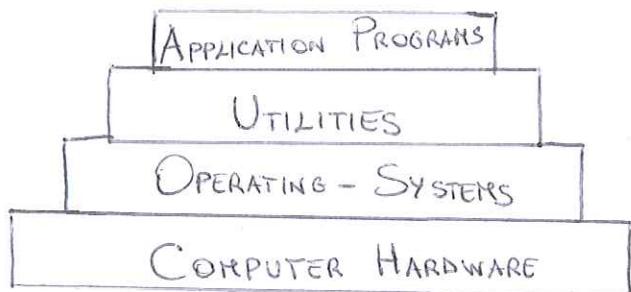


SISTEMI OPERATIVI

1. SO - INTRODUZIONE

Un **SISTEMA OPERATIVO** è un componente software che costituisce una soluzione ragionevole al problema dell'utilizzabilità delle strutture di calcolo.



Ha come obiettivi:

- **Semplicità**: rende lo sviluppo del software più semplice, nascondendo le peculiarità dell'hardware;
- **Efficienza**: OTTIMIZZA l'uso delle risorse da parte delle applicazioni;
- **Flessibilità**: garantisce la trasparenza verso le applicazioni di modifiche dell'hardware, garantendo la **PORTABILITÀ** del software.

Un modulo, una routine di un programma applicativo può chiamare un modulo del software del Sistema Operativo, affinché tutto funzioni correttamente ed in maniera OTTIMIZZATA per la SPECIFICA ARCHITETTURA HARDWARE.

La sezione delle UTILITIES è formata da altri moduli software su cui un programma applicativo può appoggiarsi e che può chiamare (per esempio le LIBRERIE).

Se, nello scrivere un programma si fa uso unicamente di UTILITIES, nel caso di corrispondenza di tali librerie tra S.O. diversi, si ha la PORTABILITÀ del software.

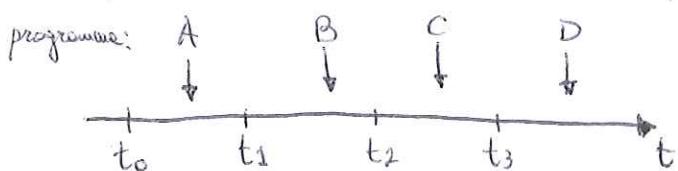
* Dunque, in genere, programmare direttamente sul Sistema Operativo rende il software NON PORTABILE! Se ne potrebbe beneficiare però in termini di efficienza.

• VIRTUALIZZAZIONE DELLE RISORSE:

All'utente ed alle applicazioni vengono mostrate risorse VIRTUALI (i FILES) più semplici da usare rispetto alle risorse fisiche, che corrispondono tra le 2 e sono mantenute in maniera trasparente dal SO (al livello fisico esiste l'HARD-DRIVE).

I vantaggi della virtualizzazione delle risorse sono la possibilità di soddisfare più utenti o più applicazioni in simultanea.

Si può virtualizzare anche la CPU, per far girare più programmi (TIME-SHARING):



⇒ Gestione del Sistema Operativo tramite interrupt.

• ELABORAZIONE SERIALE (anni '40 - '50):

Può essere eseguito un solo job per volta (schede perforate). L'utente deve caricare il compilatore ed il programma da eseguire in memoria. La quantità di tempo NON è MINIMA solo per gestire la preparazione di un job.

• SISTEMI OPERATIVI BATCH (anni '50 - '60):

Si basano sull'utilizzo di un MONITOR (monitoratore): insiemi di moduli software, specie di contenuto del Sistema Operativo.

L'insieme dei moduli software del MONITOR vengono caricati in memoria di lavoro e rimangono lì per l'eventuale esecuzione futura di altri programmi applicativi: dicono SOFTWARE RESIDENTE.



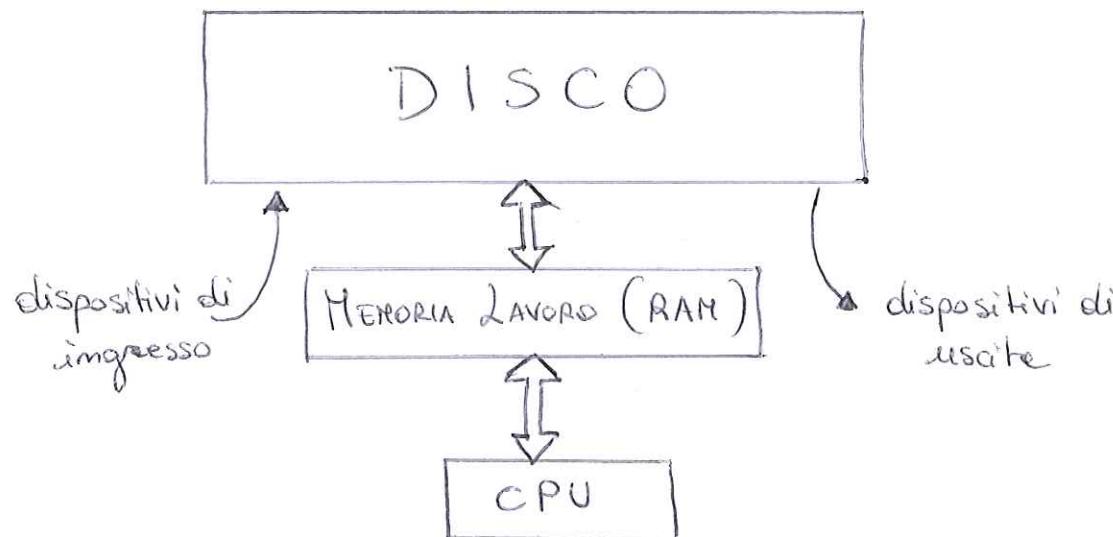
Limiti Principali: monoprogrammazione (un solo job per volta); sottoutilizzo delle CPU per le funzionalità dei dispositivi.

• SPOOLING:

SPOOLING := Simultaneous peripheral operation on-line

È una tecnica che si basa sull'uso di un DISCO DI MEMORIA (più veloce dei singoli dispositivi), utilizzato come BUFFER di memoria TAMPONE per i dispositivi input/output.

L'input viene anticipato su disco, l'output riferito da disco, in modo da ottenere una RIDUZIONE delle percentuali d'ATTESA delle CPU e la CONTEMPORANEITÀ di input e output di job diversi.



• SISTEMI OPERATIVI BATCH MULTITASKING:

Utilizzano le tecniche dello SPOOLING per eseguire più job simultaneamente:

Se il programma A richiede le chiamate ad un driver di dispositivo, nel frattempo, anziché attendere l'output del dispositivo, si può iniziare l'esecuzione del programma B. Se anche B richiede l'uso di un dispositivo (diverso da quello utilizzato da A), si ottimizza di molto l'uso delle CPU e le chiamate dei dispositivi.

ATTENZIONE!: c'è bisogno che entrambi i job siano caricati in MEMORIA DI LAVORO, lo svantaggio è che si riduce la capacità di mantenere in memoria job molto grandi (PARTIZIONI MULTIPLE della memoria, eseguite ai vari job). → SOLUZIONE: SPooling con uso di un dispositivo di memorizzazione ausiliario (disco), dove avviene una PARTIZIONE SINGOLA per i vari job che vengono di volta in volta scritti in o lette dalla memoria. Si paga un po' in latenze.

Dunque, si ha:

- gestione SIMULTANEA (multitasking) di molti job;
- un job per volta impegnare le CPU;
- più job possono impegnare i dispositivi CONTEMPORANEAMENTE.

LIMITI PRINCIPALI: Il controllo viene restituito al Monitor solo in caso di richiesta verso un dispositivo, terminazione o errore;

- (1) Rischio di sottoutilizzo delle risorse;
- (2) L'esecuzione di job con frequenti chiamate a dispositivi può essere penalizzata. Non si riesce quindi a gestire bene i PROGRAMMI INTERATTIVI.

• SISTEMI OPERATIVI TIME-SHARING (anni '60-'70):

Sorta di VIRTUALIZZAZIONE delle risorse CPU: il tempo di CPU viene eseguito ai vari job secondo un determinato ALGORITMO DI SCHEDULING stabilito dal monitor (per esempio, ROUND-ROBIN, tempi fissi per i job, scanditi da regole distinte).

Il monitor è un vero e proprio software di Sistema Operativo.

E' possibile che l'esecuzione di un job venga interrotta indipendentemente dal fatto che il job effettua una richiesta verso un dispositivo (PREEMPTION, prelezione).

Gli Interrupt:

Gli INTERRUPT rappresentano il metodo pratico che corrisponde al concetto di PREEMPTION.

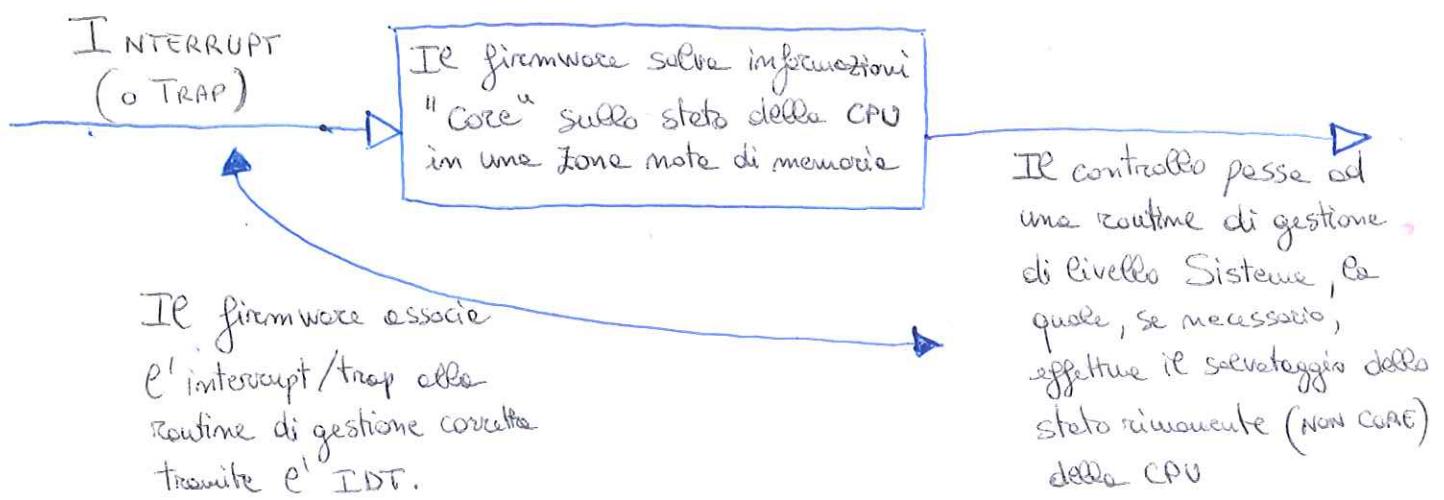
- Un INTERRUPT è in grado di trasferire il controllo ad una "ROUTINE DI INTERRUPT", tipicamente tramite un INTERRUPT VECTOR, cioè una TABELLA con una serie di entry, ognuna delle quali punta (PUNTATORI) ad un MODULO SOFTWARE nelle RAM dedicate al Sistema Operativo per la gestione di uno specifico interrupt.

In pratica, è una struttura dati di S.O., c'è anche un TIMER di sistema che può lanciare un interrupt, nel caso di "monopolizzazione" delle CPU da parte di un processo.

Nei processori x86, l'interrupt vector è chiamato IDT (Interrupt Description Table). Lo slot delle tabelle da scegliere per puntare al modulo software corretto che utilizzerà è determinato dal codice di interrupt che arriva.

- L'INTERRUPT è in grado di modificare il flusso d'esecuzione, cambiando il contenuto del registro PROGRAM COUNTER (PC). Fare solo questo, però, non basta; c'è un supporto hardware (o livello firmware) che provvede al salvataggio del valore dei registri di CPU e al valore del PC, prima di modificarlo, per poter riprendere l'esecuzione in un secondo momento.

- È chiaro che c'è sempre bisogno di sapere dove l'INTERRUPT VECTOR, infatti la CPU è dotata di un registro, l'IDTR (Interrupt Description Table Register), che punta sempre alle tabelle.



• SISTEMI OPERATIVI REAL-TIME:

A differenza dei Time-Sharing, devono far sì che un programma esegue specifici task entro delle DEADLINES TEMPORALI prestabilite.

Ci sono gli **HARD REAL-TIME**, in cui le deadlines devono essere assolutamente rispettate (per esempio per il controllo di processi industriali automatizzati).

E poi ci sono i **SOFT REAL-TIME**, in cui le deadlines dovrebbero essere rispettate, ma se si va oltre non è un problema ingestibile (per esempio applicazioni multimediali).

I PROCESSI

Sono introdotti per monitorare e controllare in modo sistematico l'esecuzione dei programmi. Di fatto, un **PROCESSO** è un programma in esecuzione con associati:

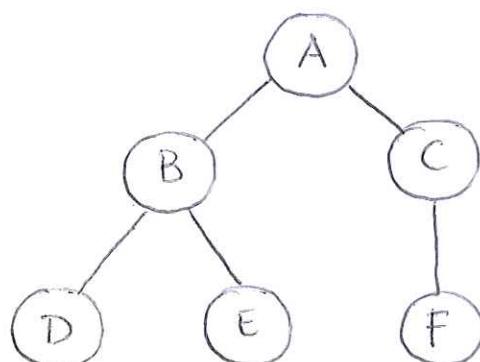
- (1) i dati su cui opera;
- (2) un CONTESTO DI ESECUZIONE: cioè le informazioni necessarie al S.O. per schedularlo, quali sono quanti job ci sono, gli screenshots delle CPU nelle IDTR, etc...,

È sempre controllato dal software di Sistema Operativo.

Quando un processo va in esecuzione sulla CPU puoi interagire sia col S.O. che con altri processi.

Il Sistema Operativo stesso puoi essere basato su un insieme di processi.

Un processo puoi creare altri processi (child), creare quindi dei rapporti di dipendenze o parentele e quindi delle GERARCHIE DI PROCESSI:



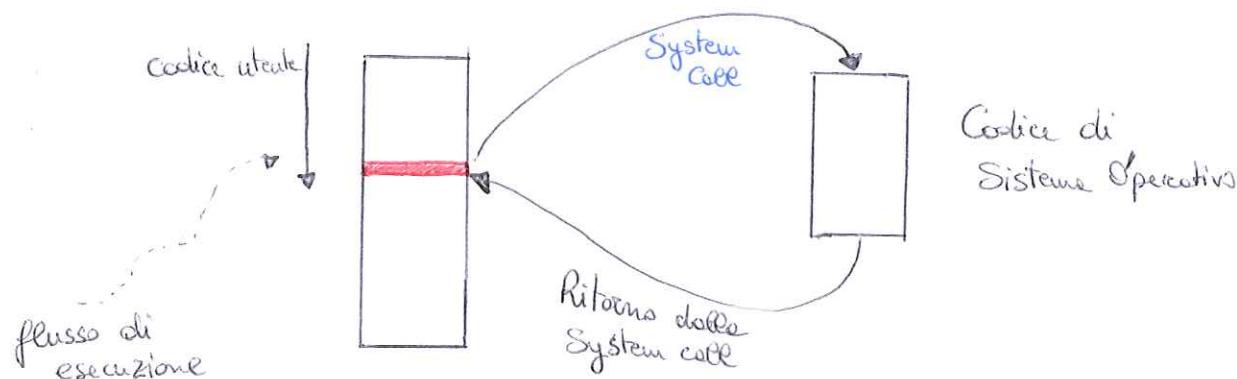
• SERVIZI CLASSICI DI UN SISTEMA OPERATIVO:

- Gestione dei processi;
- Gestione delle memorie primarie e secondarie;
- Gestione dei file;
- Gestione dell'I/O (inclusi dispositivi di rete);
- Protezione delle risorse, security.

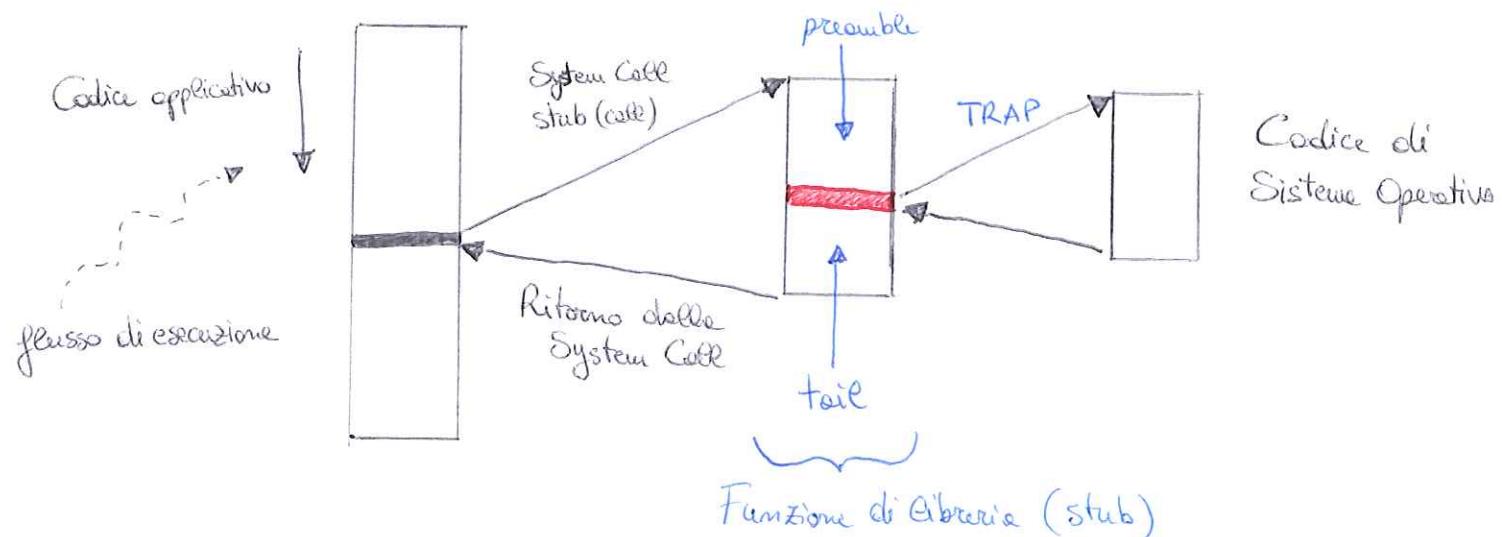
MECCANISMO DELLE "SYSTEM CALL"

Per poter chiamare e passare il controllo al software di Sistema Operativo, non si puo' fare una semplice jol, poiche il software di SO è PRIVILEGIATO rispetto a quello delle applicazioni; c'è bisogno di delle TRAP o delle "SYSTEM CALL".

Dunque, il meccanismo delle system call fornisce un'interfaccia per l'ACCESSO AL SOFTWARE del SISTEMA OPERATIVO.



Il reale supporto per le system call, sono le istruzioni di TRAP; quindi, invocare una syscall implica la presenza nel codice applicativo di istruzioni di trap. In realtà, ciò avviene tramite un INTERMEDIARIO: si usano "stub" (parti di codice) di LIBRERIE che ottengono delle istruzioni meccaniche per passare il controllo al SO, il quale usa un'altra parte di STACK, diverse da quella dei programmi applicativi. C'è un CAMBIO DI RING (livello di privilegio): RING di SO è il massimo livello.



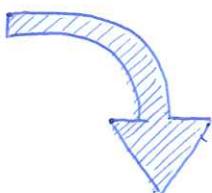
• ANATOMIA DI UNA SYSTEM CALL :

Una System Call viene vista dal codice applicativo come una semplice funzione di librerie, ma, in realtà la sua reale implementazione è tipicamente "MACHINE DEPENDENT", in quanto lavora a livello firmware. Quindi, richiede una specifica programmazione in Assembly (ASM) per poter far uso di istruzioni di trap e istruzioni di manipolazione dei registri di CPU.

• DA CODICE C VERSO CODICE MACCHINA:

Compilando come file oggetto (-c) con "gcc -c -fomit-frame-pointer":

```
int f(int x) {
    if (x==1) return 1;
    return 0;
}
```



0000 0000 0000 0000 <f>:

0:	89 7C 24 fc
4:	83 FC 24 fc 01
9:	75 07
b:	b8 01 00 00 00
10:	eb 05
12:	b8 00 00 00 00
17:	c3

<i>f(x==1)</i>	mov %edi, -0x4(%rsp)
	cmpb \$0x1, -0x4(%rsp)
	jne 12 <f+0x12>
	mov \$0x1, %eax
	jmp 17 <f+0x17>
	mov \$0x0, %eax
	retq

• ALCUNE NOTAZIONI x86 :

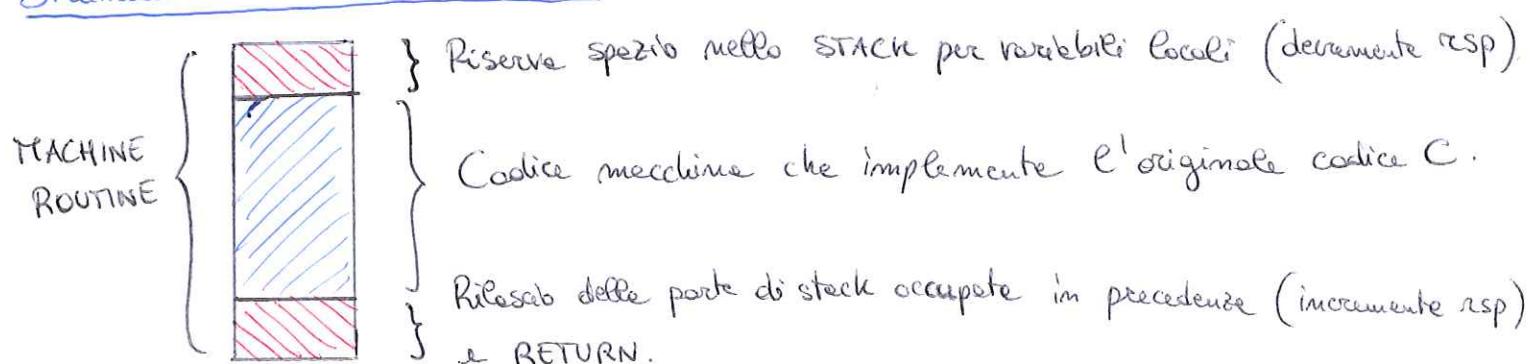
rsp : register STACK POINTER

edi : registro di CPU general purpose (per uso generale)

retq : control return to caller (PC salvato nello stack)

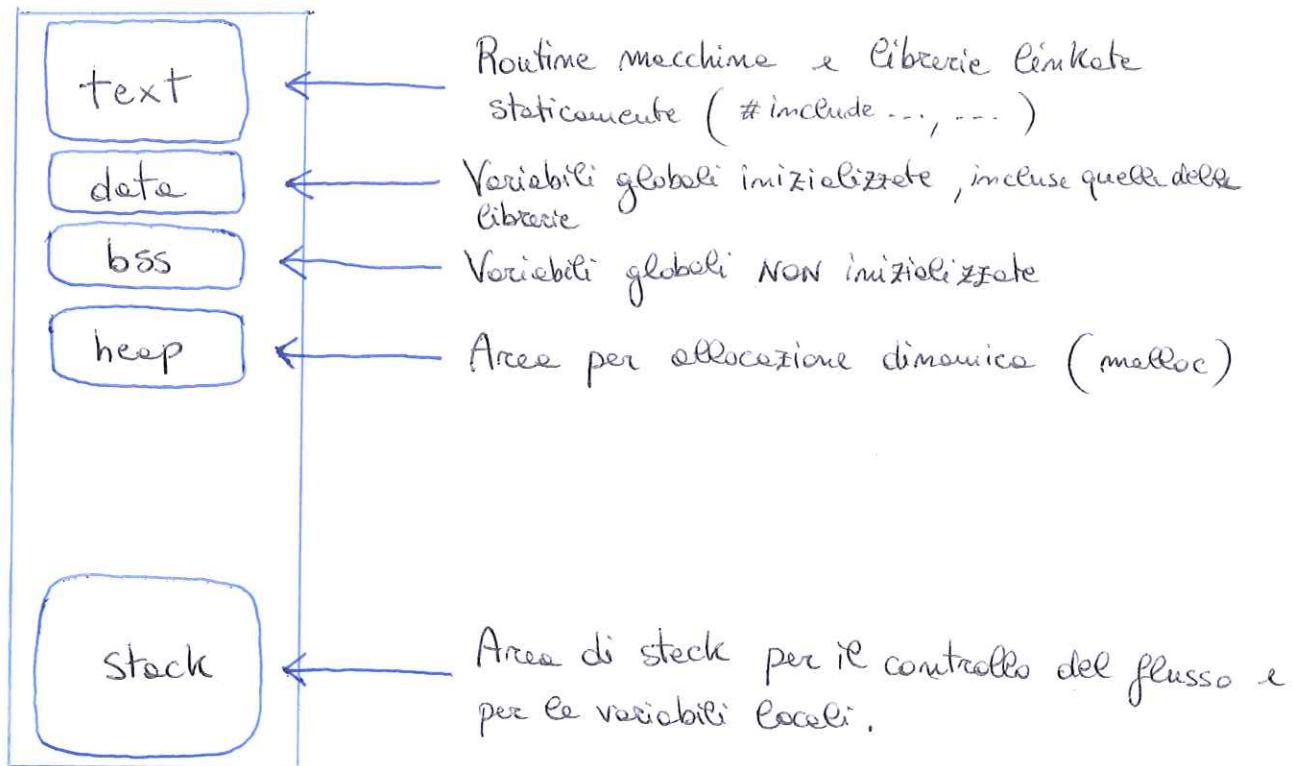
mov : istruzione per spostare dati

Strutturazione del codice macchina:



• ADDRESS SPACE , SPAZIO DI INDIRIZZAMENTO:

Ogni qual volta viene compilato un codice applicativo, viene inizializzato il cosiddetto ADDRESS SPACE, che contiene, all'interno della memoria RAM, tutte le informazioni necessarie all'esecuzione. Esso è una sorta di CONTENITORE e, cosa importante, ogni locazione di memoria è individuata ed accedute in base ad un OFFSET a partire dall'inizio del contenitore.



- ad ogni funzione C compilata e linkata staticamente corrisponderà un'unica routine di istruzioni macchine collocate nella sezione "TEXT";
- ad ogni variabile globale dichiarata dal programmatore o all'interno di librerie linkate staticamente corrisponderà una locazione nella sezione "DATA";
- La sezione "HEAP" permette uso di ulteriore memoria, per esempio tramite la libreria malloc (la stessa malloc tiene traccia di quali aree dell'heap vengono consegnate in uso ed un'opp).

La sezione HEAP è esponibile fino alla SATURAZIONE dell'intero ADDRESS SPACE.

Esempio:

int x = 10; → DATA
char V[1024]; → BSS

```
void function (void) {
    int x;
    ...
    return 0;
}
```

Codice macchine corrispondente → TEXT

STACK (un'istante per ogni attivazione di funzione)

I FORMATTI

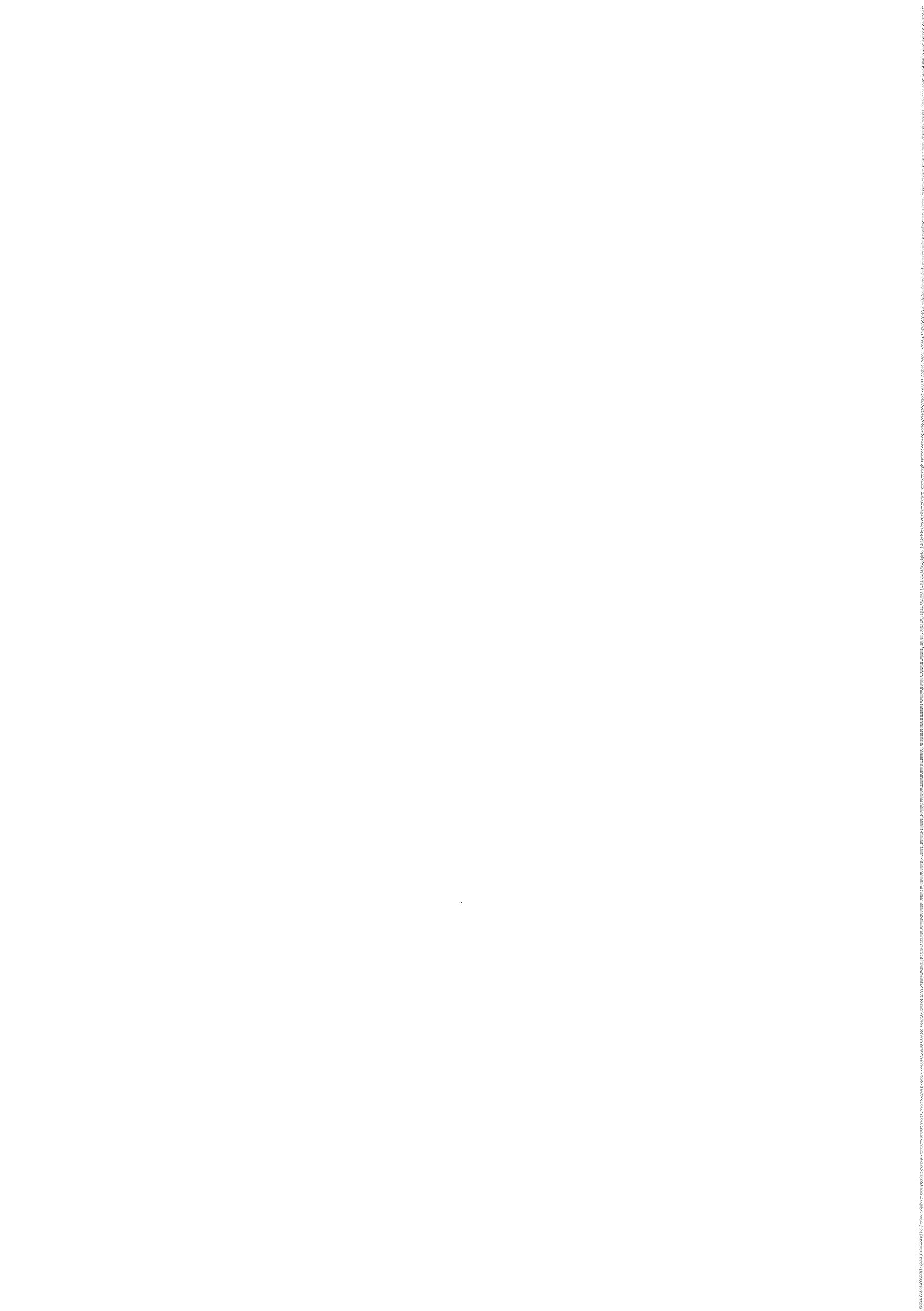
- In sistemi UNIX la strutturazione delle varie sezioni per ogni singolo programma è specificata tramite un formato chiamato ELF (Executable and Linkable Format).
- In sistemi WINDOWS la strutturazione è specificata tramite un formato chiamato PE (Portable Executable), basato su COFF (Common Object File Format), tramite il quale si descrive il contenuto effettivo di un file exe.
- È compito dei tool di compilazione (per es. GCC) generare ELF/exe a partire dai sorgenti del programma e delle direttive di compilazione fornite tramite l'uso di appropriati flag:
 - Specifica di quali moduli vanno inclusi in compilazione
 - Specifica di quali sezioni (per esempio mon di defaceit) debbono essere incluse

VARIABILI PUNTATORE

Possono registrare il valore di un indirizzo di memoria, che altro non è che uno SPIAZZAMENTO all'interno dell'ADDRESS SPACE dell'applicazione.

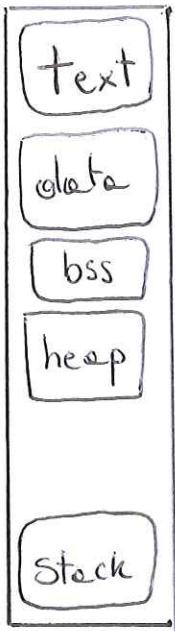
```
int *x;  
float *y;  
double *z;  
  
void function(void){  
    x = y = z;  
}
```

SI PUO' FARE!
Genera solo un warning che, su gcc,
si inhibisce con -Werror



1. INTRODUZIONE

• ADDRESS SPACE:



- ← Istruzioni macchine e librerie linkate staticamente
- ← variabili globali inizializzate } comprese quelle delle librerie
- ← variabili globali NON inizializzate
- ← Aree per allocazione dinamica (malloc, mmap, ...)
- ← Aree di Stack: variabili locali, controllo di flusso,

- Di norma è introdotta la RANDOMIZZAZIONE dell'ADDRESS SPACE e spiegamento randomico: in tal modo indirizzi molti a tempo di compilazione non coincidono con quelli effettivi a tempo d'esecuzione, rendendo il codice meno vulnerabile ad eventuali attacchi.

2. PROCESSES AND THREADS

- PROCESSO: entità dinamica costituita da un insieme di tracce d'esecuzione (threads) e formato quindi da CODICE ESEGIBILE + INFORMAZIONI DI GESTIONE

Il SO mantiene una TABELLA DEI PROCESSI attivi per poterli gestire.

- IMMAGINE DI PROCESSO: programma + dati + stack + STACK DI SISTEMA + PCB

PCB (Process Control Block): informazioni, METADATI per la gestione del processo

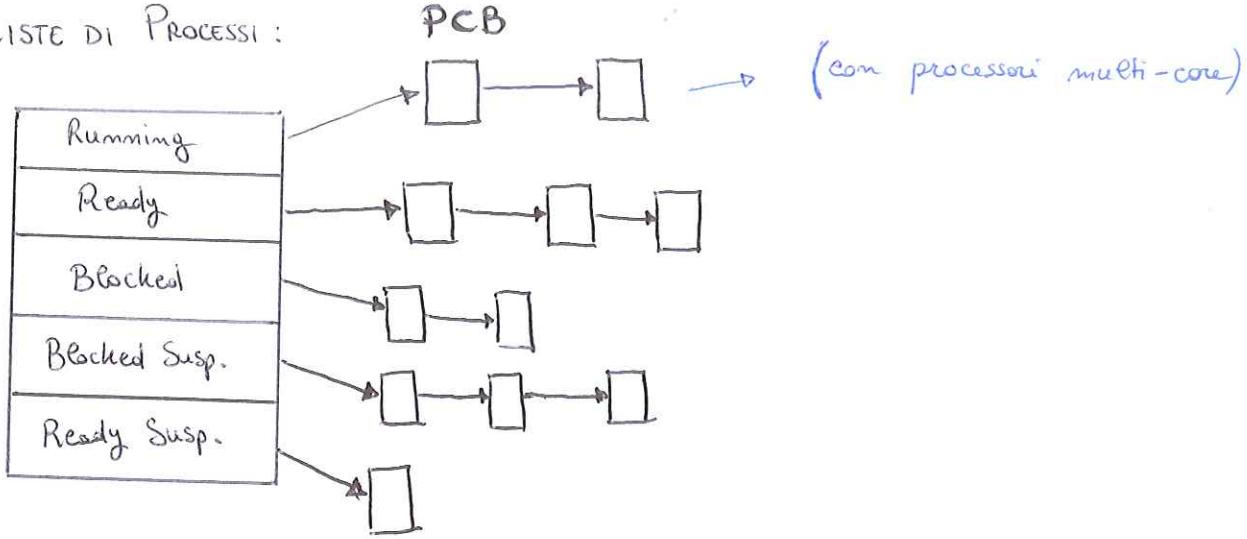
* Anche se il processo è SWAPPED OUT, PCB e STACK DI SISTEMA sono sempre mantenuti in memoria di lavoro (RAM)

PCB contiene:

- pid proprio; pid di processi relazionati
- stato processo (RUNNING, BLOCKED, ...)
- permessi d'accesso
- snapshot di CPU (PC, SP, registri di processore, ...)

- | - Info di Scheduling: priorità, effinità, ...
- | - Info di Stato: evento atteso

• LISTE DI PROCESSI:



- CAMBIO DI MODO D'ESECUZIONE: quando passo dall'eseguire in modo USER al modo KERNEL (alto privilegio) e viceversa.

CAUSE:

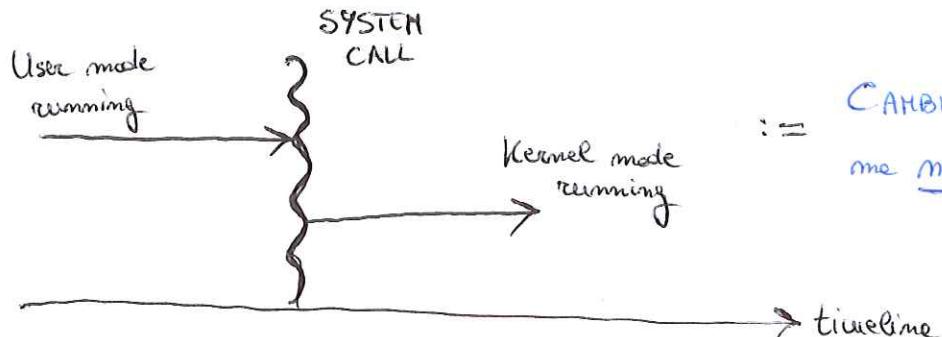
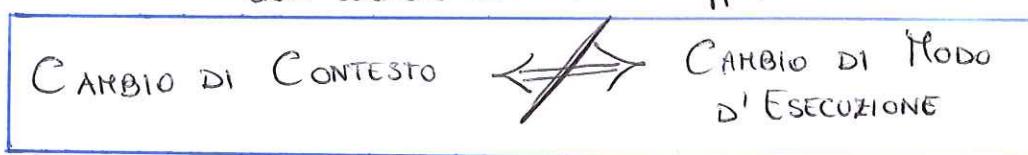
- system call e relative trap al Kernel;
- gestione di routine di interrupt da I/O.

• CAMBIO DI CONTESTO:

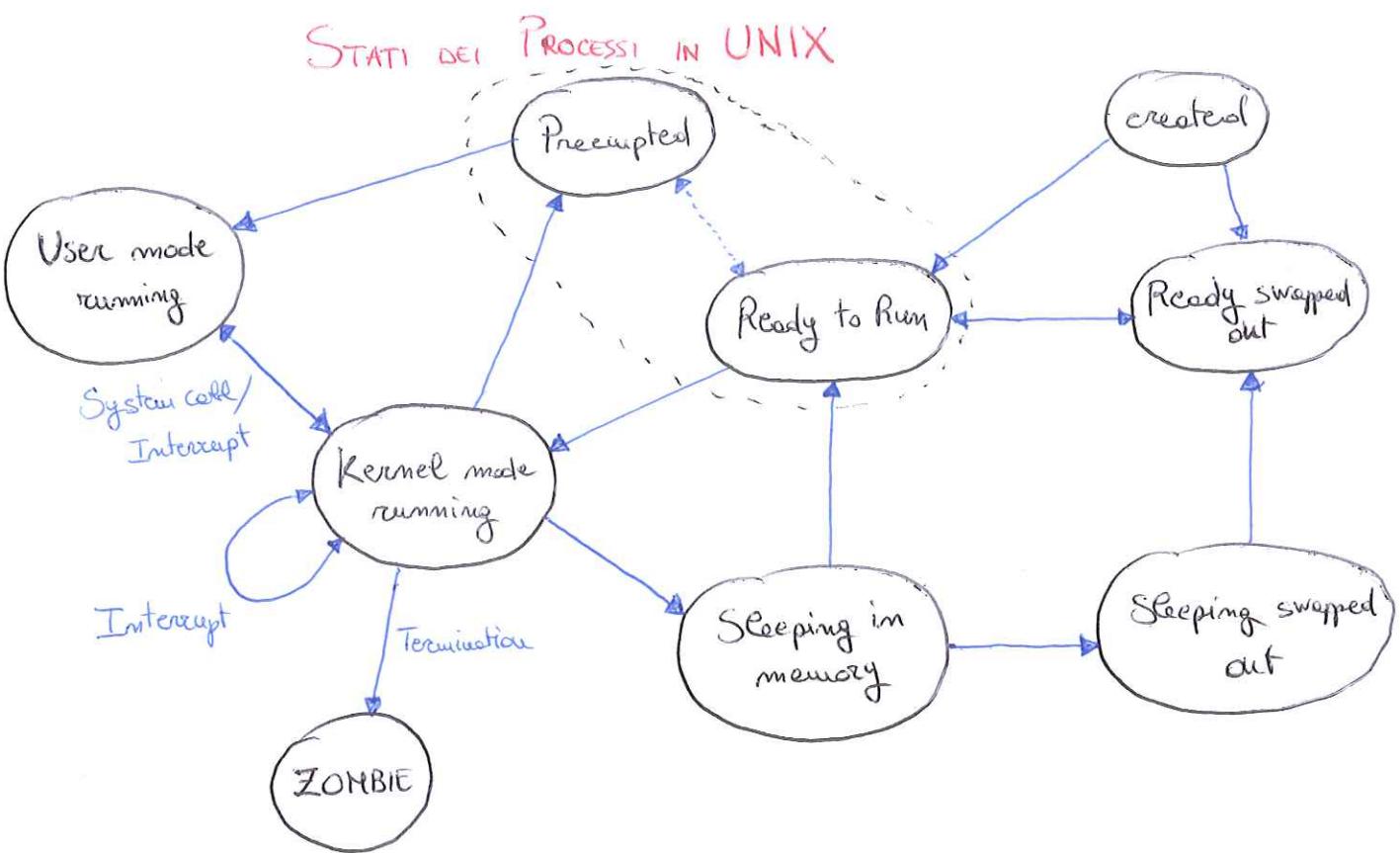
quando vengo disceso dall'essere running o wait ed ho un CAMBIO DI STATO.

CAUSE:

- interrupt da timer, smetto di essere running;
- interrupt da I/O, vengo messo wait;
- fault di memoria (per memoria virtuale) quando vengo deattivato o devo caricare memoria swapped out.

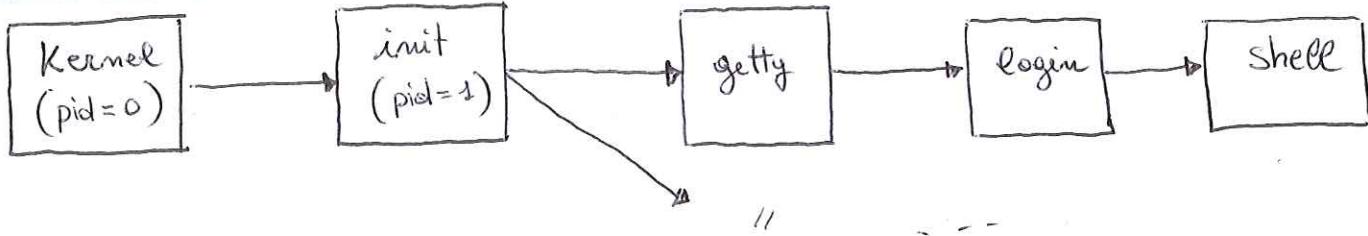


:=
CAMBIO DI MODO,
ma non di contesto



- User mode running \leftrightarrow Kernel mode running : CAMBIO DI MODO
- ZOMBIE: alla terminazione viene mantenuto il PCB fin quando qualcuno non richiede il codice di terminazione del processo.
- Preempted: è avvenuto e previsibile e cause di interrupt da timer.

• AVVIO DEL SISTEMA:



- Kernel (pid=0): detto anche IDLE PROCESS fa l' AUTOSETUP delle STRUTTURE del KERNE^L
- init (pid=1): ora è "systemd"; Legge info da /etc/inittab e /etc/ttysTAB
- getty: get terminal \rightarrow 1 per ogni terminale da gestire

• System call execX():

Chiamano internamente execve(), unica vera System call

→ IL PROCESSO su cui esegue le exec ricevute lo stesso! Ma esegue un NUOVO PROGRAMMA, sostituendo le informazioni e i dati nelle' address space.

#include <unistd.h>

```
int exec (const char* path, [char* argv[0], ..., char* argv[N],] 0)
```

gli argomenti devono terminare con NULL;

cerca il nome del programma path tra le variabili d'ambiente, in particolare im PWD (Process Working Directory)

• execvp(): fa lo stesso, ma cerca nella variabile d'ambiente PATH.

```
int execvp (const char* file, const char* argv[], const char* envp[])
```

Stavolta passa un pointer sull'ARRAY di argomenti;

permette di passare anche un pointer ad un ARRAY di VARIABILI D'AMBIENTE.

* Nota: la clonazione dovuta alla fork() permette di ereditare tutte le variabili ambientali del processo padre.

AMBIENTE DEL KERNEL:

Le variabili d'ambiente sono utilizzate solo in User Space; il Kernel mantiene solo le variabili: PWD e ROOT.

PROCESSI IN WINDOWS

PROCESSO: OGGETTO di tipo processo, con i suoi attributi (metodati) e servizi

Ogni processo ha una **TABELLA DEGLI OGGETTI** per processo, con le quale, tramite HANDLE, eccede agli oggetti cui il processo ha il permesso d'accesso.

- Creatzione: `CreateProcess (...)`

Viene mappato su `CreateProcessA (...)` → ASCII } A seconda di com'è definito
`CreateProcessW (...)` → UNICODE } il tipo di elenco TCHAR

Se definisco la macro _UNICODE: TCHAR si mappa su wchar_t

* Definendo la macro _CRT_SECURE_NO_WARNINGS posso usare funzioni poco sicure o deprecate come `scanf()` o `gets()`; di quest'ultime dovrà dichiarare prima il prototipo.

- Attese: `\XWaitForSingleObject (HANDLE hHandle, DWORD dwMilliseconds)`

* Serve per Sincronizzare processi! Ma, a differenza di UNIX, non restituisce il valore di ritorno! (UNIX: `wait (&status)`)

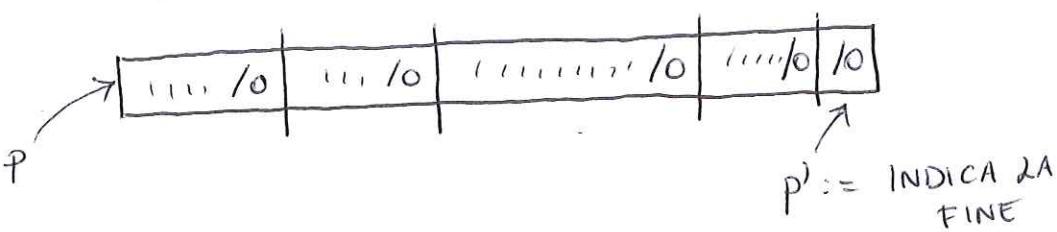
- CODICE TERMINAZIONE: `int GetExitCodeProcess (HANDLE hProcess, LPDWORD lpExitCode)`

Non è BLOCCANTE! Prende solo il codice di terminazione, altrimenti ritorna "STILL_ACTIVE".

- VARIABILI D'AMBIENTE: `2PTCH WINAPI GetEnvironmentStrings (void)`

Si tratta di un puntatore! In UNIX: `char** environ` → ARRAY di puntatori

Questo perché in Windows le info ambientali sono imprecate così:

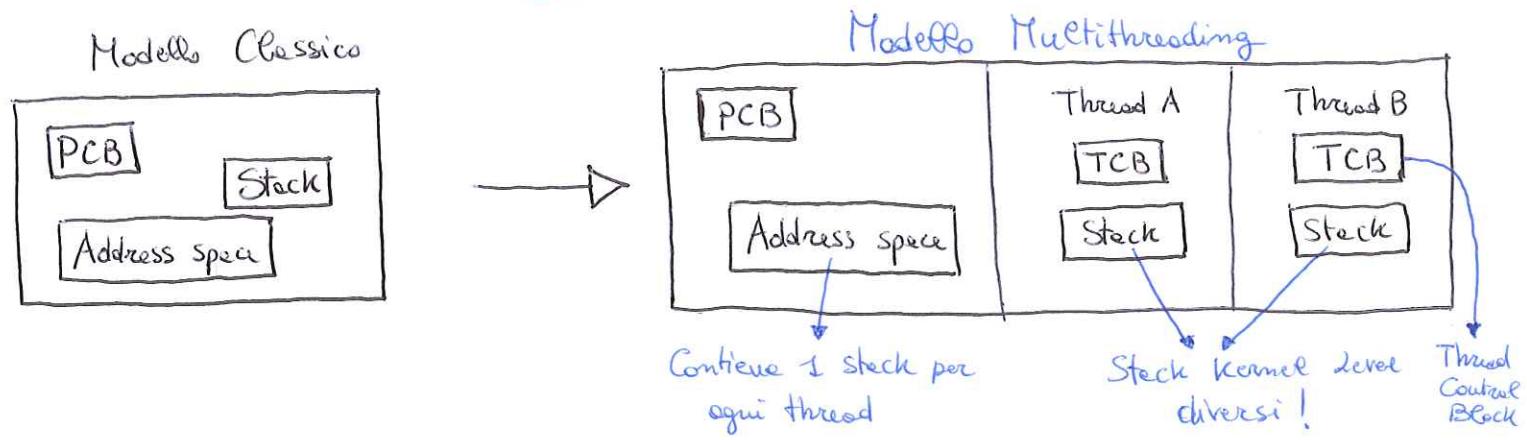


E' la regola di TOKENIZZAZIONE (`strtok()`)!

I THREAD

- THREAD: tracce d'esecuzione; è l'UNITÀ BASE per il DISPATCHING.
Un processo può essere visto come un insieme di threads, ognuno con le proprie tracce d'esecuzione → MULTITHREADING

- * Si può lavorare con tracce multiple, ma nello **STESO ADDRESS SPACE**: bisogna gestire lo scheduling per le concordanze dei vari threads, in modo tale che i dati e le informazioni dello stack e del contenitore siano COERENTI.
→ Per garantire ciò, **per ogni thread** uso un' area di STACK DIFFERENTE



- * È importante avere STACK KERNEL SEPARATI, poiché, se sto eseguendo su 2 CPU-cores diversi e 2 thread chiamano simultaneamente una system call, bisogna gestirle separatamente (il Kernel è 1 solo).

- * Ciò che i diversi thread vedono IN COMUNE sono le VARIABILI GLOBALI!
Tuttavia, un aggiornamento di tali variabili richiede più istruzioni macchina, quindi non ha ATOMICITÀ → potrei avere Problemi di Consistenza!

• VARIABILI TLS:

TLS := Thread Local Storage

Sono variabili globali, ma mi viene creata un'ISTANZA per ogni THREAD, in quale lavorerò solo sulle proprie istanze;

Posix : __thread <var.name>

Windows: __declspec(thread) <var.name>

THREAD IN WINDOWS

Windows mette già indirizzato al multithreading, quindi ogni thread è un OGGETTO con un proprio ID, un proprio HANDLE e i propri metadati.

- Creatione: `HANDLE CreateThread (...)`
- Terminazione: `VOID ExitThread (DWORD dwExitCode)`
`int GetExitCodeThread (...)`
- Attese: sempre le solite WaitForSingleObject (...)

THREAD IN UNIX

Non implementano direttamente il multithreading, quindi sono visti solo come processi più leggeri.

LWP - Light Weight Process: i metadati restano quelli del processo, dovrò solo aggiungere stack e metadati (per scheduling o altro) per ogni thread.

include <pthread.h> ; compile with -lpthread

- Creatione: `int pthread_create (pthread_t *tid, pthread_attr_t *attr, void *(*funct) (void *), void *arg)`

funct (void*) : modulo software del thread ; prende come parametri un puntatore a void.

* pthread_t è un unsigned int !

- Terminazione: `Void pthread_exit (void * status)`

ATTENZIONE! Si posse un puntatore (costato a void) al codice di terminazione

- Attese: `int pthread_join (pthread_t tid, void ** status)` → E' la wait() per i thread

- TID: `pthread_t pthread_self (void)`

Esistono threads di livello Kernel, che girano solo in Kernel mode; questi sono detti **DEMONI KERNEL**. Alcuni esempi sono:

- è l' IDLE PROCESS : fa l'autosetup delle strutture del Kernel (pid=0)
- kswapd : gestisce lo swapping di threads e processi.

3 - CPU SCHEDULING

Scheduling:

- A lungo termine : sull'aggiunta o meno di un processo tra quelli attivi
 - A medio termine : se aggiungere o meno un ^{*}address space in RAM per un processo (swapping)
 - A breve termine : come assegnare le CPU ai processi in stato ready (CPU - scheduling)
 - I/O scheduling : come schedulare le richieste ai dispositivi di I/O
- FCFS (First Come First Served) :

Unisce READY QUEUE con sistemi di servizio di tipo FIFO rispetto all'entrata dei processi nello stato ready.

- PROBLEMI :
1. Non c'è preemption
 2. ~~Processi I/O bound causano sottoutilizzo delle CPU~~
processi CPU bound causano sottoutilizzo dei dispositivi di I/O
 3. Non minimizza il tempo d'attesa

• ROUND ROBIN :

"Ceroneffo" → un po' a testa ; viene introdotto il QUANTO DI TEMPO, (timeout), quindi stasera c'è PREEMPTION!

- PROBLEMI :
1. Sfiorisce processi I/O bound, poiché usano una minima parte del quanto di tempo
 2. Di conseguenza sfiorisce opp interattive e c'è sottoutilizzo dei dispositivi di I/O
 3. Criticità nelle scelte del quanto di tempo

• ROUND - ROBIN VIRTUALE:

Abbiamo 2 Ready Queues :

PRELACIONATI : (priorità bassa)

BACK FROM I/O : (priorità alta)

Venne introdotto il concetto di PRIORITÀ, strettamente correlato alle code multiple.

Se un processo ha eseguito il prelascio, ha usato tutto il suo quanto di tempo Δ e viene messo a bassa priorità.

Se invece un processo ha usato solo X del quanto di tempo Δ per interagire con un dispositivo di I/O, verrà messo a priorità alta, ma il suo QUANTO DI TEMPO sarà $\underline{\Delta - X}$.

• SPN - Shortest Process Next:

CPU BURST := intervalli di tempo in cui viene occupata la CPU da parte di un'opp.

Processi mandati in esecuzione in ORDINE CRESCENTE di "CPU burst"

→ Minimizza il tempo d'attesa e il tempo di turnaround, ma non c'è preemption:

VARIANTE: SRTN - Shortest Remaining Time Next :

Si esegue la CPU non in base al CPU burst, bensì sulle quantità rimanente di quanto di tempo assegnato ($\Delta - X$ minore esegue prima).

Se $X = 0 \rightarrow$ quanto di tempo = $\Delta \rightarrow$ c'è PREEMPTION!

* È relativamente adeguato per processi interattivi.

PROBLEMI: 1. Richiede previsione del "CPU burst" o del "Remaining Time"; sbagliare le previsione può portare a criticità.

2. Può causare STARVATION ai processi con CPU burst molto lunghi.

Stime dei CPU BURST:

• media aritmetica: $S_{m+1} = \frac{1}{m} \cdot \sum_{i=1}^m s_i$, dove s_i è le stime dei CPU burst

• media esponenziale: $S_{m+1} = \alpha T_m + (1-\alpha) S_{m-1}$

con $\alpha \rightarrow 1$, dà più peso alle osservazioni recenti.

• HRRN - Highest Response Ratio Next:

Serviti processi con RAPPORTO DI RISPOSTA (RR) maggiore:

$$RR = \frac{W + S}{S}$$

W := tempo di attesa nello stato ready

S := tempo di servizio (esecuzione)

Non crea STARVATION; Favorisce processi I/O bound, poiché S è molto piccolo.

- PROBLEMI:
1. Richiede sempre PREDIZIONE del tempo di servizio S.
 2. Bisogna misurare il tempo W per ogni processo ready.

• MULTI-LEVEL FEEDBACK-QUEUE:

- FEEDBACK: cambio di classe di priorità, dovuto allo stato da cui proviene il processo e dell'utilizzo o meno della PREEMPTION.

- Use CODE DI PRIORITA' multiple \rightarrow N livelli di PRIORITA'
- Se un processo utilizza tutto il suo quanto di tempo e fa prelascio, viene declassato alla classe di priorità inferiore (feedback negativo)
- Quanto di tempo FISSO per ogni classe di priorità'

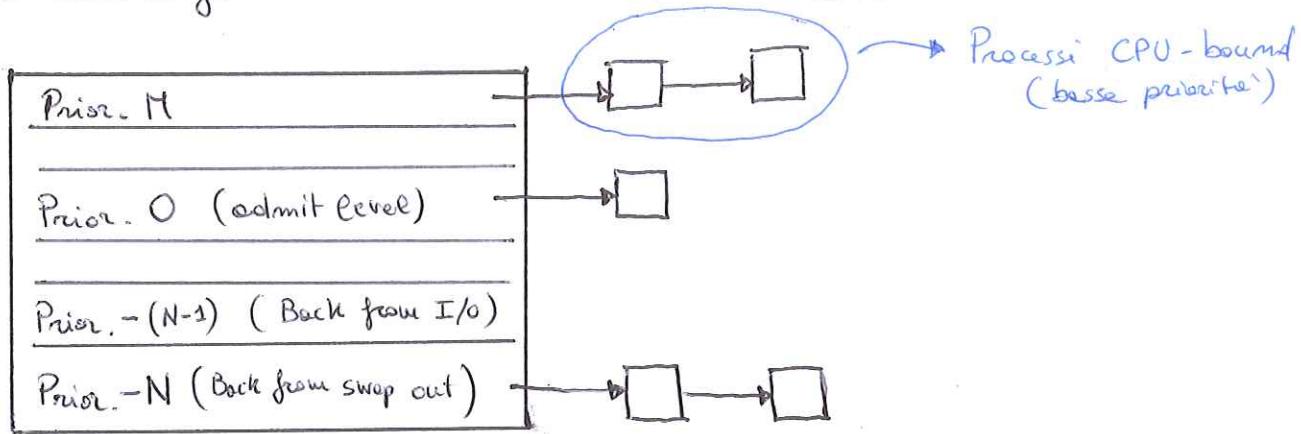
* Favorisce processi I/O bound, che hanno priorità' migliore

- PROBLEMI:
1. STARVATION sui processi molto lunghi che vengono declassati.

\rightarrow Soluzione parziale: quanto di tempo $\Delta_i = 2^i$, dove i è il livello di priorità (i cresce con priorità' più bassa).

SCHEDULER UNIX TRADIZIONALE

- Multi-Level Feedback Queue
- Un livello di priorità per ogni codice
- Ogni codice è gestito interamente con politica ROUND-ROBIN



Passa da una codice all'altra (feedback) se:

1. rientra Ready dopo essere stato Sleep (o Blocked)

2. variazione di priorità imposte dal Sistema periodicamente:

$$P = \text{base} + \frac{\text{CPU}}{2} + \text{mice}$$

Dove:
base = priorità dell'admit level (0)

$\frac{\text{CPU}}{2}$ = normalizzazione del tempo di CPU usato in precedenze

mice = niceness del processo (più è basso, maggiore è la priorità)

• Ho 40 livelli di priorità, da -20 a +19!

SYSTEM CALL:

- 1) int nice (int inc) : incrementa la niceness di inc e $\in [-20, +19]$
- 2) int getpriority (int which, int who) : which è PRIO_PROCESS o PRIO_PGRP o PRIO_USER
- 3) int setpriority (int which, int who, int prio)
- 4) >>> renice -n <nice> -p <pid>

* Con SISTEMI MULTIPROCESSORE nasce anche il problema dell'assegnazione dei processi ai CPU-cores, sia per quanto riguarda app User Level che Kernel Level:

[USER]: Policy DINAMICA: assegnazione a processori diversi durante tutte le fasi del processo.

Maggior efficienza dei CPU-cores, ma maggior overhead

[KERNEL]: Approccio PEER: il software di livello Kernel può essere eseguito su qualsiasi CPU-core.

Tuttavia aumenta la complessità di progetto e bisogna gestire anche problemi di coerenza sui dati.

SCHEDULING DI THREADS

I thread sono **INTERATTIVI** tra loro! A seconda dei singoli thread che lo compongono, un processo può essere dell'essere CPU-bound od I/O bound e viceversa più volte. Ciò è dovuto anche alla SINCRONIZZAZIONE tra thread.

• Politica LOAD SHARING:

- CODA GLOBALE UNICA con gestione delle **PRIORITÀ** per threads ready
- esecuzione possibile su tutti i CPU-cores
- con macchine altamente parallele (multi CPU-cores) aumenta di molto le probabilità di dover ottenere la sincronizzazione
→ problemi di **SCALABILITÀ** delle code globali e poche efficienze

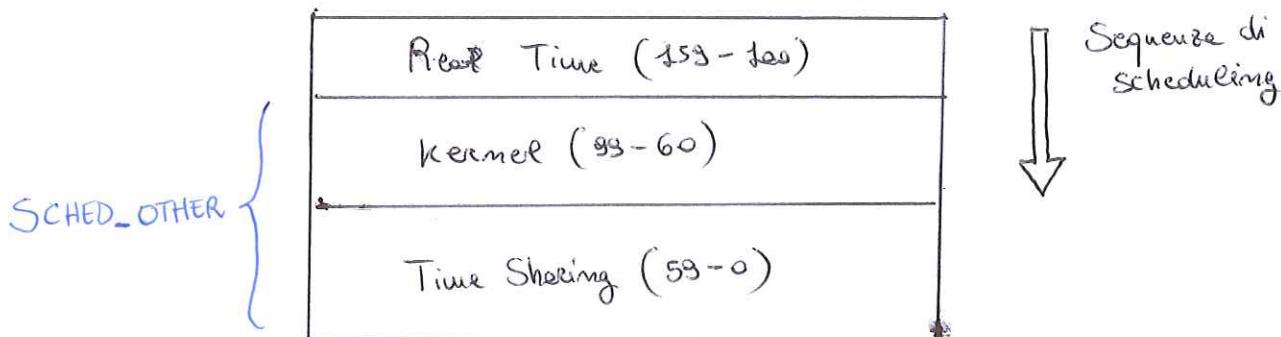
• Politica LOAD BALANCING:

- CODA MULTIPLE, una per ogni CPU-core
- migliore scalabilità
- possibilità di sovraccaricare una CPU o scarico di altre, ma risoltivo con REBALANCING PERIODICO di PCB o TCB sulle code seguendo un algoritmo di ribilanciamento del carico.

SCHEDULING UNIX SVR4

System Fire - Release 4.

- 160 livelli di priorità e 3 classi : REAL TIME (159-100), KERNEL (99,60), TIME-SHARING (59-0)
- Anche il Kernel è PREEMPTABLE, ma la PREEMPTION avviene solo in SAFE PLACES
- Uso di bitmap per identificare livelli non vuoti
- QUANTO DI TEMPO variabile in funzione di classe e livello.



Scheduling LINUX attuale:

100 livelli di priorità; un livello può rappresentare una CLASSE.

REAL TIME (RT) 99

TIME-SHARING 0 → classe SCHED_OTHER

Il vari livelli differiscono tra loro in base alla NICENESS.

SYSTEM CALL:

#include <sched.h>

REAL-TIME o SCHED-OTHER

- 1) int sched_setscheduler (pid_t pid, int policy, const struct sched_param *p)
- 2) int sched_getscheduler (pid_t pid)
- 3) >>> chrt -p <pid> <prio>, da chiamare con "sudo"

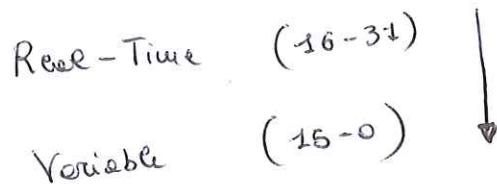
EPOCHE DI SCHEDULUNG

È un periodo di tempo per l'operatività del Sistema, prima di riorganizzarla.
All'inizio di ogni epoca, ai ogni thread vengono assegnati dei quanti di tempo.
Thread figli prendono parte dei quanti di tempo del padre.

SCHEDULER WINDOWS

Abbiamo 32 livelli di priorità e 2 classi: REAL-TIME e VARIABLE;

Ogni livello è una CODA MULTIPLA con gestione di tipo ROUND-ROBIN;



La PREEMPTION è basata sul livello di priorità:

- priorità BASE: di riferimento per i PROCESSI
- priorità DINAMICA: per i THREADS, funzione di quella base

* FEEDBACK: solo nella classe VARIABLE !!!

Nella classe Real-Time non c'è possibilità di cambiare livello di priorità!

System Calls:

- 1) SetPriorityClass (...)
- 2) SetThreadPriority (...)