Sistemi Operativi 1

Davide Cerotti

- ***Contatti:***
  - ***email: davide.cerotti@uniupo.it***

- ***Ricevimento su appuntamento***
  - ***martedì o giovedì mattina o dopo le 16 previo accordo via mail***
  - ***a distanza tramite Meet previo accordo via mail***

Obiettivo del corso di Sistemi Operativi (1 e 2): insegnare i concetti di base dei moderni sistemi operativi (non gli aspetti relativi ai sistemi in rete e distribuiti)

Sistemi Operativi 1 (6 CFU) :

- **Concetti generali sui sistemi operativi**
- **Processi e thread (gestione della CPU, sincronizzazione)**
- **Gestione della memoria**
- **Attività pratica su Unix (Linux) con particolare interesse per la programmazione concorrente**

Sistemi Operativi 2 (II semestre, 6 CFU) è dedicato a:

- **Gestione dell' I/O**
- **File system**
- **Virtualizzazione**

Libro di testo* (per SO 1 e 2)*:

A.S. Tanenbaum. H. Bos, *Modern Operating Systems*, 4th ed, Pearson, 2015, ed. Italiana *I Moderni Sistemi Operativi*, 4a ed. 2016

L'accesso alla pagina del corso su DIR è senza chiave di iscrizione

Esame

Sono previsti:

- **Due appelli scritti nella prima sessione**
- **Gli altri 4 appelli orali**

Per SO da 12 CFU il voto è unico per SO1 e SO2

L'esame è anche sugli argomenti introdotti per le esercitazioni in laboratorio

Per accedere all'esame sarà richiesta la consegna su DIR di alcuni degli esercizi

Consegnare vuol dire «ho capito questa cosa e all'esame posso rispondere su questo argomento»

Il regolamento didattico di ateneo prevede di non sostenere più di 3 volte l'anno uno stesso esame

Esame

È un esame difficile? No, e recentemente i risultati sono migliori. Però:

- **Le lezioni servono**
  - **ma serve meno seguire una lezione si e una no; se si perde una puntata è difficile capire il «seguito»**
  - **serve meno seguire con il cervello «spento» o altrove, tanto sono cose banali, tanto c'è scritto sulle slides... Ma è facile fare confusione, credere di aver capito**
- **Gli esercizi in laboratorio servono**
  - **non guardare un compagno che li fa**
- **Studiare serve**
- **6 crediti = 150 ore per lo studente (di cui 48 di lezione)**

Esame

- **Seguire e capire:**
  - **Perché i S.O. sono fatti così? Il perché non è un *optional*: vediamo cosa c'è nei S.O. *general purpose* (per desktop, laptop, servers) e come ci si è arrivati, per quali esigenze (che non ci sono sempre), dove serve supporto dall'*hardware...***
  - **Ma i S.O. costano: le idee bisogna realizzarle, bisogna che *l'hardware* fornisca supporto (che costa); il S.O. usa risorse del sistema (CPU, memoria, dispositivi)**
  - **Il perché serve anche a sapere quando possiamo o dobbiamo rinunciarci (non servono e/o costano troppo)**
- **Fare gli esercizi di laboratorio**
  - **Che cosa ho imparato da (o verificato in) questo esercizio?**
- **Studiare**
  - **Le nozioni bisogna conoscerle (se no di cosa si discute?)**
  - **L'idea mi ha convinto? Perché? Saprei convincere un altro che è una buona idea evidenziando perché o in quale caso serve?**

Introduzione

Esempi di sistemi operativi

- Windows (95, 98, NT, 2000, XP, Vista, 7, 8, 10)
- Unix-like:
  - Linux (Varie distribuzioni)
  - MacOS X
  - FreeBSD
  - Solaris
- Sistemi operativi Real Time
  - RT-Linux
  - …
- Sistemi per dispositivi mobili
  - iOS (iPhone, iPad)
  - Android
  - …

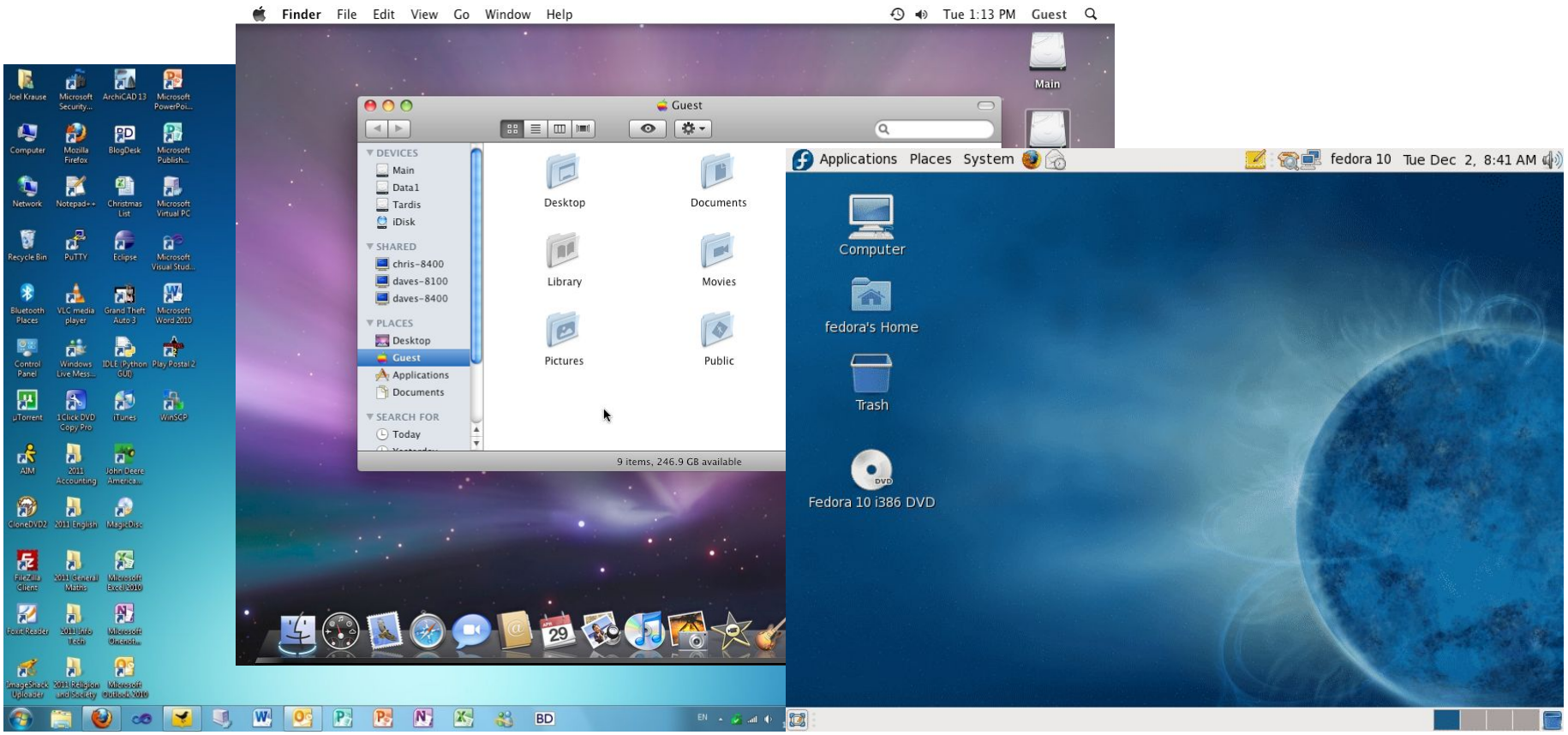
  - Sistemi operativi per automobili
  - vw.os*

* https://arstechnica.com/cars/2019/09/volkswagen-audi-porsche-vw-group-plans-one-os-to-rule-them-all/

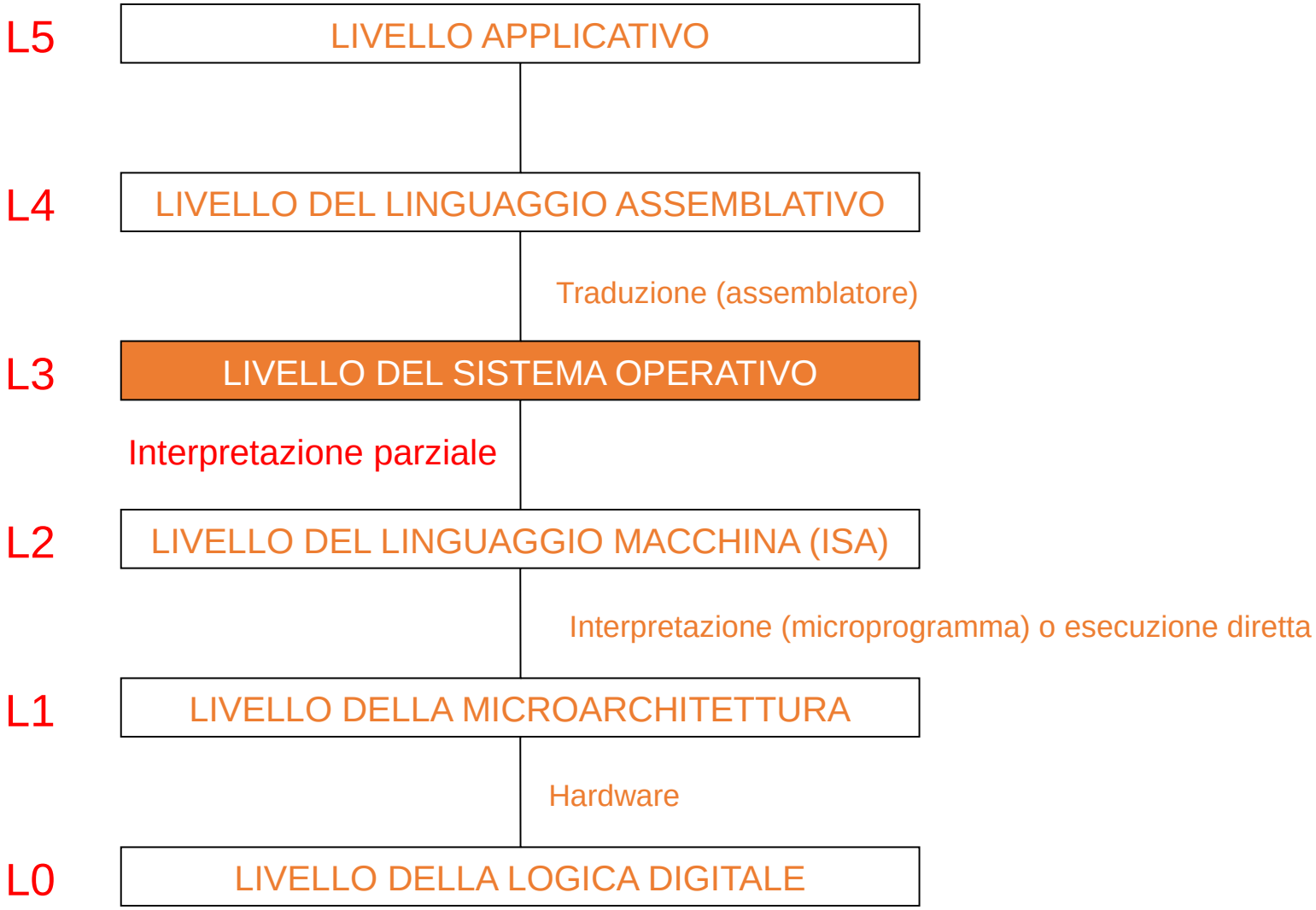
Cosa intendiamo per sistema operativo

Non l'interfaccia grafica, che si può cambiare (a meno di questioni di brevetti), e l' insieme di applicazioni disponibili

Ciò conta molto nel successo di un S.O., ma gran parte degli aspetti riguardano la disciplina "Interazione Uomo-Macchina" (che non riguarda solo i S.O.)



Il SO nella gerarchia dei livelli



Realizzazione di L2 su L1

Da L1 a L2 (quando non coincidono):

- **L2 realizzata su L1 grazie ad un MICROINTERPRETE**
  (che esegue ripetutamente prelievo, decodifica ed esecuzione di istruzioni del linguaggio L2)
- **Cosa offre in piú ?**
  - **punto di vista piú astratto dell'architettura**
  - **istruzioni piú "potenti" *(nelle architetture CISC)***
  - **CHIAMATA DI PROCEDURE**
    (astrazione: una procedura come un'istruzione)

Realizzazione di L3 su L2

Da L2 a L3:

- **L3 realizzata su L2 tramite il *software* del sistema operativo; L3 *comprende tutte le istruzioni di L2 piú altre* (system call) che permettono di accedere alle funzioni aggiuntive**
- **Cosa offre in piú ?**
  - **punto di vista piú astratto della macchina**
  - **nuove "istruzioni" per lanciare più programmi da eseguire contemporaneamente, per tenere conto di utenti diversi, per creare, modificare, leggere file (contenenti dati o programmi) e raccoglierli in directory (cartelle), per usare facilmente dispositivi (dischi, stampanti, telecamere, …), comunicare tramite la rete con altre macchine**

Macchina virtuale di livello 3

Cosa intendiamo per sistema operativo

PUNTO DI VISTA DEL PROGRAMMATORE: FACILITÀ D'USO

- **Macchina Virtuale di livello 3: estende il livello sottostante fornendo operazioni astratte per l'interazione con i dispositivi di I/O e per l'esecuzione contemporanea di più programmi**

PUNTO DI VISTA DEL SISTEMA: GESTORE/CONTROLLORE

- **Gestore di risorse: coordina l'uso *condiviso* delle risorse del sistema (CPU, memorie, dischi, stampanti, ecc.) da parte di *più programmi* (in esecuzione *simultanea*) cercando di utilizzarle al meglio:**
  - **minimizzare i tempi morti (risorsa non utilizzata)** ≠ !!
  - **minimizzare le attese distribuendole in modo «adeguato» al tipo di applicazione (per lo meno, evitare che una applicazione non vada avanti)**
- **Controllore: evita che un programma in esecuzione acceda ad informazioni di altri programmi in esecuzione o del sistema operativo, compromettendone la riservatezza o il buon funzionamento (ma per esercitare questa funzione il SO ha bisogno di girare)**

Cosa intendiamo per sistema operativo

Sistemi Operativi diversi:

- **forniscono insiemi di chiamate di sistema diverse (non necessariamente, es. tutti gli Unix-like forniscono un ampio sottoinsieme delle chiamate dello standard POSIX)**
- **sono progettati e realizzati in modo diverso**
- **possono quindi svolgere il ruolo di gestore e controllore in modo diverso**

Intermediario tra programmi e risorse

***Risorse del sistema*: CPU, Memoria, Dischi, ...**

- **permette a diversi programmi di condividere la CPU, utilizzandola un po' ciascuno (time multiplexing)**
- **permette a diversi programmi di condividere la memoria suddividendola in sezioni separate assegnate a programmi diversi (space multiplexing)**
- **permette a diversi programmi di condividere lo spazio su disco suddividendolo in sezioni separate assegnate a utenti diversi (space multiplexing)**
- **permette a diversi programmi di condividere vari dispositivi di I/O (stampante, unità nastro) utilizzandoli un po' ciascuno (time multiplexing)**

Evitare interferenze:

garantire che i programmi utente in esecuzione :

- **non interferiscano con il sistema operativo**
- **non monopolizzino le risorse**
- **non interferiscano fra loro**

Intermediario tra programmi e risorse

***Risorse del sistema*: CPU, Memoria, Dischi, ...**

- **permette a diversi programmi di condividere la CPU, utilizzandola un po' ciascuno (time multiplexing)**
- **permette a diversi programmi di condividere la memoria suddividendola in sezioni separate assegnate a programmi diversi (space multiplexing)**
- **permette a diversi programmi di condividere lo spazio su disco suddividendolo in sezioni separate assegnate a utenti diversi (space multiplexing)**
- **permette a diversi programmi di condividere i dispositivi di I/O (stampante, unità nastro, ...) (time multiplexing)**

Evitare interferenze:

garantire che i programmi

- **non interferiscano con il sistema**
- **non monopolizzino le risorse**
- **non interferiscano fra loro**

È necessario l'aiuto dell'Hardware: due modalità d'esecuzione, modo kernel e modo utente

Cosa fa parte del sistema operativo

Nell'organizzazione classica del software di un sistema operativo, il S.O. vero e proprio è il cosiddetto "kernel" (nucleo) che viene *eseguito in modalità kernel*, a differenza del codice dei programmi utente che invece viene eseguito in *modalità utente*

In modalità kernel si possono eseguire istruzioni che in modalità utente non si possono eseguire; e si può accedere a indirizzi di memoria riservati al sistema operativo – compresi quelli dei dati del S.O.: i dati che utilizza per gestire il sistema di elaborazione

Si passa a modalità kernel e ad eseguire codice del S.O. :

- **Quando si effettua una *chiamata di sistema* (che proprio per il cambiamento di modalità differisce da una funzione di libreria)**
- **Quando un programma cerca di eseguire una azione non permessa incorrendo in una *trap* (tecnicamente, il meccanismo di chiamate di sistema e trap è lo stesso ma nel primo caso ci si fa "intrappolare" volontariamente)**
- **Quando c'è una *interruzione* hardware e si passa ad eseguire la routine di gestione dell'interruzione**

Cosa fa parte del sistema operativo

Il kernel comprende tipicamente componenti come la gestione dei processi, la gestione della memoria virtuale e del file system, i driver delle periferiche, ... di cui si tratta nei corsi 1 e 2.

Peraltro, nei S.O. con architettura microkernel, solo alcune funzioni "minime" del S.O. fanno parte del kernel, altri moduli girano in modalità utente; questo rende il S.O. più robusto:

- **le operazioni "delicate" possono essere effettuate solo da una piccola parte del codice, sperabilmente più facile da rendere priva di bug**
- **gli altri moduli (es. driver di dispositivi forniti assieme agli stessi) possono fare meno danni**

Intermediario tra programmi e risorse

***Risorse del sistema*: CPU, Memoria, Dischi, ...**

- **permettere a diversi programmi di condividere la CPU, utilizzandola un po' ciascuno (time multiplexing)**
- **permettere a diversi programmi di condividere la memoria suddividendo sezioni secondo assegnate a programmi diversi (space multiplexing)**
- **permettere a diversi programmi di condividere lo spazio su disco suddividendo lo spazio secondo assegnato a programmi diversi (space multiplexing)**
- **permettere a diversi programmi di condividere dispositivi di I/O (stampante, ...) secondo un criterio temporale (time multiplexing)**

In sintesi:
permette l'esecuzione *protetta* e *simultanea* di più programmi

Evitare interferenze:

garantire che i programmi utente durante l'esecuzione:

- **non interferiscano con il sistema operativo**
- **non monopolizzino le risorse**
- **non interferiscano fra loro**

Il Sistema Operativo come macchina virtuale: le astrazioni introdotte dal S.O. e le funzioni offerte ai programmi e agli utenti per utilizzarle

Astrazioni introdotte dal SO

- **PROCESSI**
- **MEMORIA VIRTUALE**
- **CANALI DI COMUNICAZIONE TRA PROCESSI**
- **GESTIONE I/O**
- **FILE e DIRECTORY**
- **MECCANISMI DI PROTEZIONE**

Le *chiamate di sistema* sono funzioni che rendono possibile agire sulle entità astratte realizzate dal sistema operativo

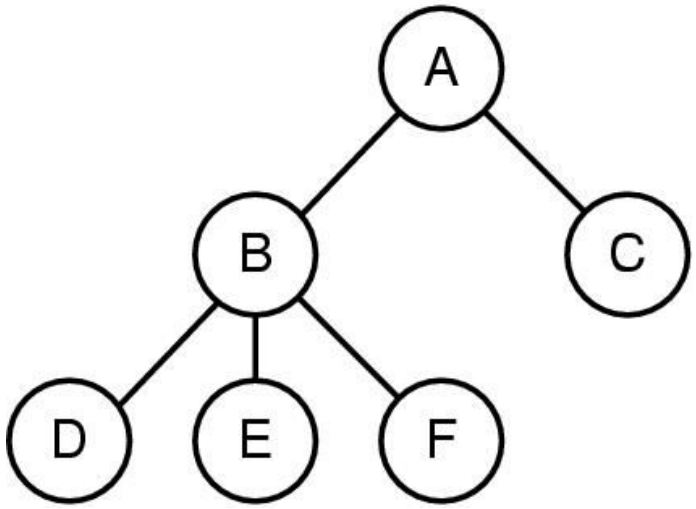
Sono le istruzioni aggiuntive della macchina virtuale di livello 3

Vedremo un esempio d'uso di system call: la *shell* (guscio) cioè l'interprete di comandi di UNIX

Astrazioni introdotte dal SO: i processi

- **PROCESSI**
- **MEMORIA VIRTUALE**
- **CANALI DI COMUNICAZIONE TRA PROCESSI**
- **GESTIONE I/O**
- **FILE e DIRECTORY**
- **MECCANISMI DI PROTEZIONE**

Un PROCESSO è un'attività di elaborazione guidata da un programma.
È caratterizzato da un suo spazio (privato) di indirizzi di memoria, dal valore dei registri (che insieme alla memoria determinano lo stato dell'esecuzione), più altre informazioni memorizzate in una *tabella dei processi* mantenuta dal sistema operativo



Un processo può creare altri processi, es. in figura (albero di processi)

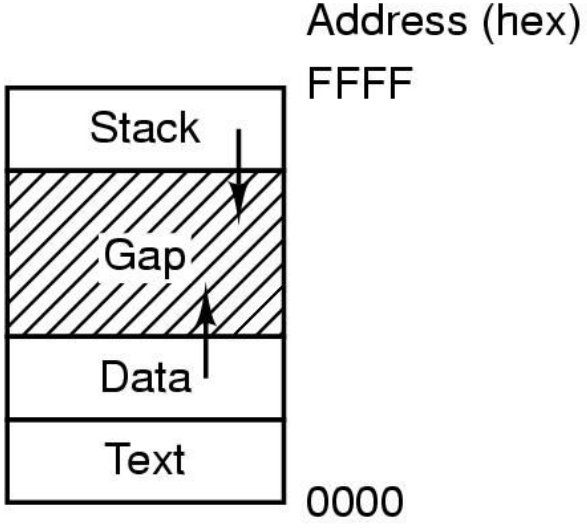
- **A ha due processi figli, B e C**
- **B ha tre processi figli, D, E, e F**

Astrazioni introdotte dal SO: memoria

- PROCESSI
- MEMORIA VIRTUALE
- CANALI DI COMUNICAZIONE TRA PROCESSI
- GESTIONE I/O
- FILE e DIRECTORY
- MECCANISMI DI PROTEZIONE

I processi hanno uno spazio di indirizzi che potrebbe essere più grande della RAM disponibile; e nei sistemi multiprogrammati (più processi in RAM) lo spazio necessario potrebbe essere maggiore di quello disponibile in RAM

Il sistema operativo gestisce (in modo *trasparente,* cioè, per quanto possibile, invisibile) la memoria in senso lato (RAM + disco) in modo da creare l'illusione di una memoria (virtuale) dedicata al processo.

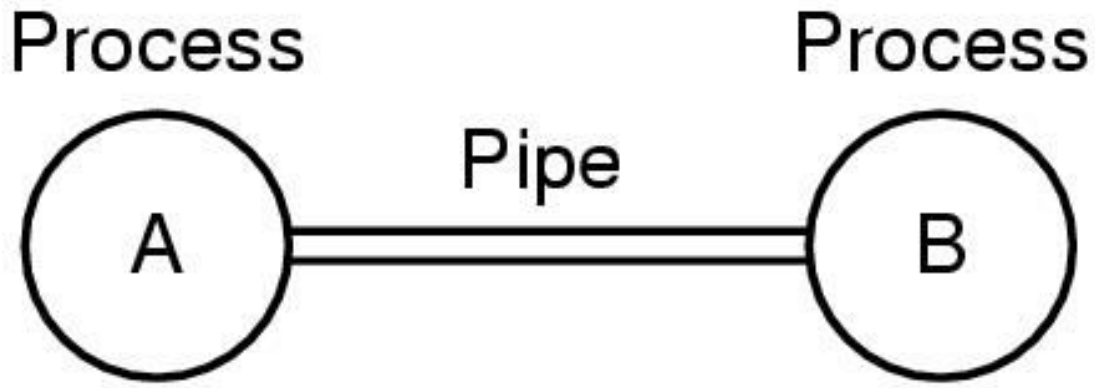


Esistono delle system call per l'allocazione dinamica della memoria (segmento dati): brk (la funzione di libreria è la malloc)

Inoltre le system call per la creazione (e terminazione) dei processi gestiscono anche l'allocazione (e il rilascio) dello spazio per il nuovo processo (o del processo terminato)

Astrazioni introdotte dal SO: comunicazione

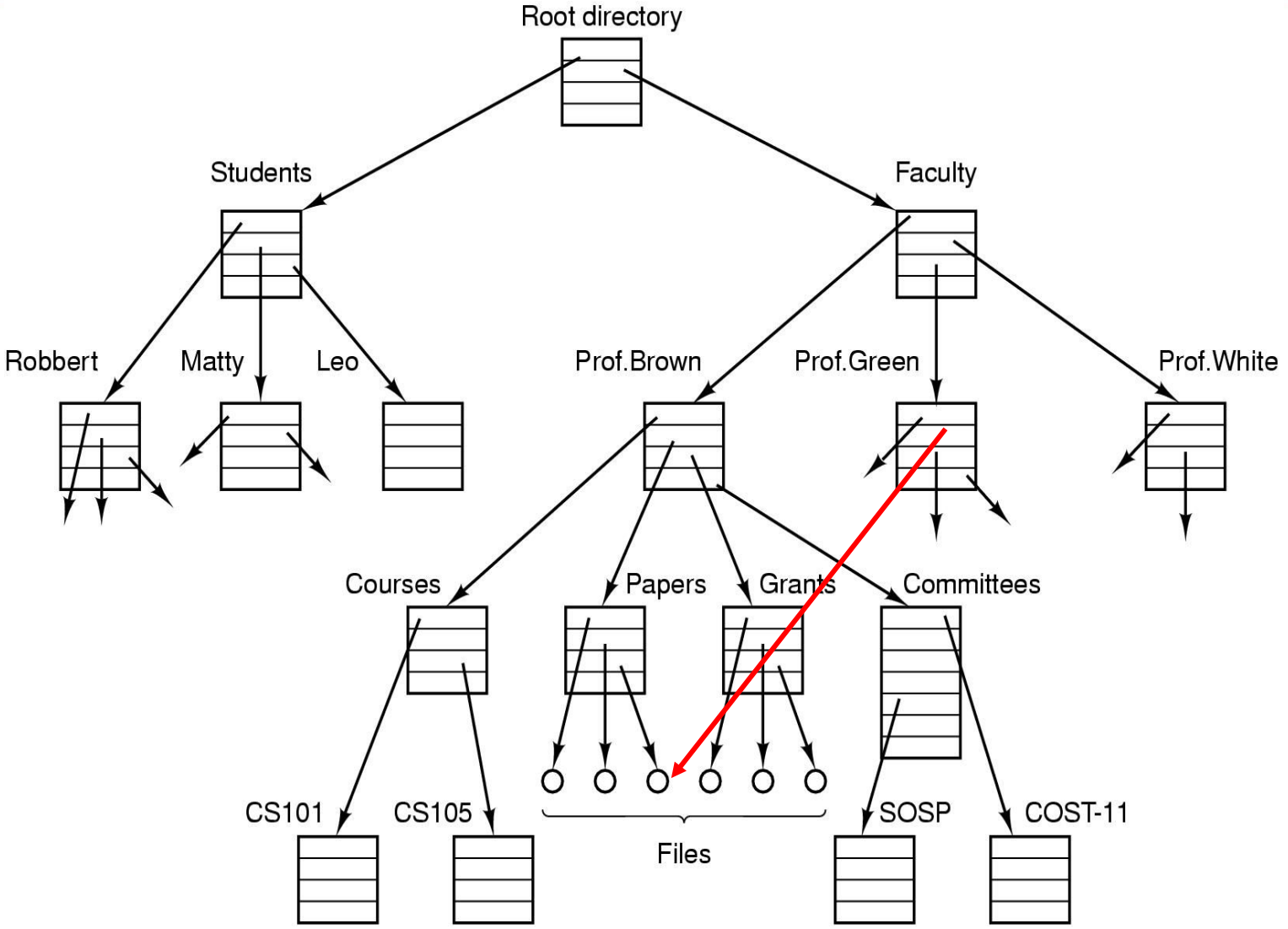
- **PROCESSI**
- **MEMORIA VIRTUALE**
- **CANALI DI COMUNICAZIONE TRA PROCESSI**
- **GESTIONE I/O**
- **FILE e DIRECTORY**
- **MECCANISMI DI PROTEZIONE**



Due processi possono comunicare, ad esempio per mezzo di una *pipe* (conduttura). La pipe è vista per certi versi come un file, ma con due estremità, una solo scrivibile, una solo leggibile. Si può leggere e scrivere con le funzioni di read/write file (la read in questo caso "consuma" i byte letti). In più il S.O. gestisce la sincronizzazione fra lettori e scrittori (lettore attende se pipe vuota, scrittore attende se pipe piena)

Astrazioni introdotte dal SO: file e directory

- PROCESSI
- MEMORIA VIRTUALE
- CANALI DI COMUNICAZIONE TRA PROCESSI
- GESTIONE I/O
- FILE e DIRECTORY
- MECCANISMI DI PROTEZIONE



***File*: collezione di dati correlati fra loro**
***Directory*: usate per organizzare i file**
ogni file/directory è identificato da un *pathname* (percorso).

Astrazioni introdotte dal SO: protezione

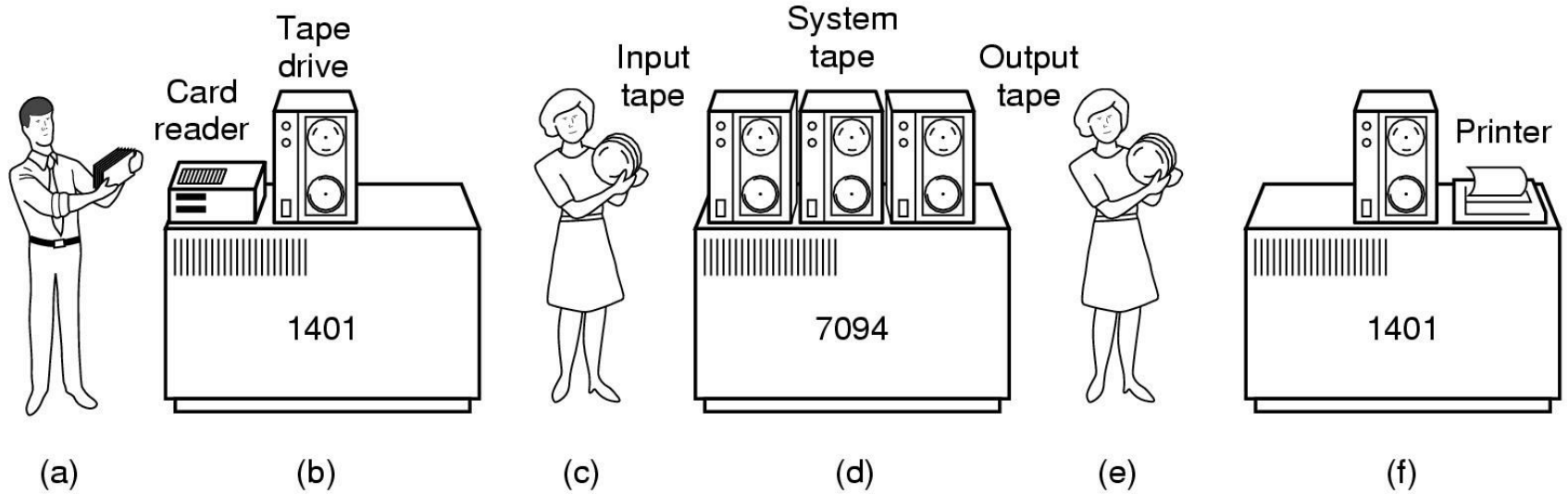
- PROCESSI
- MEMORIA VIRTUALE
- CANALI DI COMUNICAZIONE TRA PROCESSI
- GESTIONE I/O
- FILE e DIRECTORY
- MECCANISMI DI PROTEZIONE

***Meccanismi di protezione*: garantiscono un accesso controllato ai dati (memorizzati sul file system o in aree di memoria condivisa) da parte di utenti diversi.**

Per garantire la protezione di dati e programmi in esecuzione deve essere introdotto il concetto di "utente" o "gruppo di utenti", fornire *meccanismi di autenticazione* (per poter verificare che un utente sia effettivamente chi dice di essere) e *meccanismi per concedere ai diversi utenti permessi di tipo diverso* (scrivere/leggere files o directories, uccidere processi, leggere, modificare o cancellare aree di memoria condivisa o mailbox, ...)

La multiprogrammazione e i sistemi time sharing

Input da schede perforate o nastro / Output su stampante o nastro

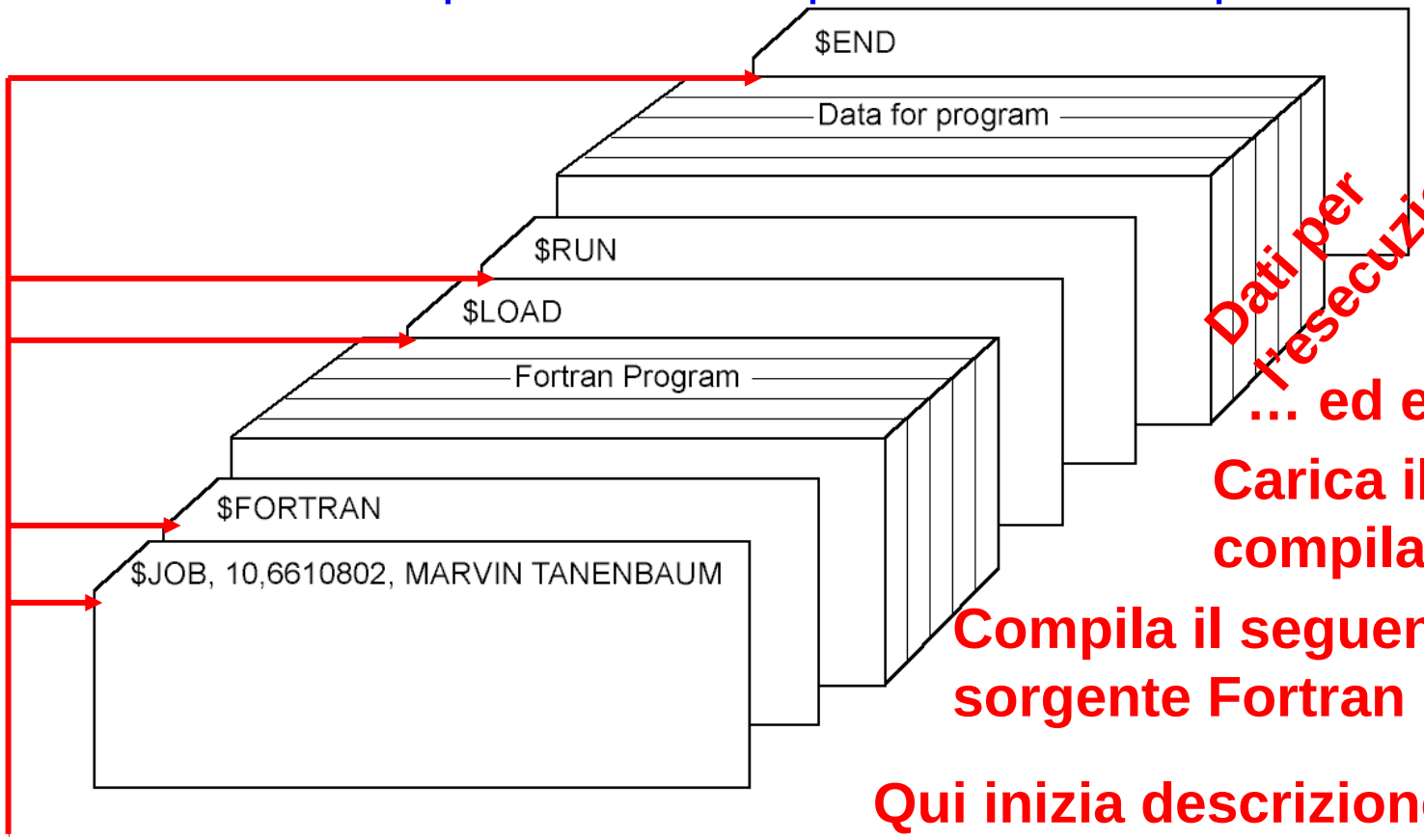


In un esempio di sistema batch ("a lotti" di job affini), l'operatore:

- **porta le schede al 1401 che trasferisce il contenuto su nastro**
- **inserisce il nastro nel 7094 (che effettua la vera computazione)**
- **inserisce il nastro su un altro 1401 che stampa i risultati**

Prime idee: disaccoppiare I/O lento dalla computazione
Su un nastro ci possono essere più job (un "lotto") che possono essere gestiti da un *Monitor Residente*, un programma sempre installato che regola il traffico, ad esempio eseguendoli uno dopo l'altro…
ma si può fare di meglio

Input tramite schede perforate / Output tramite stampante



Fine descrizione job …

Dati per l'esecuzione

… ed eseguilo …

Carica il programma compilato …

Compila il seguente sorgente Fortran …

Qui inizia descrizione job

Job Control Language

Sistemi NON INTERATTIVI: un operatore si occupa di far eseguire i job, Dispositivi di I/O sequenziali (e molto lenti), mentre le schede vengono lette la costosa CPU è inutilizzata

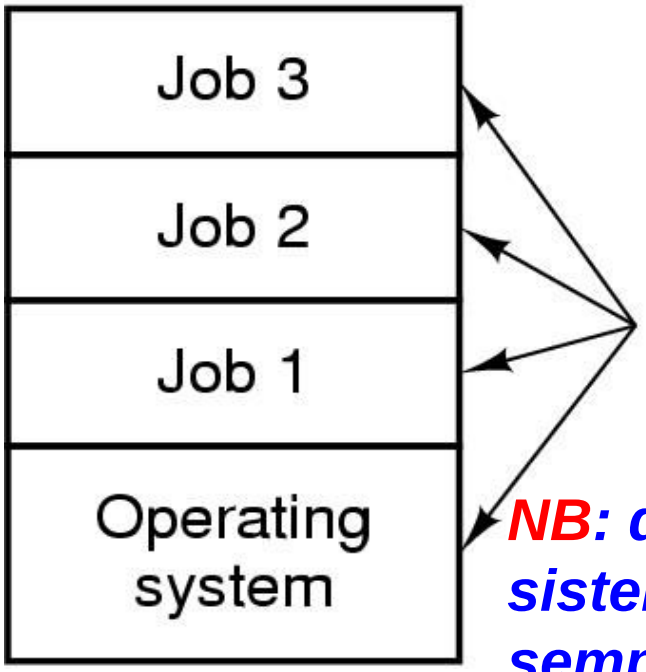
Unica componente del SO: routine per gestire l'I/O

La multiprogrammazione

Più job da eseguire possono essere a disposizione, in un lotto da nastro, o, con l'avvento dei dischi magnetici, su disco (lettura/scrittura più veloce)

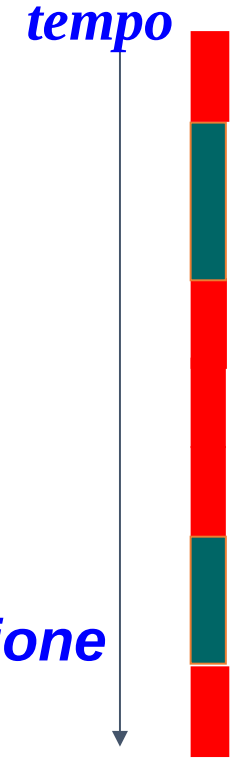
Obiettivo: utilizzare al massimo la CPU evitando i tempi morti dovuti alla lentezza dell'I/O rispetto alla CPU

Più job vengono caricati simultaneamente in memoria.



NB: questo è un sistema molto semplice di gestione della memoria

Osservazione: i job in esecuzione sono costituiti da una sequenza di CPU burst alternati ad I/O burst



CPU burst ***= sequenza di operazioni in cui un programma (se ha a disposizione tutto il sistema di elaborazione) usa in modo continuativo la CPU***

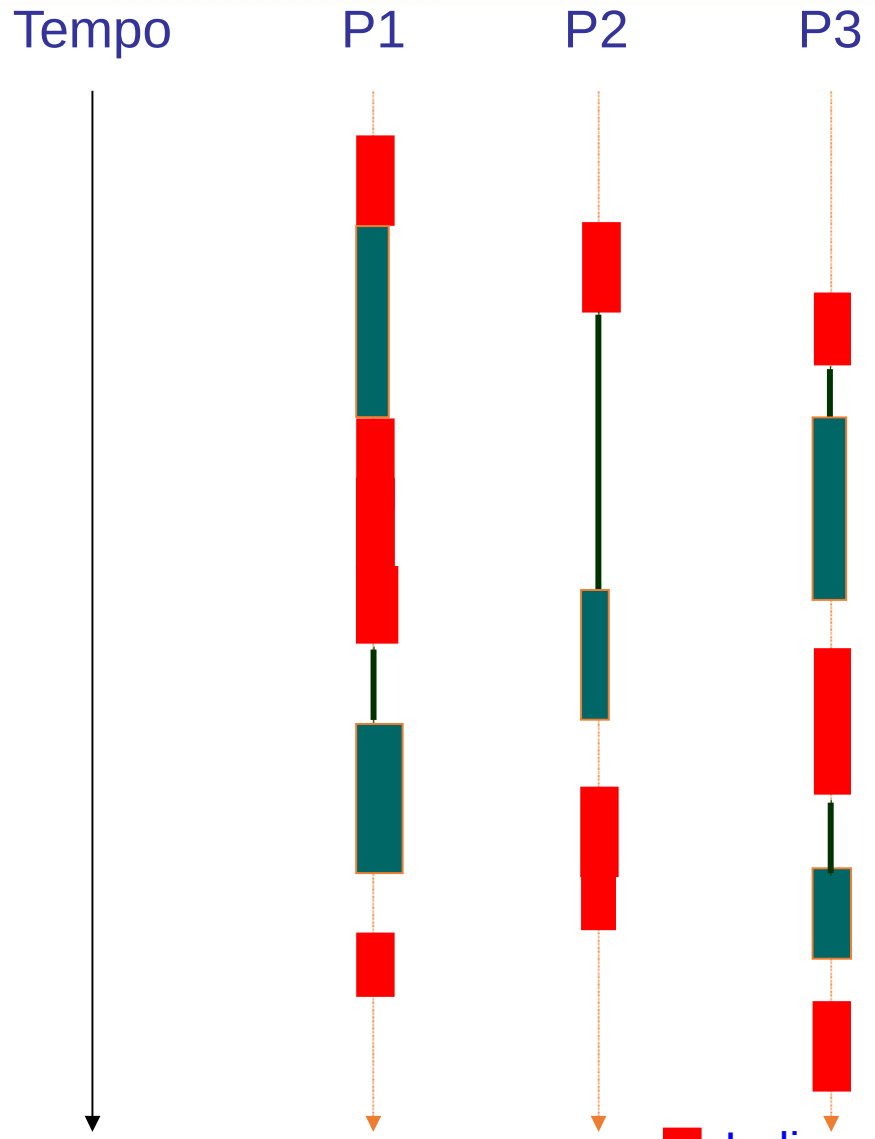
I/O burst

CPU burst

I/O burst

CPU burst

Esecuzione di un sistema multiprogrammato



- **Più job vengono caricati simultaneamente in memoria.**
- **La CPU viene assegnata in momenti diversi a diversi programmi in esecuzione: quando il programma a cui è assegnata la CPU chiede (al S.O.) l'esecuzione di una operazione di I/O, la CPU viene assegnata ad un altro programma: si ha sovrapposizione di I/O e computazione**
- **OSSERVAZIONE: Se il *livello di multiprogrammazione* è alto, la CPU risulterà poco inutilizzata.**

Context switch (Cambio di contesto)

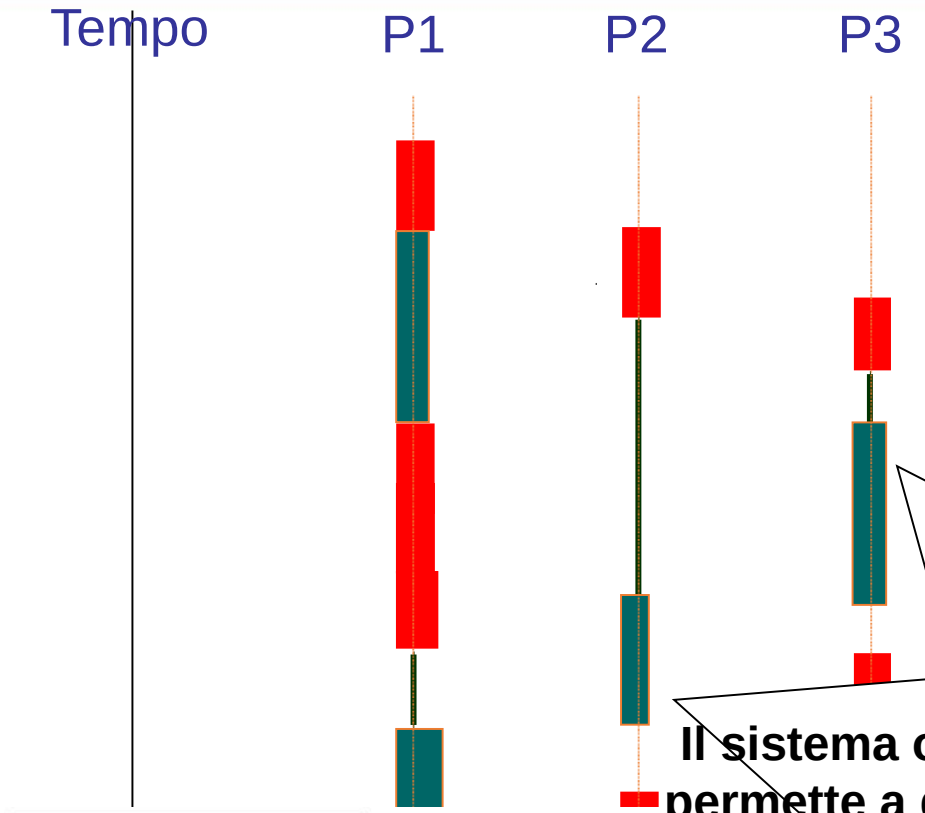
Indica chi è *running* in ogni istante

Indica per chi lavora il disco in ogni istante

Indica un processo in attesa di iniziare un I/O burst

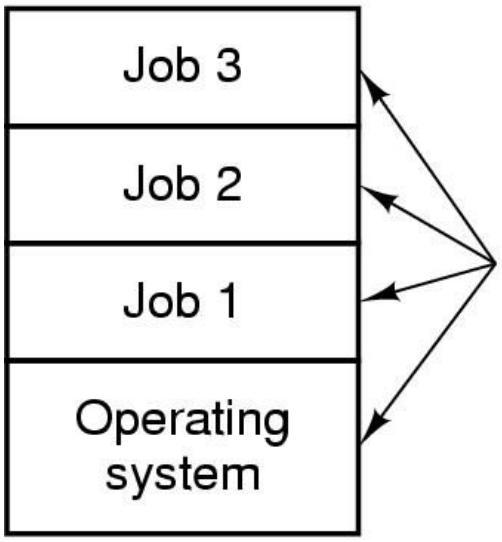
Indica un processo in attesa di iniziare un CPU burst

Esecuzione di un sistema multiprogrammato

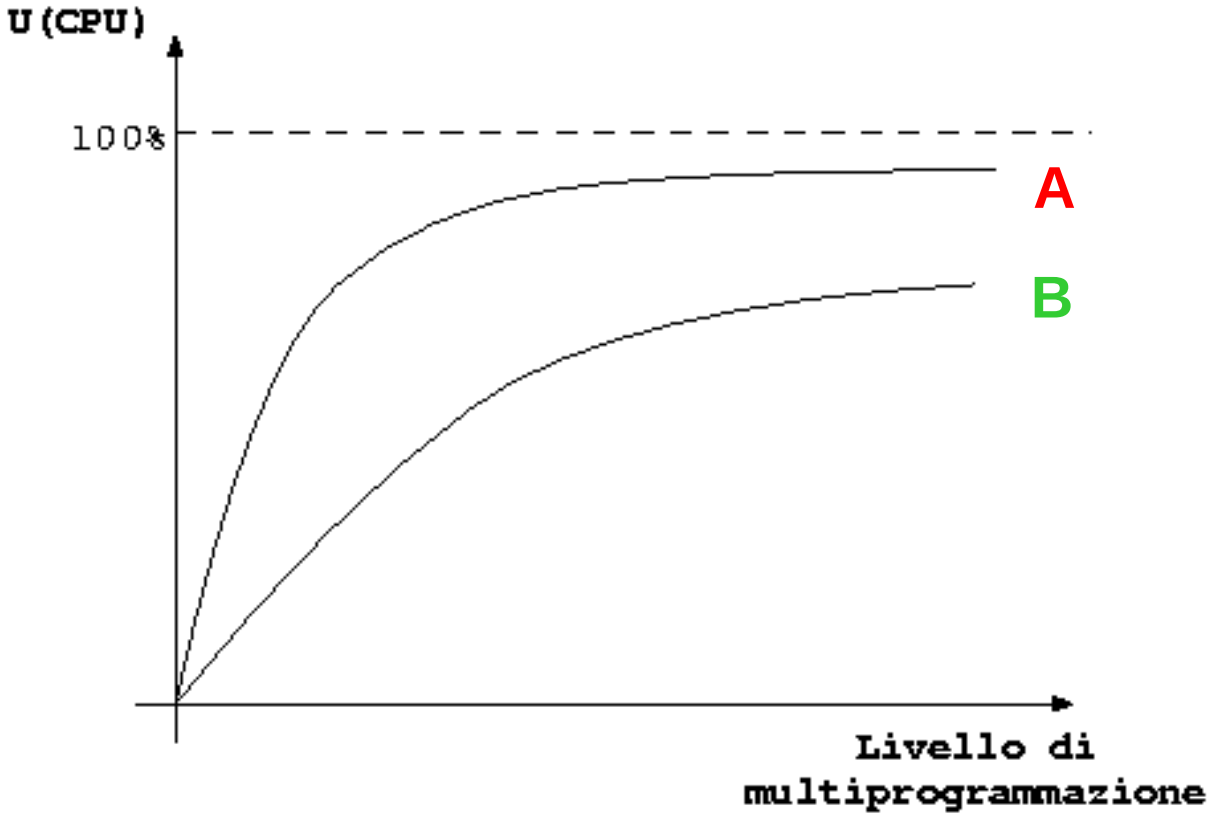


- **Più job vengono caricati simultaneamente in memoria.**
- **La CPU viene assegnata in tempi diversi ai vari programmi in esecuzione**

Il sistema operativo (nella veste di gestore delle risorse) permette a diversi programmi di condividere la CPU (time multiplexing) e la memoria disponibile (space multiplexing)



Livello multiprogrammazione e



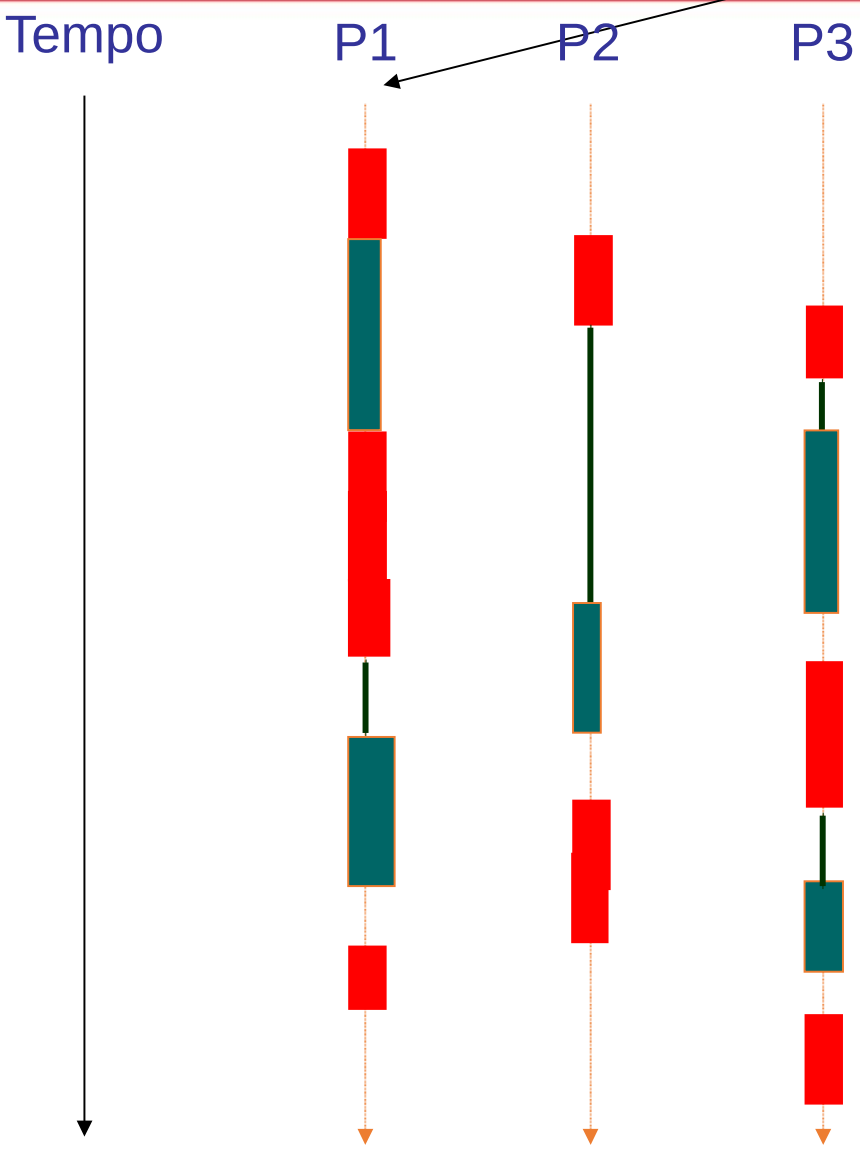
A: carico di tipo CPU-bound (i programmi richiedono poco frequentemente I/O, cioè hanno CPU burst piuttosto lunghi)
B: carico di tipo I/O-bound (i programmi richiedono molto frequentemente I/O, cioè hanno CPU burst brevi)

***Livello di multiprogrammazione* = numero di programmi simultaneamente caricati in memoria**

Il context switch

Il trasferimento della CPU da un programma ad un altro si chiama *context switch (cambio di contesto)*: ad ogni evento di questo tipo, il S.O. deve *salvare lo stato del programma* (i registri della CPU) in esecuzione e caricare lo stato salvato del programma prescelto tra quelli in attesa della CPU.

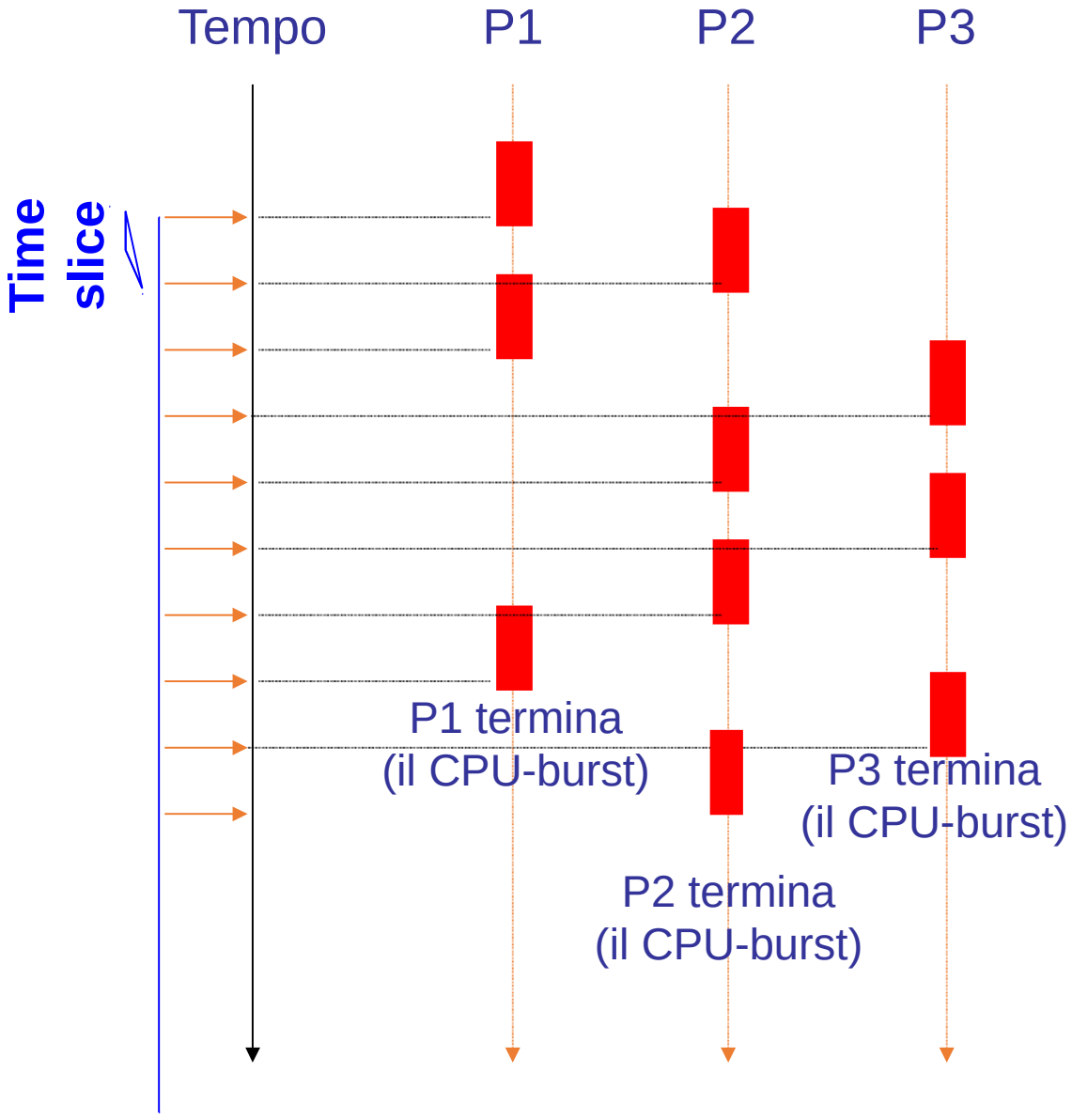
Esecuzione di un sistema multiprogrammato



Il *cambio di contesto* deve essere eseguito dal S.O.
Quindi bisogna passare ad eseguire codice del S.O.
Approfittiamo del fatto che *anche per eseguire I/O* c'è bisogno del S.O.: il programma effettua una chiamata di sistema, che si occupa di passare la richiesta al dispositivo e – se opportuno – effettua il cambiamento di contesto
Questo meccanismo è sufficiente per mantenere la CPU utilizzata

Multiprogrammazione vs time sharing

Dai sistemi multiprogrammati ai sistemi interattivi (time sharing)



Obiettivo: garantire tempi di "reazione" rapidi ai programmi *interattivi*, evitando per lo meno che uno monopolizzi la CPU

La CPU viene assegnata a turno ai vari programmi in esecuzione (un *time-slice* ciascuno). Un dispositivo hardware, il *timer,* invia un interrupt alla CPU per garantire che ciascun programma non occupi la CPU più a lungo di un *time slice* (se ce ne sono altri che devono girare).

Indica chi è *running* in ogni istante

Scelta della durata del time slice

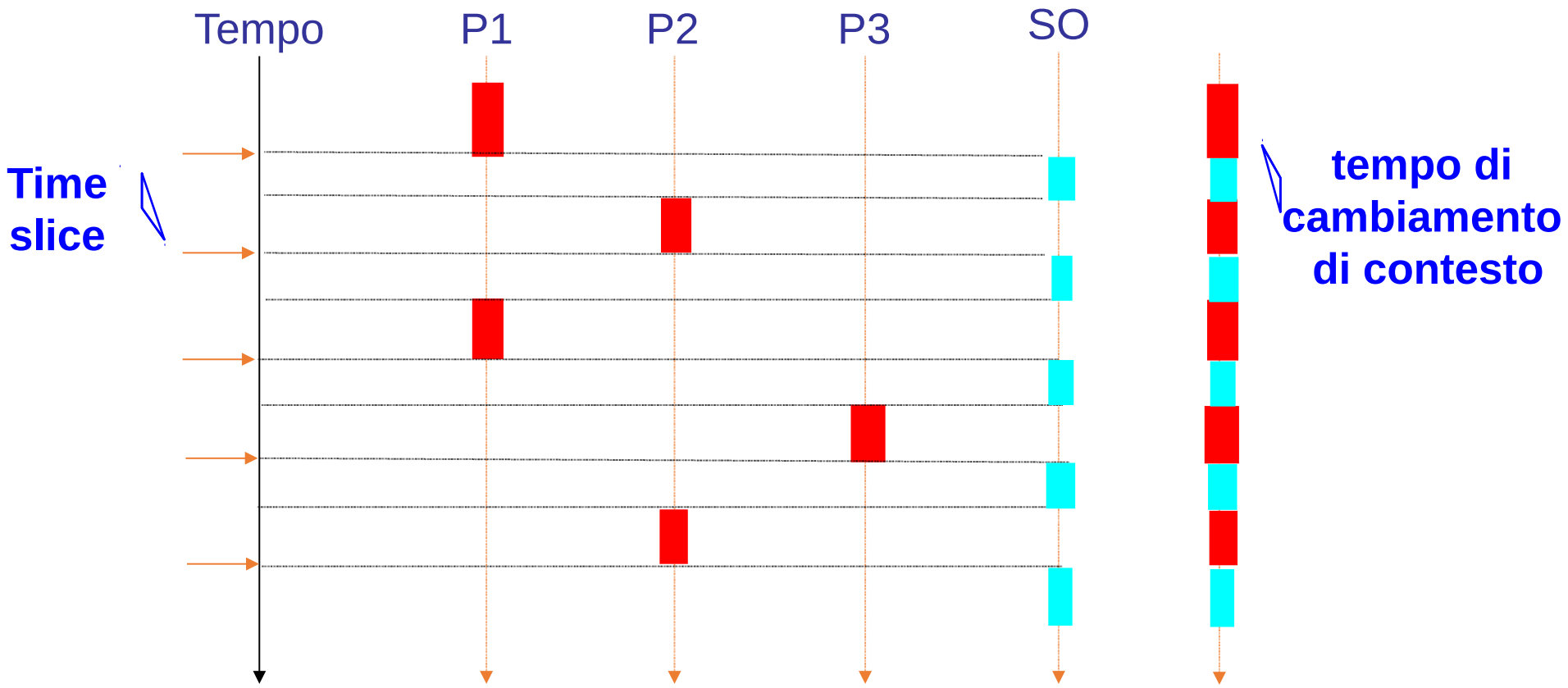
Per ottenere risposte rapide ai comandi degli utenti interattivi conviene impostare il *time slice* (o *quanto di tempo*) ad un valore "piccolo" (se k processi girano a turno, ciascuno girerà entro k quanti)

Tuttavia *time slice* troppo brevi causano un significativo *overhead :* costo di gestione aggiuntivo rispetto al costo in tempo e spazio per l'esecuzione dei programmi. In questo caso, è il tempo per il cambio di contesto, eventualmente preceduto dalla scelta del programma da far eseguire

Il cambiamento di contesto richiede un tempo piccolo ma non nullo, dipendente dall'architettura

Se è troppo frequente, il sistema di elaborazione passa una frazione di tempo troppo grossa a fare cambi di contesto

Scelta della durata del time slice

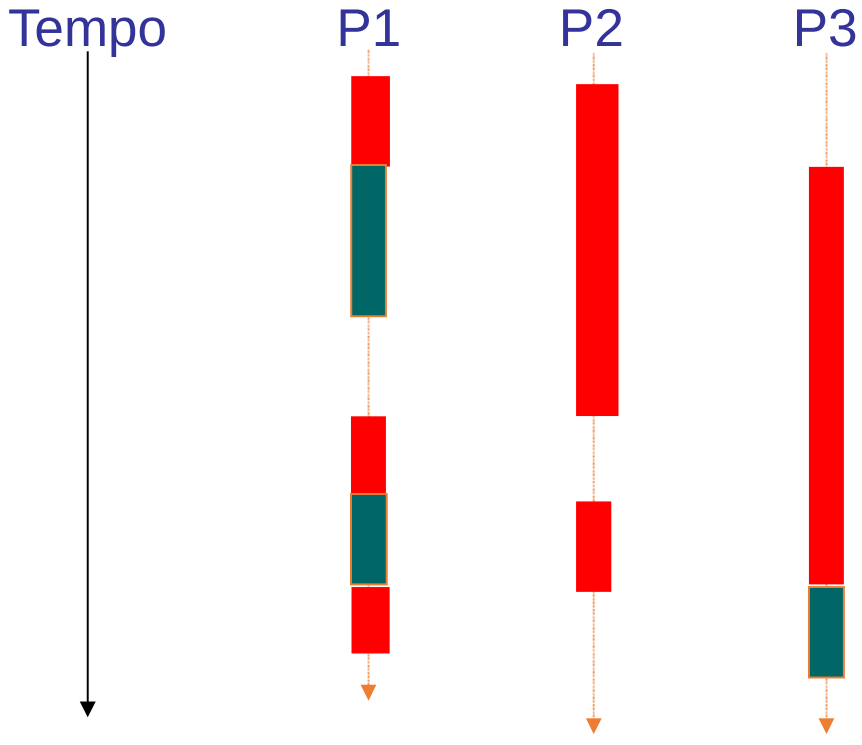


se quanto di tempo e tempo di cambiamento di contesto sono dello stesso ordine di grandezza, il sistema di elaborazione passa troppo tempo a fare gestione

Nelle architetture moderne

Oggi è comune avere a disposizione macchine che permettono, con una o più tecnologie (CPU multiple, CPU multicore, multithreading simultaneo), di avere vero parallelismo: fino a k programmi a cui è contemporaneamente assegnata una risorsa di elaborazione.

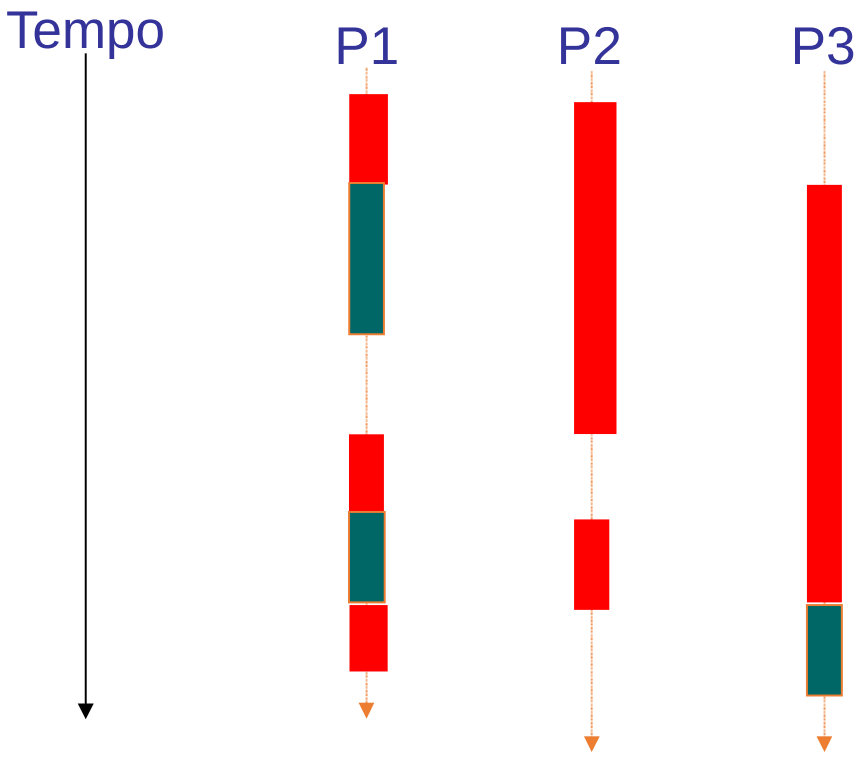
Le soluzioni precedenti continuano ad avere senso, la differenza è che possiamo avere fino a k programmi che stanno effettuando un CPU burst (nel disegno k=2)



Nelle architetture moderne

Rimane il fatto che se ho più di k programmi "pronti" (non devono attendere I/O o altri eventi se non la disponibilità di una CPU):

1. **Quando uno dei k programmi che stanno usando i k "posti disponibili" per usare la/le CPU deve attendere (I/O), uso il suo "posto" per un altro programma "pronto" (i criteri per scegliere *quale* possono cambiare rispetto al caso di CPU singola)**
2. **Non voglio che k programmi monopolizzino le CPU a danno degli altri**



La multiprogrammazione: vantaggi e problemi

***Vantaggi*: mentre un programma deve attendere la fine di una operazione di I/O, un altro può essere eseguito, quindi *la CPU (e ogni altra risorsa) viene sfruttata meglio*.**

***Problemi*: Bisogna far convivere diversi programmi, appartenenti anche ad utenti diversi, senza che interferiscano. Occorre gestire in modo appropriato le risorse (CPU e Memoria, in primo luogo) e assicurare una *esecuzione protetta*.**

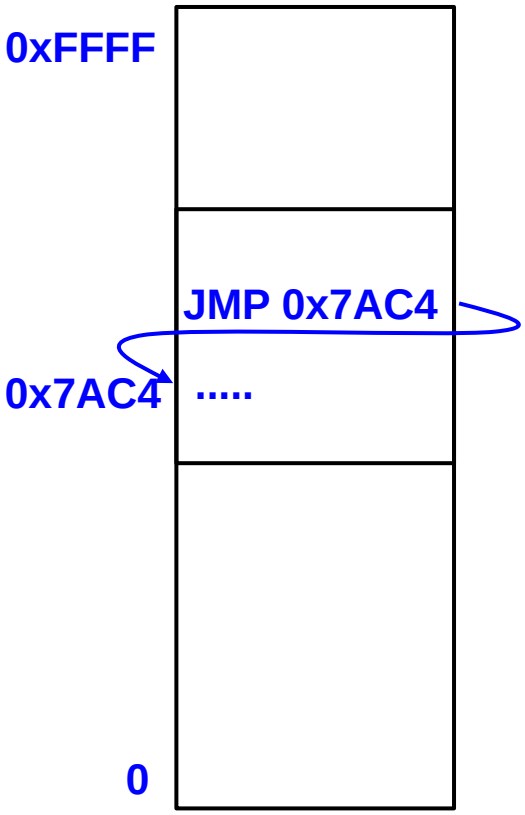
Per realizzare questo, c'è bisogno di aiuto da parte dell'hardware :

- **(almeno) due modalità di esecuzione della CPU (kernel e user)**
- **la distinzione tra istruzioni (di livello ISA) PRIVILEGIATE, che possono essere eseguite SOLO IN MODO KERNEL, e NORMALI che possono essere eseguite in qualunque modo**
- **supporto per la gestione della memoria, per fare in modo che un programma non possa accedere a codice e dati del S.O. e degli altri programmi**

Memoria, indirizzi virtuali e indirizzi fisici

Real programmers code in binary.

RAM

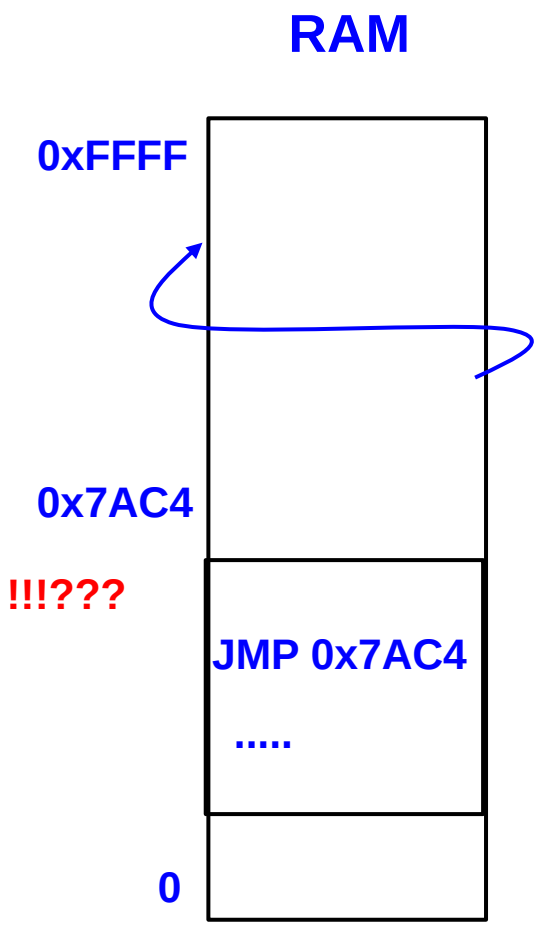


Come possono convivere più programmi in memoria?
Come si decide dove allocare uno specifico programma?
Come si fa a gestire flessibilmente l'allocazione senza dover ricalcolare, a cura del programma, gli indirizzi che compaiono nel codice ? (indirizzi di variabili da caricare/modificare o di istruzioni a cui saltare)

Idea: utilizzare *indirizzi relativi (o virtuali)* e poi fare *traduzione in indirizzi fisici* quando si sa dove è allocato il programma

- **a tempo di compilazione (troppo vincolante)**
- **a tempo di caricamento (non si può spostare durante l'esecuzione)**
- **a tempo di esecuzione (massima flessibilità)**

Memoria, indirizzi virtuali e indirizzi fisici

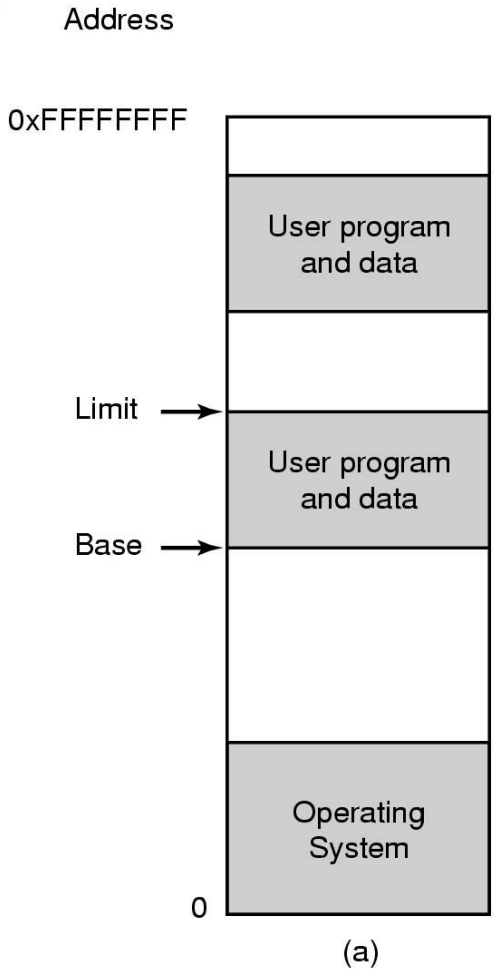


Come possono convivere più programmi in memoria?
Come si decide dove allocare uno specifico programma?
Come si fa a gestire flessibilmente l'allocazione senza dover ricalcolare, a cura del programma, gli indirizzi che compaiono nel codice ? (indirizzi di variabili da caricare/modificare o di istruzioni a cui saltare)

Idea: utilizzare *indirizzi relativi (o virtuali)* e poi fare *traduzione in indirizzi fisici* quando si sa dove è allocato il programma

- **a tempo di compilazione (troppo vincolante)**
- **a tempo di caricamento (non si può spostare durante l'esecuzione)**
- **a tempo di esecuzione (massima flessibilità)**

Gestione memoria con protezione:



I registri Limit e Base possono essere modificati solo in modo kernel (quindi solo dal S.O.). La traduzione indirizzo logico-indirizzo fisico e il controllo di non superamento dei limiti viene fatta in hardware, dalla MMU (Memory Management Unit).

I/O guidato da interrupt

Le trap: interrupt software

GLI INTERVENTI DEL SISTEMA OPERATIVO SONO INNESCATI DA INTERRUPT (hw) E TRAP (sw) questo meccanismo (unitamente alle 2 modalita' di esecuzione della CPU) permette al S.O. di svolgere il suo ruolo di CONTROLLORE

Gli interrupt

La CPU impartisce comandi ai dispositivi scrivendo determinate informazioni nei registri dei controller.

L'operazione di I/O può essere controllata da programma, oppure essere basata su interrupt. Il primo approccio impegna la CPU durante lo svolgimento dell'operazione da parte del dispositivo. Il secondo invece lascia libera la CPU, e la avverte inviando sul bus un *interrupt* quando è effettivamente necessario il suo intervento (alla fine dell'operazione).

Le istruzioni per la comunicazione con i controller dei dispositivi di I/O sono privilegiate (quindi eseguibili esclusivamente in modo *kernel*): in questo modo si impone che solo il S.O. possa comunicare con i dispositivi di I/O a basso livello, garantendo un uso protetto degli stessi da parte dei processi utente.

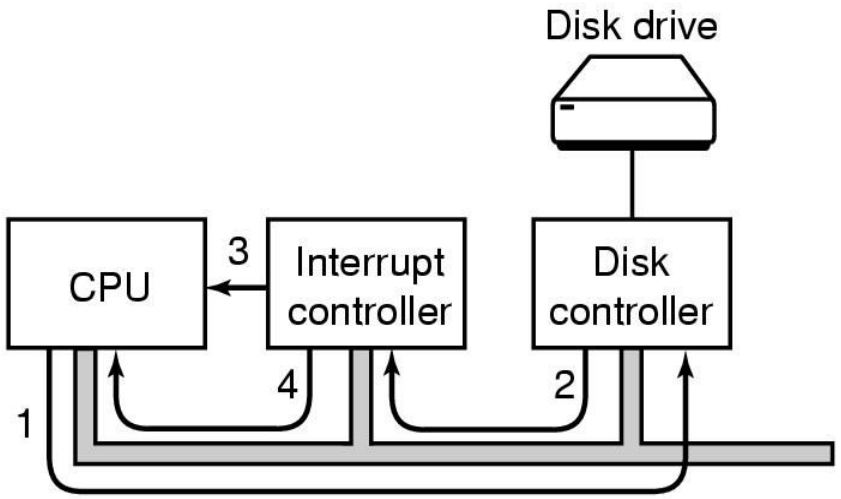
Gestione degli interrupt

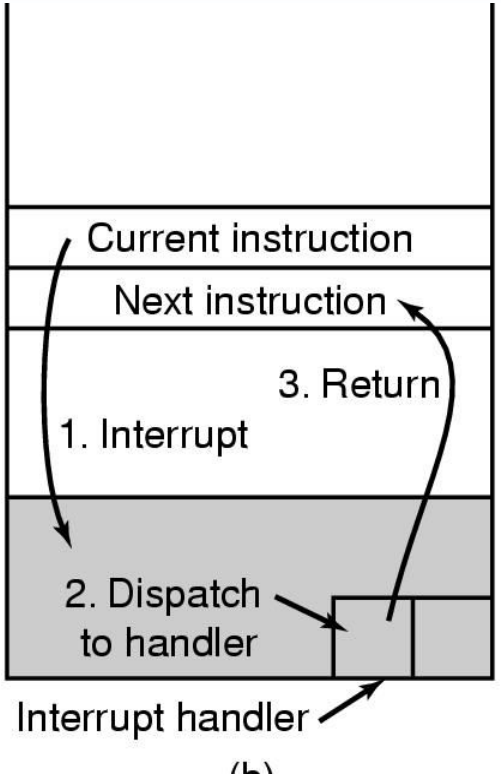
Cosa succede quando la CPU riceve un interrupt:
Nota: quando ha ricevuto l'interrupt, la CPU stava eseguendo un programma, probabilmente NON quello che aveva richiesto l'I/O.

- **l'istruzione macchina (livello ISA) in corso viene completata (le interruzioni, se non disabilitate, vengono «sentite» all'inizio di ogni ciclo)**
- **si salva in memoria lo stato della computazione interrotta (come nel caso di context switch)**
- **si salta ad una posizione in memoria dove è contenuto il gestore (handler) di quel tipo di interruzioni (es. driver del disco)**
- **dopo aver gestito l'interrupt, (quasi sempre) viene ripristinato lo stato del programma interrotto e la sua esecuzione prosegue come se nulla fosse: l'interrupt è trasparente (invisibile) al programma**

Sulle architetture moderne è più complicato, anche per un programma sequenziale l'esecuzione delle istruzioni è parzialmente sovrapposta nel tempo

Sequenza di gestione dell'interrupt





1. il driver dice al controller cosa fare
2. il controller, al termine dell'operazione, invia un interrupt
3. l'interrupt controller (se libero) inoltra il segnale alla CPU e
4. indica alla CPU l'ID di chi ha generato l'interrupt, poi…

1. … la CPU termina l'istruzione corrente e ...
2. ... passa il controllo all'interrupt handler;
3. terminata la gestione dell'interrupt riprende l'esecuzione.

Alcuni esempi di interrupt handler

L'interrupt handler per un *interrupt da disco*, se esistono richieste di I/O in attesa del disco, ne inoltra una al controller. Inoltre il programma che era in attesa dell'I/O appena terminato viene riportato nell'insieme dei programmi "pronti" a cui può essere assegnata la CPU.

L'interrupt handler per un *timer interrupt* si occupa dell'assegnazione della CPU ad un altro programma, fra quelli pronti per l'esecuzione.

Rilevamento e abilitazione/disabilitazione degli interrupt

Per poter gestire gli interrupt la CPU deve poter *rilevare la presenza di un interrupt* (effettuando un test su un bit di un registro dedicato a questo scopo), leggere il *device_id* inviato dal controller che ha generato l'INT, disabilitare gli interrupt (ad es. per evitare interruzioni annidate) impostando un bit in un apposito registro, passare al modo kernel, e dopo aver gestito l'interrupt riabilitarle gli interrupt e tornare ad eseguire il processo interrotto.

Comportamento della CPU senza gestione degli interrupt:

```
while (not halt ) do
  { fetch istruzione;
    decodifica istruzione;
    esegui istruzione;
  }
```

Comportamento della CPU con gestione degli interrupt:

```
while (not halt ) do
  { if (interrupt & not int_disable)
       {salva stato; invia INTA; %ack
         leggi device_id;
         int_disable = true;modo=kernel;
         PC = int_vector[device_id];}
    fetch istruzione;
    decodifica istruzione;
    esegui istruzione;  }
```

Rilevamento e abilitazione/disabilitazione degli interrupt

Il set di istruzioni della ISA potrà comprendere una istruzione di "Ritorno da Interrupt" (RETINT), una di "Disabilita interrupt" (DISINT) e una di "Abilita interrupt" (ENABINT).

L'esecuzione della istruzione RETINT consiste nel riabilitare gli interrupt (int_disable = false) e ripristinare lo stato che era stato salvato prima di chiamare l'interrupt handler (riportando tra l'altro PC alla prossima istruzione da eseguire del programma interrotto).

L'esecuzione di DISINT e ENABINT consiste semplicemente nell'impostare il bit int_disable rispettivamente a true e false.

Le istruzioni RETINT, DISINT, ENABINT sono PRIVILEGIATE (eseguibili solo in modo kernel). In questo modo si evita ad esempio che un processo utente monopolizzi la CPU chiamando DISINT

Per questo c'è "modo=kernel" appena viene rilevato un interrupt. La RETINT ripristinando lo stato ripristina anche il registro PSW riportando "modo == utente"

Istruzioni privilegiate

La fase di esecuzione di una qualsiasi istruzione privilegiata inizierà con un test sul bit di modo: se questo è impostato a "kernel" l'esecuzione prosegue, altrimenti viene generata una eccezione (trap), il programma che ha tentato di eseguire l'istruzione privilegiata viene interrotto passando alla routine di gestione della trap

Essendo privilegiate:

- **le istruzioni di I/O di basso livello**
- **le istruzioni per impostare il valore dei registri utilizzati per il calcolo degli indirizzi fisici a partire dagli indirizzi logici**
- **le istruzioni di disabilitazione degli interrupt**
- **le istruzioni la cui esecuzione comporta il passaggio da modo utente a modo kernel**

si può garantire la protezione necessaria dal comportamento scorretto di programmi utente.

Il passaggio da modo utente a modo kernel avviene esclusivamente in occasione di un interrupt hardware o software (trap)

Le trap: interrupt software

Le trap possono essere viste come *interrupt software*, cioè una interruzione provocata dall'esecuzione di una determinata istruzione del programma in esecuzione:

- **istruzioni con errore (opcode, parametri, overflow, privilegi,...)**
- **istruzioni esplicite di "trap" al sistema operativo.**

Le trap vengono trattate in modo del tutto analogo agli interrupt: ciascun tipo di trap ha un identificatore che viene usato per cercare nell'interrupt vector l'indirizzo della prima istruzione del corrispondente *trap handler*.

Le system call sono realizzate tramite trap. Per effetto di un interrupt/trap la CPU passa da user mode a kernel mode: in questo modo il gestore dell'interrupt/trap (che è un componente del software di sistema operativo) esegue in modalità *kernel*. Alla istruzione di ritorno da interrupt/trap, viene ripristinata la vecchia status word (PSW) del programma interrotto, e così si ritorna a modalità *user*.

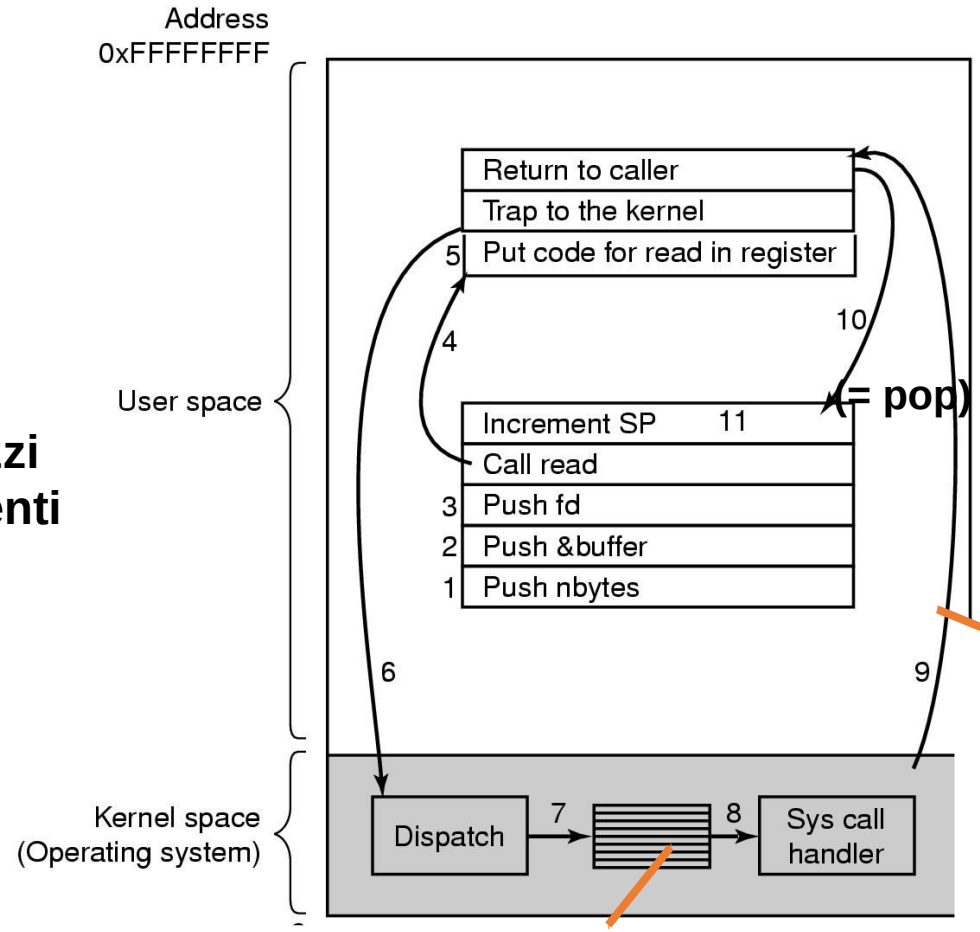
Gli Interrupt sono ASINCRONI rispetto al programma in esecuzione mentre le Trap sono SINCRONE: causate dal programma in esecuzione.

Passi nell'invocazione di una system call

Si accede ai servizi del sistema operativo tramite System Call

Esempio: i passi necessari per eseguire

read (fd, &buffer, nbytes)



Indirizzi crescenti

(codice della funzione read)

(codice che chiama read; usa stack qui non disegnata)

L'esecuzione della chiamata potrebbe sospendere il processo (es. se come qui è stata chiesta un'operazione di I/O), nel qual caso non si ritorna: si dà la CPU a un altro processo

Tabella di puntatori alle routine di gestione delle chiamate di sistema