
#define __NR_read 0

ssize_t my_read(int fd, void *buf, size_t count)
{
    return syscall(__NR_open, fd, buf, count);
}

```

Ovviamente, al posto di `__NR_read` va usato il numero usato in `syscall_64.tbl`.

Le chiamate di sistema devono verificare la validità dei parametri di ingresso dal processo chiamante; ad esempio, se si passa in ingresso un PID, occorre accertarsi che il PID esista davvero nel sistema, altrimenti è a rischio la sicurezza del kernel. In particolare, occorre garantire che i puntatori di memoria passati siano validi, per prevenire che la system call legga/scriva aree di memoria non autorizzate; a tal proposito, è possibile utilizzare `copy_from_user()` e `copy_to_user()` per garantire la copia sicura dei dati da e verso il processo.



Un modulo kernel malevolo (rootkit, come Diamorphine) permetterebbe di nascondere i processi, i file e le cartelle, le connessioni, di ottenere una shell di root e di fornire una backdoor per il sistema. In particolare, Diamorphine usa la tecnica dell'hooking, per la quale:

- Nella syscall table è inserito un puntatore ad una funzione malevola;

- Quando un processo chiama la syscall vittima (ad esempio, `read`), il kernel segue la tabella e chiama la funzione malevola;
- La funzione malevola chiama la system call;
- La funzione malevola modifica i risultati della system call vittima (ad esempio, omettendo un file o un processo per nasconderlo).

