# Riassunto argomenti sistemi operativi

Curzio e Tullio

November 8, 2022

## Contents

# 1 Capitolo: Introduzione

## 1.1 Che cos'è un sistema operativo

### 1.1.1 Definizone

Il sistema operativo viene prevalentemente eseguiuto con il processore in modalità kernel, mentre le applicazioni vengono eseguite in modalità utente.
Il sistema operativo esegue principalmente 2 funzioni:

- Fornisce ai programmatori di applicazione e ai programmi applicativi un insieme di risorse astratte.
- Gestisce le risorse hardware.

### 1.1.2 Obiettivi dei sistemi operativi

- Portabilità
- Efficenza
- Robustezza
- Scalabilità
- Estensibilità
- Sicurezza
- Protezione
- Interattività
- Usabilità

Alcuni di questi vanno in contrasto tra di loro;
come per esempio efficenza e portabilità, in quanto se si avessero entrambe si otterrebbe una macchina piccola e con alte prestazioni.
Oppure portabilità e scalabilità. La scalabilità denota in genere la capacità di un sistema di aumentare o diminuire di scala in funzione delle necessità e disponibilità

## 1.2 Storia dei sistemi operativi

### 1.2.1 Prima generazione (1945-55): valvole termoioniche

- Primo computer digitale funzionante → Iowa state univerity, funzionante grazie a 300 valvole termo.
- Successivi computer → Mark 1, Colossus, Eniac
- I computer erano programmabili in linguaggio macchina o attraverso il cablaggio manuale dei cavi fino all'introduzione delle schede perforate

### 1.2.2 Seconda generazione (1955-65): transistor e sistemi batch

- Avvento dei transistor → suddivisione dei ruoli di progettisti, costruttori, manutentore, operatori, programmatori
- Macchine giganti chiamate mainframe
- Sequenza esecuzione → programma su scheda perforata, operatore che carica programma in sala input, ed aspettava output in sala output
- Sistema inefficiente → creazione sistema Batch
- Sequenza esecuzione sistema batch → riversare schede perforate in nastro magnetico, nastro montato su computer principale, output scritto su un secondo nastro, portato ad un computer più piccolo per farlo stampare(stampa offline grazie a computer IBM 1401)
- Struttura input: $JOB → $FORTRAN → $LOAD → $RUN → $END;

### 1.2.3 Terza generazione (1965-80): IC - Multiprogrammazione

- 2 categorie di PC disponibili all'epoca:
  - word oriented: usati per calcoli scientifici (IBM 7094)
  - character oriented: usati per ordinamenti su nastro (IBM 1401)
- IBM introduce System/360 (obiettivo : compatibilità tra le macchine)
- IBM 360 prima linea ad adottare IC (integrated circuits)
- Intenzione di IBM è di far sì che il software OS/360 funzionasse in tutte le macchine → risultato: sistema operativo enorme, da milioni di righe, pieno di errori (linguaggio assembly)
- Tuttavia questo sistema operativo introduce concetti chiave della programmazione come:
  - **multiprogrammazione**: partizionamento della memoria per eseguire più processi e rendere la CPU più attiva;

- **spooling**: capacità di leggere i lavori su disco appena portati in sala macchine, rendendo il trasporto nastro ed i computer IBM 1401 superflui;

- introduzione **time-sharing** (variante della multiprogrammazione): potenza della CPU assegnata ad utenti attivi al momento, ogni utente dispone di un terminale attivo. Esempio: In un sistema time-sharing se ci sono 20 utenti collegati e 17 di loro non stanno facendo un cazzo, allora allora la CPU può essere assegnata ai 3 lavori che la richiedono.

- Primo sistema di time sharing → CTSS (compatible time sharing system)

- Questo porta alla creazione del MULTICS (multiplexed information and computing system), il cui obiettivo era quello di fornire potenza di calcolo a tutta boston)

- MULTICS successo solo parziale; utilizzava il compilatore PL/I, atteso e sviluppato per anni che funzionava a malapena;

- AVVENTO E CRESCITA DEI MINICOMPUTER
  Un ingegnere che aveva lavorato al multics trova un minicomputer e scrive un sistema operativo che sarebbe poi diventato UNIX.
  Per far si che venissero scritti programmi che girassero su tutti i sistemi UNIX venne istituito uno standard creato da IEEE, lo standard POSIX, che fornisce un'interfaccia minima per le chiamate di sistema che i sistemi UNIX dovrebbero supportare.
  Viene poi creato MINIX, piccolo clone di UNIX che aveva scopi didattici.

- Il desiderio di una versione libera di MINIX portò alla scrittura di LINUX (Linus Torvalds il Frocio).

### 1.2.4 Quarta Generazione (1980-oggi): i personal computer

- Sviluppo degli LSI (large scale integration)

- Arrivo dei microcomputer (vantaggiosi in termini di prezzo rispetto i minicomputer)

- Creazione del CM/P (control program for microcomputer), scritto su floppy disk, collegato alla prima CPU 8-bit sviluppata dalla Intel (CPU chiamata 8080);

- GUI(graphic user interface) ebbero un grande successo perché user-friendly

### 1.2.5 Quinta Generazione (1990 - oggi): i computer mobili

- Primo telefono mobile (1946) peso: 40 kili (lo stesso di tua madre SSSSIUUUUUUUUM)

- Il primo vero telefono portatile (anni 70): peso 1kg, soprannome “il mattone”
- Primo vero smartphone → nokia con il N9000: telefono con PDA (personal digital assistant)
- Sistemi operativi dominanti: Android, iOS;
- Altri sistemi operativi → Blackberry OS, Symbian OS (sistema operativo eseguito negli anni 2000- 2010);

## 1.3 Analisi dell'hardware

Il sistema operativo è intimamente legato all'hardware in cui gira.

### 1.3.1 Processore

La CPU è il cervello del PC, riceve istruzioni da un programma e le esegue in linguaggio macchina.

La CPU ha un insieme di istruzioni specifiche che può eseguire. Un processore ARM non può eseguire programmi per x86 e viceversa.

Tutte le CPU contengono al loro interno dei registri per memorizzare variabili importanti o risultati. I registri sono memorie ad alta velocità situate sui processori, tutti i dati prima di essere elaborati dalla CPU devo essere nei registri. Ci sono diversi tipi di registri:

- **Program counter** - Contiene indirizzo di memoria della successiva istruzione da eseguire.
- **Stack pointer** - Punta alla cima dello stack attualemnte in memoria. Lo stack ha un frame per ogni processo iniziato ma non ancora finito.
- **PSW (program status word)** - Contiene i bit con il codice di condizione, la priorità della CPU, modalità utente e kernel. Svolge un ruolo importante in chiamate di sistema e in I/O.

Per eseguire più di un'istruzione allo stesso tempo si addottano alcuni accorgimenti, per esempio una CPU può avere un dispositivo di recupero delle istruzioni, un dispositivo di decodifica e un dispositivo di esecuzione (tutti separati tra di loro). In questo modo mentre viene eseguita l'istruzione $n$ è possibile decodificare l'istruzione $n + 1$ e nel frattempo recuperare l'istruzione $n + 2$. Questa organizzazione è chiamata **pipeline**.

La CPU **multiscalare** contiene molteplici unità di esecuzioni per prendere più istruzioni alla volta, decodificarle e caricarle in un buffer fino alla loro esecuzione. Sta roba è più avanzata della pipeline.

Molte CPU hanno modalità kernel e modalità utente. In modalità kernel la CPU può eseguire ongi istruzione del suo insieme di istruzioni e utilizzare ogni caratteristica dell'hardware.
I programmi utente invece girano in modalità utente e quindi possono eseguire solo un sottoinsieme di istruzioni, con l'accesso solo a un sottoinsieme di caratteristiche.
Per ottenere un servizio dal sistema operativo l'utente deve fare una **System call**, che entra in modalità kernel e richiama il sistema operativo.

I **Chip multithread** hanno la possibilità di tenere lo stato di due diversi *thread*

e passare da uno all'altro in un nanosecondo. Il Multithreading ha delle implicazioni per il sistema operativo, perchè ogni thread appare al sistema come una CPU separata.

Al di là del multithreading abbiamo chip CPU con molti **core**, per utilizzare un chip multicore con tanti core è necessario un sistema multiprocessore.

La GPU è un processore che ha un fracco di core strapiccoli e vengono usati soprattutto per renderizzare cazzate o giocare a Minecraft in 4k.

### 1.3.2 Memoria & Dischi

In ogni computer il componente principale è la memoria, la quale si suddivide in strati gerarchici, variando capacità è velocità.

- Primo strato gerarchico: REGISTRI DELLA CPU hanno la velocità maggiore, il più alto costo in bit e la minor capacità. Questi registri sono veloci quanto la CPU, accedervi non comporta nessun tipo di ritardo. La capacità di memorizzazione al loro interno è di 32x32 bit in CPU a 32bit, mentre è di 64x64 bit in CPU a 64bit.

- Secondo strato gerarchico: MEMORIA CACHE La memoria cache è controllata dall'hardware, la stessa memoria principale è composta da LINEE DI CACHE tipicamente da 64 byte. Se un programma ha bisogno di leggere una parola dalla memoria, l'hardware controlla se l'informazione è presente all'interno della memoria cache. Se presente, si verifica un CACHE HIT e la richiesta viene esaudita dalla cache (i cache hit impiegano circa due cicli di clock). se non presente si verifica un CACHE MISS, e c'è bisogno di controllare la memoria principale per la soddisfazione della richiesta. l'utilizzo della cache è di fondamentale importanza, non solo per quel che riguarda le linee di cache della RAM. Infatti ovunque ci sia una grande risorsa suddivisibile in pezzi, si ricorre alla cache. le moderne CPU utilizzano due linee di cache, primo livello di cache L1, che si trova sempre all'interno della CPU, fornisce istruzioni codificate al motore di esecuzione della CPU. molti chip contengono una seconda linea di cache L1 per parole di dati usate frequentemente, la cui capacità è solitamente di 16kb. la seconda linea di cache L2, contiene molti megabyte di parole di memoria usate recentemente. La differenza tra le due sta nella velocità, dove l'accesso alla linea L1 avviene senza ritardi, mentre per la linea L2 l'accesso comporta un ritardo di uno o due cicli di clock.

- Terzo strato gerarchico: RAM/ Memoria principale (random access memory) in questo periodo storico hanno una capacità di diversi GB. Tutte le richieste insoddisfatte dalla memoria cache finiscono all'interno della memoria principale. Oltre alla memoria principale, i computer hanno spesso una piccola quantità di memoria non volatile (che non perde i dati a PC spento) che è la memoria ROM (read only memory), la quale viene

programmata dal costruttore e non può essere modificata in seguito. In alcuni pc il programma di avvio per far partire il computer risiede proprio nella ROM. Esistono anche le EEPROM (electrically erasable PROM) e le memorie flash, anch'esse non volatili ma cancellabili e riscrivibili. Un altro tipo di memoria è la CMOS, la quale viene utilizzata per memorizzare la data e l'ora correnti. Questa memoria e il circuito dell'orologio utilizzato per tenere aggiornata l'ora e la data vengono alimentati grazie ad una batteria che si scarica dopo parecchi anni, dato il bassissimo consumo di energia della memoria CMOS.

- L'ultimo strato gerarchico: Il DISCO MAGNETICO (hard-disk): la memoria su disco è di due ordini di grandezza più voluminosa ed economica della RAM, e tre ordini di grandezza più lenta della RAM. Un disco è formato da uno o più piatti metallici rotanti a diverse velocità (5400-7200-10200 giri/minuto). Informazione viene scritta su disco in una serie di cerchi concentrici e, a ogni data posizione del braccio, ogni testina legge una porzione di spazio chiamata traccia, tutte le tracce di una posizione specifica formano un cilindro. I dischi, essendo meccanici, sono piuttosto lenti. Infatti per muovere il braccio da un cilindro al cilindro successivo richiede circa 1 ms, muoverlo su un cilindro casuale richiede da 5 a 10 ms, con un ulteriore ritardo di 5-10ms per aspettare che la testina sia sull'area desiderata.

Molti computer supportano uno schema conosciuto come MEMORIA VIRTUALE, che permette l'esecuzione di programmi più grandi della memoria fisica mettendoli sul disco fisso e usando la memoria principale come una specie di cache. Questo richiede la rimappatura degli indirizzi di memoria, per convertire l'indirizzo che il programma ha generato in un indirizzo fisico della RAM. Questo è ottenuto grazie ad una parte della CPU chiamata MMU (Memory Management Unit). La presenza della cache e della MMU porta ad un forte impatto positivo sulle prestazioni.

### 1.3.3 Dispositivo I/O

I dispositivi I/O sono generalmente suddivisi in due parti: un CONTROLLER ed il DISPOSITIVO STESSO.

- **IL CONTROLLER** è un chip o un insieme di chip che fisicamente controllano il dispositivo, come per esempio deve controllare la disposizione del braccio e della testina dei dischi durante un'operazione di lettura e scrittura nella memoria.

- **IL DISPOSITIVO** ha un'interfaccia abbastanza semplice essendo che non deve compiere grandi lavori e soprattutto deve essere **STANDARD**

Poiché ogni tipo di controller è diverso, c'è bisogno di un software diverso per ognuno per controllarli. Il software che comunica con i controller vengono chiamati DRIVER DEL DISPOSITIVO (DEVICE DRIVER). Per funzionare, ogni

driver deve essere inserito all'interno del sistema operativo in modo da girare in modalità KERNEL.
Esistono tre modi in cui un driver può essere inserito nel Kernel:

- Ricollegare il kernel col driver e poi riavviarlo
- Creare una voce in un file del sistema operativo indicando le sue necessità e riavviarlo
- Fare in modo che il sistema operativo accetti il driver mentre è in esecuzione e lo installi senza la necessità di riavvio.

Per attivare il controller il driver prende un comando dal sistema operativo e lo traduce nei valori appropriati da scrivere nei registri del dispositivo. La raccolta di tutti i registri forma LO SPAZIO DI UNA PORTA DI I/O.
L'input e l'output possono essere gestiti in 3 modi diversi:

- **BUSY WAITING**: un programma utente invia una system call che il kernel traduce in una chiamata di procedura al driver adeguato. Il driver attivo l'I/O e si sofferma in un ciclo rapido e continuo interrogando costantemente il dispositivo per controllare che tutto sia stato eseguito.
- Il driver avvia il dispositivo e gli richiede un interrupt quando termina. A quel punto il driver ritorna, il sistema operativo blocca il chiamante se deve e verifica se ci siano altri lavori in sospeso. Quando il controller rileva la fine del trasferimento, genera un interrupt che segnala il comportamento.
  Gli interrupt sono importanti nei sistemi operativi, quindi vediamone il funzionamento con un processo di I/O:
  Nel primo passo il driver indica al controller cosa fare scrivendo nei suoi registri, poi il controller avvia la macchina. Quando il controller ha terminato la lettura o la scrittura, segnala l'interrupt al chip del controller utilizzando certe linee di bus. Se il controller è preparato a ricevere l'interrupt, fa un appunto nella CPU informandola. Il controller degli interrupt mette il numero del dispositivo nel bus, così che la CPU possa sapere quali dispositivi hanno finito.
  Quando la CPU ha deciso di prendere l'interrupt il PROGRAM COUNTER E IL PSW vengono inseriti nello stack attuale e la CPU girata in modalità Kernel. Il numero del dispositivo può essere usato come indice per rintracciare l'indirizzo del gestore degli interrupt per questo dispositivo. Questa parte di memoria è chiamata VETTORE DEGLI INTERRUPT.
- Il terzo metodo per gestire l'I/O utilizza un hardware speciale: un chip DMA (Direct Memory Access) in grado di controllare il flusso di bit tra la memoria e alcuni controller senza l'intervento costante della CPU.

Gli interrupt possono accadere in momenti decisamente inopportuni, per esempio mentre è in corso un altro interrupt. Per questo motivo la CPU ha un modo per disabilitare gli interrupt e riabilitarli in seguito. Mentre sono disabilitati,

il controller degli interrupt stabilisce le priorità basate sulle priorità statiche assegnate ad ogni dispositivo.

### 1.3.4 Bus

Il **bus** è un canale di comunicazione che permette alle periferiche e ai componenti di un sistema di interfacciarsi tra loro scambiandosi dati o informazioni attraverso la trasmissione e la recezione di segnali.
Il **bus** è un insieme di tracce:

- Le tracce sono sottili collegamenti elettrici che trasportano informazioni.
- Le porte sono dei bus che collegano solo 2 dispositivi
- Un canale I/O è un bus condiviso da diversi dispositivi per eseguire operazioni I/O

Ci sono diversi bus:

- Il bus principale è il bus **PCIe**, fatto da intel per sostituire la PCI. Il bus PCIe può trasferire decine di Gb al secondo. Utilizza connessioni punto a punto dedicate. Utilizza un'architettura seriale e invia tutti i bit di un messaggio attraverso una singola connessione chiamata **lane**.
- L'**USB** (universal serial bus) → inventato per connettere dispositivi lenti. Qualsiasi USB può essere immediatamente collegato al computer e funzionare senza il riavvio del sistema.
- Il bus **SCSI** (small computer system interface) → è un bus ad alte prestazioni pensato per i dispositivi che necessitanop di una grande laghezza di banda.

Il **plug and play** fa sì che il sistema raccolga autonomamente le informazioni sul dispositivo di I/O, assegni centralmente i livelli di interrupt e gli indirizzi I/O e poi indichi a ogni scheda quali sono questi numeri.

### 1.3.5 Avvio del computer

Processo di avvio → **Boot**

Ogni computer ha una scheda madre, essa contiene un programma chiamato **BIOS** (basic input output system). Il BIOS esegue il BOOTSTRAPPING, ovvero il caricamento in memoria di componenti del sistema operativo iniziali.

Una volta partito il BIOS controlla quanta RAM c'è, guarda poi se ci son tastiere o altri dispositivi (guarda che vadano correttamente), fa la scansione dei bus PCI e PCIe per trovare nuovi dispositivi (se i dispositivi sono diversi dall'ultimo avvio vengono configurati) e poi avvia il sistema.
Il BIOS quindi determina il disp. di avvio cercandolo nell'elenco di unità che è nella memoria CMOS.

## 1.4 Sistemi operativi - panoramica

### 1.4.1 Sistemi operativi per Mainframe

- Grandi dimensioni (tipo come una stanza)
- Hanno una grande capacità di I/O, per esempio un mainframe con 1000 dischi non sarebbe una novità
- Offrono generalmente tre tipi di servizi:
  - **batch**: sistema che esegue lavori di routine senza la presenza di utenti che interagiscano con la macchina
  - **elaborazione di transizioni**: trattano una grande numero di richieste minime. Ogni unità di lavoro è piccola ma il sistema ne tratta centinaia o migliaia al secondo
  - **timesharing**: il sistema del timesharing consente a diversi utenti remoti di eseguire lavori sul computer tutti insieme, come se si interrogasse un grande data base.
- Esempio di sistema per mainframe: IBM OS/390, Linux
- Questi sistemi stanno venendo sostituiti da varianti di UNIX. Tipo Linux merda.

### 1.4.2 Sistemi operativi per server

- Girano su server
- Servono molti utenti allo stesso tempo
- Offrono servizi di archiviazione, web server, ISP
- Esempio di sistema operativo per server: Solaris SUN, FreeBSD, Linux, Windowshit server

### 1.4.3 Sistemi operativi Multiprocessore

- I computer multiprocessore sono quelli con molte CPU connesse in un sistema singolo
- A seconda di come sono connesse queste CPU i computer sono chiamati paralleli, multicomputer o multiporcessori
- Sti computer necessitano di speciali sistemi operativi, che spesso sono varianti di sistemi operativi per server con partiolare caratteristiche per comunicazione, connettività e coerenza
- Esempio di sistema operativo per miltiprocessore: Linux, Windows

### 1.4.4 Sistemi operativi per personal computer

- I PC moderni supportano la multiprogrammazione
- Hanno lo scopo di fornire un valido supporto a un utente singolo
- Usati soprattutto per cosa inutili da comuni mortali, tipo scrivere cazzate, fare fogli di calcolo e guardare porno su internet
- Esempio di sistema operativo per PC: Linux, Windows, MacOs, FreeBSD, tutti insomma

### 1.4.5 Sistemi operativi per computer palmari

- Un computer palmare o **PDA** (personal digital assistant) è un piccolo computer che si può tenere in mano
- La maggior parte di questi computer oggi dispone di CPU multicore, GPS, Fotocamera, grandi quantità di memoria e sistemi operativi sofisticati
- Su molti di questi palmari si possono installare applicazioni di terze parti

### 1.4.6 Sistemi operativi integrati

- Chiamati anche *embedded system* girano su computer che non sono pensati per accettare software installato dall'utente.
- Caratterizzati da un insieme limitato di risorse specializzare
- Forniscono funzionalità per vari tipi di dispositivi come cellulari e PDA
- Gestiscono efficentemente delle risorse fondamentali per la costruzione di un buon sistema operativo
- Non accetta software esterno, tutto il software è su ROM. Quindi non c'è alcun bisogno di protezione fra le applicazioni
- Esempio: Tv, microonde, automobili, registratori DVD, ecc. ecc.

### 1.4.7 Sistemi operativi per sensori

Sistemi operativi per la rete di sensori (notato anche da WSN) sono sistemi operativi incorporati nei sensori nella rete (principalmente reti wireless ). Un sensore di rete (che si trova nelle pubblicazioni scientifiche con il termine inglese mote ) è un insieme di elementi elettronici di dimensioni ridotte, composto essenzialmente da un rilevatore , un microcontrollore , una batteria , un ricetrasmettitore e un'antenna.

Un sensore svolge tre funzioni: acquisire i dati, elaborarli e inviare le informazioni relative ai dati al gestore della rete. I dati sono grandezze fisiche dell'ambiente in cui si trova il sensore. Il vantaggio della configurazione di

rete è acquisire una grande quantità di dati per alimentare un'applicazione. L'applicazione reagisce in base ai dati inviati dai sensori e informa su un cambiamento di stato dell'ambiente monitorato. Un sensore ha una durata della batteria limitata. La rete di sensori deve essere resiliente per compensare la fine del ciclo di vita di ogni sensore.

I sistemi operativi per sensori in rete sono specificatamente progettati per ottimizzare l'uso delle limitate risorse hardware a loro disposizione: poca memoria RAM , bassa velocità di elaborazione del processore e poca energia elettrica. Esistono molti sistemi operativi specializzati, tra cui: Contiki , ERIKA Enterprise , Nano-RK , TinyOS , MantisOS, RETOS, Senses, Cormos, LiteOS , NanoQplus.

### 1.4.8 Sistemi operativi real-time

- Per tali sistemi il tempo rappresenta un parametro fondamentale
- Obiettivi con scadenza (deadline)
- Se un'azione deve avvenire in un determinato momento si tratta di un sistema **real-time stretto** (come le pussy delle bambine). Questi sistemi devono fornire garanzie assolute che una specifica azione avvenga in un determinato momento
- Un altro tipo di sistema real-time è il **real-time leggero**, in cui mancare occasionalmente un obiettivo a scadenza è accettabile
- Eseguono solo software inserito dai progettisti software

### 1.4.9 Sistemi operativi per smart card

- I più piccoli sistemi operativi girano su smart card
- Sono molto semplici
- Usati per pagamento elettronico, trasporti, ecc.

### 1.4.10 Sistemi operativi per alto livello di strazione

Occorre definire spciali requisiti di progetto e supporto hardware

- Grande memoria principale
- Hardware per usi speciali
- Grande numero di processi

## 1.5 Concetti base dei sistemi operativi

La maggior parte dei sistemi operativi presenta alcuni concetti di base, come i processi, le astrazioni, gli spazi degli indirizzi e i file.

### 1.5.1 Processi

Un concetto chiave del sistema operativo è quello dei processi, i quali sono fondamentalmente dei programmi in esecuzione.

Associato ad ogni processo c'è il suo **SPAZIO DEGLI INDIRIZZI** (un elenco di locazioni di memoria).
Associato ad ogni processo c'è anche un insieme di risorse, che comprende normalmente i registri, un elenco di file aperti, allarmi in sospeso e tutte le informazioni necessarie per far girare il programma.

In molti sistemi operativi, tutta l'informazione riguardante ciascun processo, diversa dai contenuti del suo **SPAZIO DEGLI INDIRIZZI**, è salvata all'interno di una tabella del sistema operativo chiamata **TABELLA DI PROCESSO**, che è un array di strutture.
Non so se ti interessi saperlo, ma un processo sospeso consiste nel suo spazio degli indirizzi generalmente chiamato **IMMAGINE CORE**.
Le chiamate chiave del sistema di gestione del processo sono quelle che hanno a che fare con la creazione e la chiusura dei processi.

Se un processo può creare uno o più processi questi a loro volta possono creare dei processi figli. I processi relazionati che cooperano tra di loro hanno bisogno comunicazione tra di loro. Questa comunicazione è chiamata COMUNICAZIONE TRA PROCESSI (interprocess communication).

Ad ogni processo viene assegnato lo UID (user identification) dell'utente che lo ha eseguito, stessa cosa vale per i gruppi, con lo GID (group identification).

Superuser (UNIX) o Administrator (in Windows) sono due UID speciale che hanno un potere speciale e possono violare le regole di protezione.

### 1.5.2 File

Un concetto chiave comune di fatto a tutti i sistemi operativi è il **file system**.
Per creare file, cancellarli, leggerli e scriverli sono ovviamente necessarie chiamate di sistema.

Per fornire un luogo dove tenere i file, molti sistemi operativi considerano la **directory** come un modo per raggruppare i file. Per creare e rimuovere directory servono delle chiamate di sistema, sono previste chiamate di sistema per mettere un file esistente in una directory.
Le voci di una directory possono essere file o altre directory.

Le gerarchie del processo e dei file sono entrambe organizzate come alberi, ogni file della gerarchia delle directory può essere specificato attraverso il suo **PATH NAME**, a partire dall'inizio della gerarchia delle directory, la **ROOT DIRECTORY**.

Prima che un file possa essere letto o scritto c'è bisogno di controllare i permessi. Se è consentito l'accesso allora il sistema restituisce un piccolo numero intero chiamato descrittore del file da usare nelle operazioni che seguono. Se l'accesso è proibito viene restituito un codice d'errore.

Un altro concetto importante in UNIX è il **file system montato**, consente ad una unità esterna come CD-ROM o DVD, di montare i proprio file system attraverso una connessione tramite l'albero principale.

Poi abbiamo ancora il **FILE SPECIALE**. I file speciali sono pensati per fa sì che dispositivi I/O siano visti come file. Esistono due tipi di file speciali:

- **FILE SPECIALI A BLOCCHI**: usati per modellare dispositivi costruiti da un insieme di blocchi indirizzabili casualmente, come i dischi
- **FILE SPECIALI A CARATTERI**: usati per modellare stampanti, modem ed altri dispositivi (che accettano una sequenza di caratteri in output)SENZA VITA COME QUELLA PORCODIO DI BALSAMO ESPLODESSE

L'ultima caratteristica importante di UNIX è la PIPE: una specie di pseudofile che può essere usato per connettere due processi.

### 1.5.3 Protezione

Sta al sistema opetativo gestire la sicurezza in modo efficente, facendo in modo per esempio che i file siano consultabili solo da utenti autorizzati.

I file in Unix sono protetti assegnando a ogniuno un **codice di 9 bit**.
Il codice consiste in 3 campi da 3 bit ciascuno:

- Uno per il **proprietario**
- Uno per gli altri **membri del gruppo** a cui il proprietario appartiene
- Uno per tutti gli **altri**

Ogni campo ha poi:
- un bit per la lettura (indicato con r)
- un bit per la scrittura (indicato con w)
- un bit per l'esecuzione (indicato con x)

Questi 3 bit sono conosciuti come **bit rwx**

### 1.5.4 Shell

Il sistema operativo è il codice che realizza le chiamate di sistema.
L'interprete dei comandi Unix viene chiamato **shell**.
La shell non fa parte del sistema operativo ma ne fa un uso intenso, essa è infatti l'interfaccia tra utente da terminale e sistema operativo.

La shell viene avviata quando un utente si collega, usa lo standard input/output come terminali. Comincia mostrando il **prompt**, un carattere simile sl simbolo del dollaro. Questo simbolo avvisa l'utente che la shell è in attesa di ricevere un comando.

## 1.6 Chiamate di sistema

L'interazione tra programmi utente e sistema operativo è carico delle chiamate di sistema. La chiamata di sistema è un processo che viene attivato tramite una trap dall'utente.

Ogni macchina ha un diverso meccanismo per eseguire le chiamate di sistema quindi non si può fare una descrizione che vada bene per tutti. Spesso le chiamate devono essere espresse in codice assembly ma vengono fornite anche diverse librerie per fare chiamate anche con altri linguaggi.

Ogni computer a CPU singola può eseguire un'operazione sola per volta.
Se un processo sta eseguendo un programma utente in modalità utente e necessita di un servizio di sistema deve eseguire un'istruzione di trap (per cambiare da utente a kernel) per trasferire il controllo al sistema operativo (in quanto le chiamate di sistema entrano in modalità kernel). A quel punto il sistema operativo analizzando i parametri capisce cosa voglia il processo, realizza la chiamata e restituisce il controllo all'istruzione successiva.

Esempio di chiamata di sistema $read(fd, buffer, nbytes)$:

1) Il parametro nbytes viene messo nello stack
2) Il parametro &buffer viene messo nello stack
3) Il parametro fd viene messo nello stack. Sono i compilatori C e C++ che mettono i parametri nello stack. Buffer è passato per indirizzo di memoria.
4) Viene chiamata la procedura di libreria → chiamata di read
5) Il numero della chiamata viene messo in un luogo dove il sistema operativo pensa che sia (così è facile trovarlo dio cane), come un registro.
   Viene eseguita la trap → passaggio da utente a kernel
6) Viene avviata l'esecuzione a un indirizzo fisso all'interno del kernel
7) Viene esaminato il numero di chiamata e poi indirizzata al corretto gestore di chiamata
8) Viene eseguito quindi il gestore di chiamata di sistema, una volta completato il lavoro si va al prossimo passaggio
9) Trap → passaggio da kernel a utente. Quindi il controllo viene restituito all'utente
10) La procedura ritorna al programma utente
11) Il programma utente pulisce lo stack.

## 1.7 Struttura di un sistema operativo

### 1.7.1 Sistemi monolitici

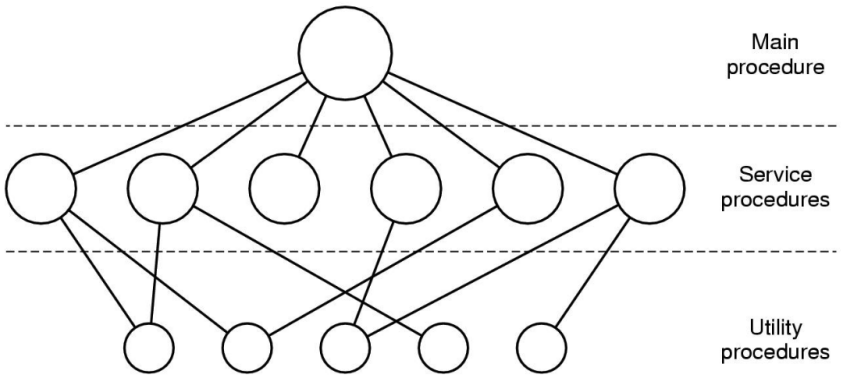
I sistemi **monolitici** sono di gran lunga l'organizzazione più comune.
L'intero sistema operativo viene eseguito come un **programma singolo e in modalità kernel**.

Il sistema operativo è scritto come una **raccolta di procedure**, collegate all'interno di un solo grande programma binario eseguibile. Quando questa tecnica è usata, ogni procedura nel sistema è libera di richiamarne un'altra. Avere migliaia di procedure richiamabili l'una con l'altra senza restrizioni da luogo a un sistema operativo pesante e difficile da capire.
Con questo approccio, per costituire l'effettivo programma oggetto del sistema operativo, prima si compilano tutte le procedure individuali per poi fonderle tutte insieme in un file eseguibile usando il linker di sistema.
(Relativamente all'information hiding non c'è nulla, perché è uno snitch → gang)

Anche all'interno dei sistemi monolitici è tutta via possibile avere qualche struttura. I servizi forniti dal sistema operativo sono richiesti mettendo i parametri in un luogo ben definito e poi eseguendo un'istruzione TRAP. Questa fa passare la macchina in modalità kernel e trasferisce il controllo al sistema operativo, il quale va poi a prendere i parametri e determina quale chiamata di sistema deve essere realizzata.

Questa organizzazione fornisce la seguente struttura base per il sistema operativo.

- Un programma principale che richiama la procedura di servizio richiesta.
- Un insieme di procedure di servizio che realizzano le chiamate di sistema
- Un insieme di procedure di supporto che aiutano le procedure di servizio.



### 1.7.2 Sistemi a livelli

E' L'organizzazione del sistema operativo come una gerarchia di layers. Il primo sistema di questo tipo fu il **THE**, aveva 6 livelli:

0) Si occupava dell'allocazione del processore, scambiava i processi quando si verificava un interrupt o scadeva un timer, forniva multiprogrammazione di base alla CPU.

1) Gestiva la memoria. allocava spazio per i processi nella memoria principale.

2) Gestiva la comunicazione fra ogni processo e la console dell'operatore

3) Gestiva gli I/O e metteva nel buffer il flusso di informazioni da e verso di loro

4) Ospitava i programmi utente, che non dovevano occuparsi di processi, memoria ...

| Layer | Function |
|---|---|
| 5 | The operator |
| 4 | User programs |
| 3 | Input/output management |
| 2 | Operator-process communication |
| 1 | Memory and drum management |
| 0 | Processor allocation and multiprogramming |

### 1.7.3 Sistemi microkernel

Con l'approccio stratificato i progettisti possono scegliere dove definire il confine tra kernel e utente. Tradizionalmente, tutti gli strati andavano nel kernel, ma ciò non è necessario.
L'idea alla base del microkernel è quella di raggiungere un'alta stabilità suddividendo il sistema operativo in piccoli **moduli ben definiti**, uno solo dei quali - il microkernel - girava in modalità kernel.

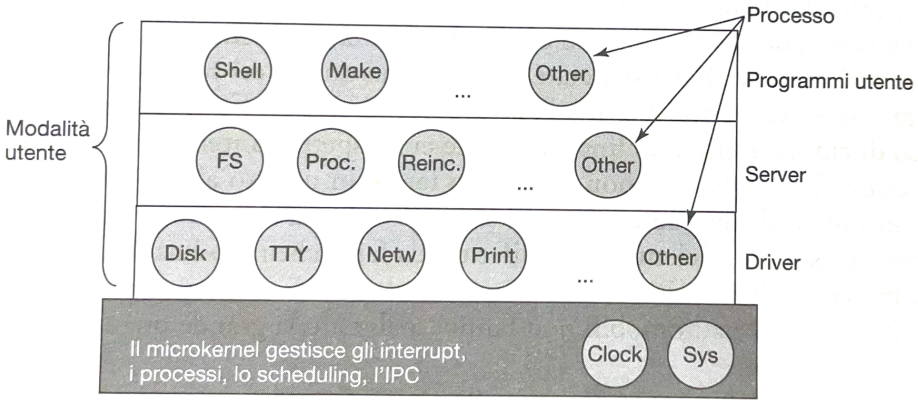
Al di fuori del kernel il sistema è strutturato in **3 strati di processi** che girano tutti in modalità utente:

- **DRIVER DEI DISPOSITIVI**: dato che girano in modalità utente, non hanno accesso fisico allo spazio della porta di I/O e non possono inviare comandi di I/O, per programmare un dispositivo di I/O, il driver costruisce una struttura che indica quali valori scrivere e in che porte di I/O e fa una chiamata al kernel chiedendogli di effettuare la scrittura.

- Al di sopra dei driver c'è un altro livello in modalità utente contenente i **SERVER**, che compiono la maggior parte del lavoro del sistema operativo.
  I programmi utente ottengono servizi del sistema operativo inviando brevi messaggi ai server.
  Un server interessante è il REINCARNATION SERVER, il cui compito è di verificare se gli altri server e driver stanno funzionando correttamente. Nel caso in cui ne riscontri uno non funzionante, viene immediatamente sostituito senza alcuna interruzione per l'utente.

- **PROGRAMMI UTENTE**

Un'idea collegata al kernel minimale è quella di mettere il meccanismo ma non la policy. Il MECCANISMO nel kernel è quello di guardare la priorità più alta, la POLICY, può assegnare la priorità. Grazie a questo il kernel può essere più piccolo.



### 1.7.4 Exokernel

Piuttosto che clonare la macchina reale, come avviene con le macchine virtuali, un'altra strategia è partizionarla, assegnando a ogni utente un sottoinsieme delle risorse.

Il livello base, che gira in modalità kernel, è un programma chiamato **exokernel**. Il suo compito è quello di allocare risorse alle macchine virtuali e poi controllare i tentativi di impiego per essere certo che nessuna macchina provi a usare risorse di un'altra.

Il vantaggio di utilizzare l'exokernel è quello di risparmiare uno strato di corrispondenza.

L'exokernel necessita solo di tenere traccia di quake sia la macchina virtuale a cui è stata assegnata una certa risorsa.
Questo metodo ha ancora il vantaggio di tenere separata la multiprogrammazione dal codice del sistema operativo utente (spazio utente), ma a un costo inferiore, dato che il compito dell'exokernel è far sì che tutte le macchine virtuali siano tra loro indipendenti.

# 2 Capitolo: Processi e thread

## 2.1 Processi

Tutti i computer moderni svolgono spesso molti compiti contemporaneamente. Una volta avviato un sistema vengono fatti partire diversi processi non visibili. Tutte queste attività devono essere gestite e in questo caso la soluzione migliore per la gestione è un sistema multiprogrammato.
In ogni sistema multiprogrammato la CPU passa da processo a processo rapidamente.

Mentre la CPU esegue un solo processo alla volta, nel corso di 1 secondo essa può lavorare su parecchi di loro, dando l'illusione di parallelismo.
Qualche volta si parla infatti di **pseudoparallelismo**, mettendolo in contrapposizione con il vero parallelismo hardware dei sistemi multiprocessore.

### 2.1.1 Il modello di processo

In questo modello tutto il software eseguibile sul computer è organizzato in un certo numero di **processi sequenziali**, detti per comodità processi.
Concettualmente ogni processo ha la sua CPU virtuale. In realtà la CPU passa avanti e indietro da processo a processo.
Questo rapido avanti e indietro è chiamato **multiprogrammazione** → In informatica, esecuzione, da parte di un elaboratore elettronico, di più programmi contemporaneamente.
Dato che la CPU passa rapidamenti da un processo all'altro, la velocità a cui un processo realizza la sua elaborazione non sarà uniforme e probabilmente nemmeno riproducibile se gli stessi processi fossero eseguiti di nuovo.

La differenza tra un processo e un programma è impercettibile ma cruciale:
Un **programma**, è un insieme di bit collocati in un file su disco: non cambia nel tempo, non modifica lo stato della memoria: è quindi un' entità statica.
Un **processo** è un entità dinamica: modifica lo stato della memoria e dei registri.

L'idea chiave in questo caso è che un processo è un'attività di un certo tipo. Ha un programma, degli input, degli output, e uno stato. Un programma, invece, è un oggetto che può essere memorizzato su disco, anche quando non è in esecuzione.

### 2.1.2 Creazione del processo

I sistemi operativi hanno bisogno di un modo per creare i processi. Nei sistemi generici, tuttavia, si ha bisogno di un modo per creare e terminare i processi secondo le necessità durante il funzionamento. Gli eventi che possono causare la creazione di un processo sono quattro:

- **Inizializzazione del sistema**: All'avvio di un sistema operativo, vengono in genere creati parecchi processi. Alcuni sono processi ATTIVI, in primo piano, che interagiscono con gli utenti. Altri sono processi in BACKGROUND, che non sono associati a utenti particolari, ma hanno invece funzioni specifiche. I processi in BACKGROUND sono usati per gestire attività come e-mail, pagine web, news. Vengono chiamati demoni (**deamon**).

- **Esecuzione di una chiamata di sistema di creazione di un processo**: Oltre ai processi creati all'avvio, ne possono in seguito essere creati anche di nuovi. Spesso un processo in esecuzione invierà chiamate di sistema per creare uno o più processi per aiutarlo nello svolgimento del suo lavoro. Su di un MULTIPROCESSORE, il fatto di consentire a ogni processo di essere eseguito su una diversa CPU fa si che il lavoro sia eseguito più velocemente.

- **Richiesta dell'utente di creare un processo**.

- **Inizio di un job in modalità batch**

In UNIX, per creare un processo nuovo esiste una sola chiamata di sistema: il FORK.
Questa chiamata crea un clone esatto del processo che esegue la chiamata. Dopo il FORK i due processi, genitore e figlio, hanno la stessa immagine della memoria, le stesse stringhe di ambiente e i medesimi file aperti.
Solamente il processo figlio poi esegue una EXECVE per cambiare la propria immagine di memoria ed eseguire un nuovo programma.

In Windows, al contrario, una sola chiamata di funzione Win32, CREATEPROCESS, gestisce sia la creazione del processo sia il caricamento del programma corretto nel nuovo processo.

Sia in UNIX che in WINDOWS, dopo la creazione di un processo, il genitore e il figlio hanno i loro spazi degli indirizzi personali. In UNIX lo spazio degli indirizzi iniziale del figlio è una copia di quello del genitore, ma ci sono due spazi degli indirizzi assolutamente distinti; non è condivisa alcuna variabile scrivibile. In alternativa, il figlio può condividere tutta la memoria del genitore, ma in questo caso la memoria è condivisa in modalità COPY-ON-WRITE, ossia se uno dei due desidera modificarne una parte, questa parte viene prima esplicitamente copiata.

### 2.1.3 Chiusura del processo

Una volta creato, un processo comincia a girare ed esegue quello che è il suo compito.
Prima o poi il nuovo processo finirà a causa di una delle condizioni seguenti:

- Uscita normale (condizione volontaria)
- Uscita su errore (condizione volontaria)
- Errore critico (condizione involontaria)
- Terminato da un'altro processo (condizione involontaria)

- La maggior parte dei processi termina perchè ha completato il proprio lavoro.
- La seconda causa della terminazione di un programma è che il processo scopre un **errore critico** (fatal error). Tipo per esempio provare a compilare un file con gcc file.c ma il file non esiste. (devi essere coglione però).
- La terza causa della chiusura è un errore causato dal processo, spesso dovuto a un difetto di programma. Come per esempio l'esecuzione di un instruzione non valida, i il riferimento a una cella di memoria che non esiste.
- La quarta è che il processo fa una chiamata di sistema indicando al sistema operativo di chiuderne altri. In Unix questa chiamata è *kill*.

### 2.1.4 Gerarchie di processi

In alcuni sistemi la creazione di un processo figlio da parte di un processo genitore crea una associazione tra i due. I processi figli possono a loro volta creare altri figli, creando una **gerarchia di processi**.

In Windows la gerarchia di processi non esiste, tutti i processi sono uguali.
L'unico indizio di gerarchia è che quando viene creato un processo, al genitore viene dato un token (chiamato **handle**) che può usare per controllare i figli.

### 2.1.5 Stati di un processo

Sebbene ogni processo sia un'entità indipendente, con il proprio contatore di programma e il proprio stato interno, i processi hanno spesso bisogno di interagire con altri processi.
Un processo può generare alcuni output che un altro processo inizializza come input.
Quando un processo si blocca lo fa perchè logicamente non può proseguire, generalmente perchè in attesa di un input non ancora disponibile. (è anche possibile che un processo si fermi perchè la CPU viene allocata a un altro processo).

Ci sono tre stati in cui può trovarsi un processo:

1) In **esecuzione**: (la CPU è effettivamente in uso in quel momento)

2) **Pronto**: (può essere eseguito, è temporaneamente sospeso per consentire a un altro processo di essere eseguito)

3) **Bloccato**: (incapace di proseguire finchè non avviene un qualche evento esterno)

Logicamente i primi 2 stati sono molto simili, perchè in entrambi il processo continuerà. Il terzo stato invece è diverso perchè il processo non può essere eseguito anche se la CPU non ha nulla da fare.

Fra i tre stati sono disponibili 4 transizioni:

1) Nella prima transizione accade che il processo si blocca perchè in attesa di input. Quindi passa dallo stato di ESECUZIONE a quello di BLOCCATO.

2) Nella seconda lo scheduler prende un altro processo, perciò dallo stato di ESECUZIONE passa a quello di PRONTO nell'attesa che l'altro processo si concluda.

3) La terza transizione si presenta quando tutti i processi che lo scheduler aveva fatto partire prima finiscono. Perciò il nostro processo può nuovamente passare dallo stato di PRONTO a quello di ESECUZIONE.

4) La transizione quattro di svolge quando un processo che era bloccato finalmente può ripartire grazie ad un evento esterno (tipo l'arrivo di un input). Quindi passa dallo stato di BLOCCATO a quello di PRONTO.

### 2.1.6 Implementazione dei processi

Per implementare il modello di processo, il sistema operativo mantiene una tabella, chiamata **tabella dei processi**, organizzata per voci, una per ogni processo.

Questa voce contiene importanti informazioni sullo stato del processo, come il contatore di programma, il puntatore allo stack, l'allocazione della memoria, lo stato dei suoi file aperti e le informazioni relative allo scheduling.

Inoltre è possibile spiegare come sia mantenuta l'illusione di molteplici processi sequenziali su una CPU:
associata ad ogni classe di I/O c'è una posizione chiamata **vettore di interrupt**. Questa contiene l'indirizzo della procedura di servizio dell'interrupt.
Tutti gli interrupt iniziano salvando i registri, spesso nella voce della tabella dei processi del processo attuale. Poi l'informazione spinta nello stack dall'interrupt viene rimossa e il puntatore allo stack è impostato in modo che punti a uno stack temporaneo usato dal gestore del processo.

Azioni come salvare i registri e impostare il puntatore allo stack sono eseguite da una piccola routine in linguaggio assembly.

Questa routine è terminata quando viene chiamata una procedura C per completare il lavoro per uno specifico tipo di interrupt.

### 2.1.7 Modellazione della multiprogrammazione

L'utilizzo della CPU può essere migliorato per mezzo della multiprogrammazione. Se il processo medio esegue calcoli solo per il 20% del tempo in cui risiede in memoria, con cinque processi in memoria contemporaneamente la CPU dovrebbe essere occupata tutto il tempo. Questo modello è irrealisticamente ottimista, presuppone tacitamente che tutti e cinque i processi non siano mai in attesa di I/O nello stesso momento. Considerando l'uso della CPU da un punto di vista statistico, si ottiene un modello migliore. Supponete che un processo impieghi una frazione p del suo tempo in attesa che l'I/O sia completato. Con n processi in memoria contemporaneamente, la probabilità che tutti gli n processi stiano aspettando l'I/O è $p^n$. Quindi:

$$Utilizzo\,CPU\ = 1 - p^n$$

Introduciamo l'immaginazione dell'utilizzo della CPU come una funzione di n, chiamata grado di multiprogrammazione. Se i processi impiegano l'80% del loro tempo in attesa dell'I/O, devono come minimo esserci contemporaneamente 10 processi in memoria perché lo spreco sia sotto il 10%.
Il modello probabilistico appena descritto è solo un'approssimazione. Presuppone che tutti i processi siano indipendenti. Con una singola CPU, non possiamo avere tre processi in esecuzione contemporaneamente, così un processo che diventa pronto mentre la CPU è impegnata dovrà rimanere in attesa. In questo i processi non sono indipendenti.
Benché il modello immaginato sia semplicistico, può in ogni caso essere usato per fare previsioni specifiche, sebbene approssimative, sulle prestazioni della CPU.

## 2.2 Thread

### 2.2.1 Uso dei thread

Ci sono parecchie ragioni per avere questi miniprocessi, chiamati **thread**.
La ragione principale per avere i thread è che in molte applicazioni ci sono molteplici attività contemporanee che possono bloccarsi.

Suddividendo tali applicazioni in molteplici thread sequenziali eseguiti quasi in parallelo il modello di programmazione diventa più semplice.
Invece di pensare a interrupt, timer e cambi di contesto, possiamo pensare a processi paralleli. Solo che con i thread aggiungiamo un nuovo elemento: la capacità per entità parallele di condividere tra loro uno spazio degli indirizzi e tutti i suoi dati.

Un secondo motivo per avere i thread è che poiché sono più leggeri dei processi, sono più facili da creare e cancellare.

Una terza ragione per avere i thread è data dalle prestazioni. I thread non producono un guadagno di prestazioni quando sono CPU bound, ma quando c'è un'attività considerevole di elaborazione.

Si consideri un esempio dell'utilità dei thread, attraverso la gestione di un sito World Wide Web. In ingresso arrivano richieste di pagine e la pagina richiesta è inviata al client. I Web server per migliorare le prestazioni mantenendo in memoria un insieme di pagine usate frequentemente, per eliminare la necessità di caricarle dal disco. Questo insieme è detto CACHE ed è usato analogamente in molti altri contesti.
Un modo per organizzare il Web Server è questo:

- Attraverso un thread, il DISPATCHER. Legge le richieste di lavoro in arrivo dalla rete.
- Dopo aver esaminato la richiesta, sceglie un THREAD WORKER inattivo (bloccato) e gli fa gestire la richiesta.
- Il DISPATCHER risveglia poi il worker dormiente mettendolo in stato di pronto. Quando il worker si sveglia controlla se la richiesta può essere esaudita dalla CACHE, altrimenti avvia un'operazione di READ per prendere la pagina dal disco e si blocca finché l'operazione non viene completata.
- Quando il thread si blocca sull'operazione su disco, viene scelto un altro thread, eventualmente il dispatcher, nell'ottica di assumere più lavoro o eventualmente un altro worker pronto per essere eseguito.

Un secondo esempio in cui i thread sono utili è in applicazioni che devono elaborare grossissime quantità di dati. L'approccio usuale è leggere un blocco di dati, elaborati e poi scriverlo di nuovo. Il problema è che se sono disponibili

solo chiamate di sistema bloccanti, il processo si blocca mentre i dati stanno entrando e uscendo.
I thread offrono una soluzione. Il processo potrebbe essere strutturato con un thread per l'input, un thread per l'elaborazione e uno per l'output. Il thread di input legge i dati nel buffer di input, il thread che esegue l'elaborazione per i dati dal buffer, li elabora e mette i risultati nel buffer di output. Il buffer di output scrive infine questi risultati su disco.

### 2.2.2 Modellazione a thread classico

Il modello a processi si basa su due concetti indipendenti:
Raggruppamento di risorse ed esecuzione:

- **Raggruppamento di risorse:** Un processo può essere visto come un modo per raggruppare risorse relazionate. Un processo ha uno spazio degli indirizzi contenente testo, programma e dati, così come altre risorse, che possono includere file aperti, processi figli, situazioni d'allarme in sospeso e altro.
  → Riunendo questi elementi possono essere gestiti più facilmente.

- **Esecuzione:** Un processo puà essere visto come un thread in esecuzione, abitualmente abbreviato semplicemente come **thread**. Il thread ha un contatore di programma, ha dei registri con le variabili di lavoro, ha uno stack con la storia della sua esecuzione.

Anche se i thread devono essere eseguiti in un processo, i thread e il relativo processo sono concetti differenti:
I processi sono usati per raggruppare risorse, mentre i thread sono entità schedulate per l'esecuzione sulla CPU.
Il valore aggiunto dei thread è quello di consentire molteplici esecuzioni nello stesso ambiente del processo, con un alto grado di indipendenza l'una dall'altra.

Avere più thread in esecuzione parallela su un processo è come avere più processi in parallelo su un computer.
Dato che i thread condividono alcune proprietà dei processi vengono spesso chiamati **processi leggeri**, è inoltre possibile avere più thread eseguiti nello stesso processo, in quest'ultimo caso si parla di **multithreading**.
Il **multithreading** lavora in questo modo: La CPU passa rapidamente avanti e indietro fra i thread, dando l'illusione che i thread siano eseguiti in parallelo.

Thread diversi nello stesso processo non sono così indipendenti come processi diversi. Infatti tutti i thread hanno esattamente lo stesso spazio degli indirizzi, il che significa che condividono anche le stesse varibili globali.

Tra thread non c'è protezione, a differenza dei processi diversi che potrebbero essere di utenti differenti e potrebbero contrapporsi.

Oltre a condividere lo stesso spazio degli indirizzi possono anche condividere lo stesso insieme di file aperti, processi figli, allarmi, segnali e così via.

Di conseguenza l'organizzazione in figura 2.11 dovrebbe essere applicata quando i tre processi sono essenzialmente non relazionati, mentre la figura 2.11 sarebbe appropriata quando i thread sono in effetti parte dello stesso lavoro e sono in stretta e attiva collaborazione l'uno con l'altro.
Gli elementi della prima colonna sono proprietà di un processo, non proprietà di un thread. Per esempio, se un thread apre un file, quel file è visibile agli altri thread nel processo che possono leggerlo e scriverlo. Questo è logico, dato che l'unità di gestione delle risorse è il processo non il thread. Se ogni thread avesse il proprio spazio degli indirizzi personale, i propri file aperti, i propri allarmi in sospeso e così via, sarebbe un processo separato.
Come un processo tradizionale (cioè solo un thread), un thread può trovarsi in uno dei tanti stati: in esecuzione, bloccato, pronto o terminato.
Un thread bloccato è in attesa di qualche evento che lo sblocchi.
Un thread può bloccarsi in attesa di un evento esterno o di essere bloccato da un altro thread. Un thread in stato di pronto viene schedulato per essere eseguito e lo sarà giunto il suo turno.

Con il multithreading i processi normalmente partono con un singolo thread, questo ha la capacità di crearne altri chiamando una procedura di libreria, thread_create, il quale specifica in genere il nome di una procedura che il nuovo thread deve eseguire.

Talvolta i thread sono gerarchici, con relazione genitore-figlio, ma spesso non esiste alcuna relazione e tutti i thread sono uguali.

Quando un thread ha finito il suo lavoro, esso può uscire chiamando la procedura di libreria thread_exit. In alcuni sistemi, un thread può rimanere in attesa che un thread esca, con la procedura thread_join.
Un'altra chiamata comune è thread_yeld, che consente a un thread di rilasciare volontariamente la CPU per consentire l'esecuzione di un altro thread. Questo tipo di chiamata è importante, perché non si verifica un interrupt del clock per attuare effettivamente la multiprogrammazione come avviene con i processi.

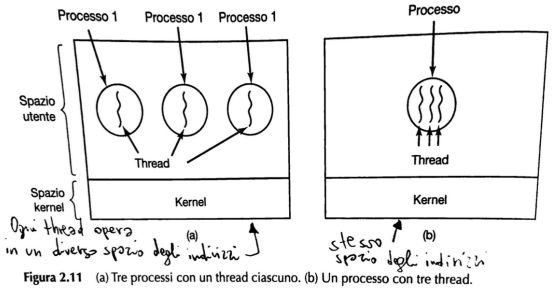


**Figura 2.11** (a) Tre processi con un thread ciascuno. (b) Un processo con tre thread.

### 2.2.3 Thread Posix

Per permettere la scrittura di programmi portabili che usino i thread, IEEE ha definito i thread in uno standard, IEEE 1003.1c. Il package di thread così definito si chiama Pthread.
La maggior parte dei sistemi UNIX li supporta.
Un nuovo thread si crea usando la chiamata pthread create, l'identificatore del thread appena creato è restituito come valore della funzione. Questa chiamata è volutamente del tutto simile alla chiamata di sistema FORK, con l'identificatore del thread che gioca il ruolo del PID, per lo più per identificare i thread cui si fa riferimento in altre chiamate.
Quando un thread ha completato il lavoro che gli è stato assegnato, può terminare chiamando pthread exit.
Prima di continuare, spesso un thread necessita di attendere che un altro thread finisca il proprio lavoro ed esca. Il thread in attesa chiama Pthread join per attendere che un altro specifico thread termini.
Talvolta accade che un thread non sia logicamente bloccato, ma senta di essere stato in esecuzione abbastanza tempo e voglia dare a un altro thread la possibilità di essere eseguito. Può assolvere a questo obiettivo chiamando pthread yeld.
Le due chiamate successive hanno a che fare con gli attributi.
Pthread attr init crea la struttura di attributi associata al thread e la inizializza con i valori di deafult. Questi valori sono modificabili manipolando i campi nella struttura attributi. Infine pthread attr destroy rimuove la struttura attributi di un thread, liberando così la memoria che occupa.

### 2.2.4 Implementare i pacchetti di thread nello spazio utente

Per realizzare un pacchetto di thread esistono principalmente due modalità: nello **spazio utente e spazio kernel**:
Il primo metodo è mettere il pacchetto di thread interamente nello spazio utente. Il kernel non ne è a conoscenza. Per quanto riguarda il kernel, gestisce processi ordinari, a singolo thread. Il primo vantaggio è che può essere implementato un pacchetto di thread utente su di un sistema operativo che non supporta i thread.
Quando i thread sono gestiti nello spazio utente, per tener traccia dei thread in quel processo, ogni processo ha bisogno della sua tabella dei thread personale, tabella analoga alla tabella dei processi nel kernel, fatta eccezione che tiene traccia solo delle proprietà dei thread, come il contatore di programma di ogni thread, puntatore dello stack, i registri, lo stato e così via.
La tabella dei thread è gestita da un sistema run-time. Quando un thread viene passato nello stato di pronto o bloccato, l'informazione necessaria per ripartire viene salvata nella tabella dei thread, esattamente nello stesso modo in cui il kernel salva le informazioni riguardanti i processi nella tabella dei processi.
Quando un thread fa qualcosa che potrebbe causare il suo blocco, richiama una procedura di sistema run-time. Questa procedura controlla per vedere se il

thread deve essere messo in stato bloccato. Se è così, salva i registri del thread nella tabella dei thread, cerca nella tabella un thread pronto da eseguire e ricarica i registri della macchina con i valori salvati del nuovo thread.
Vi è tutta via una differenza chiave con i processi. Quando un thread ha terminato di essere eseguito per il momento, esempio quando si chiama una procedura pthread yeld, il suo codice può salvare le informazioni del thread nella stessa tabella dei thread. Inoltre può richiamare lo scheduler dei thread per prendere un altro thread da eseguire. Le procedure che salvano lo stato dei thread e lo scheduler sono locali, così richiamarle è molto più efficace che fare una chiamata al kernel. Inoltre non occorrono trap ne cambi di contesto. Questo rende lo scheduling dei thread molto veloce.
I thread utente presentano anche altri vantaggi. Essi consentono a ogni processo di avere il proprio algoritmo di scheduling personalizzato.
Nonostante le migliori prestazioni, i pacchetti thread utente hanno alcuni considerevoli problemi. Primo fra tutti c'è il problema di come sono implementate le chiamate di sistema bloccanti. Supponiamo che un thread legga la tastiera prima che sia premuto un tasto. Consentire che il thread gestisca realmente la chiamata di sistema è inaccettabile, dato che fermerebbe tutti i thread.
Uno degli obiettivi principali dell'avere i thread è innanzitutto quello di permettere a ognuno di usare le chiamate bloccati, ma di impedire che un thread bloccato possa interferire con gli altri. Con le chiamate di sistema bloccanti, è difficile vedere come si possa raggiungere questo scopo rapidamente.
Qualcosa di analogo al problema delle chiamate bloccanti è quello dei page fault. I computer possono essere impostati in maniera tale che non tutto il programma sia nella memoria principale contemporaneamente. Se il programma fa una chiamata o salta a un'istruzione non in memoria, si verifica un page fault e il sistema operativo va a prendere l'informazione mancante dal disco.
Se un thread causa un page fault, il kernel, ignorando l'esistenza dei thread, blocca naturalmente l'intero processo finché l'IO sul disco non sia stato completato, sebbene altri thread potrebbero essere eseguibili.
Un altro problema dei pacchetti di thread utente è che, se un thread inizia l'esecuzione, non sarà eseguito alcun altro thread finché il primo thread non rilasci volontariamente la CPU.
Una possibile soluzione al problema dei thread sempre in esecuzione, anche se non ideale, è di lasciare che il sistema run-time richieda a un segnale del clock (interrupt) di dargli il controllo una volta al secondo.
Un'altra ragione contro l'uso dei thread utente, è che i programmatori generalmente vogliono i thread proprio in applicazioni dove i thread si bloccano spesso. Questi thread gestiscono costantemente chiamate di sistema. Una volta che è avvenuto un trap nel kernel per realizzare la chiamata di sistema, spetta ancora al kernel eseguire lo scambio di thread se il vecchio è bloccato; ciò elimina la necessità di fare costantemente chiamate di sistema SELECT che verificano se la chiamata di sistema READ è sicura.

### 2.2.5 Realizzazione dei thread nel kernel

Supponiamo ora che il kernel sia a conoscenza dei thread e li gestisca. Non occorre un sistema run-time ognuno. E non c'è alcuna tabella dei thread in ogni processo.
Il kernel dispone invece di una tabella dei thread che tiene traccia di tutti i thread del sistema. Quando un thread vuole crearne uno nuovo o distruggerne uno esistente, fa una chiamata kernel, che esegue la creazione o la distruzione, aggiornando la tabella dei thread nel kernel. Le informazioni al suo interno sono le stesse dei thread utente, MA ora stanno nel kernel. Tutte le chiamate che potrebbero bloccare un thread sono realizzate come chiamate di sistema, a costi considerevolmente più alti di una chiamata di una procedura di un sistema run-time. Quando un thread si blocca, il kernel ha la possibilità di poter eseguire sia un altro thread dello stesso processo sia un thread di un diverso processo. Con i thread utente il sistema run time tiene in esecuzione i thread del proprio processo finché il kernel non gli porta via la CPU. A causa dei costi relativamente superiori.

### 2.2.6 Thread pop-up

I thread sono spesso utili nei sistemi distibuiti.
Ad esempio nella gestione dei messaggi in ingresso, per esempio la richiesta di un servizio:
L'approccio tradizionale è quello che un processo o un thread siano bloccati su una chiamata di sistema *receive* per un messaggio in arrivo. Ricevuto il messaggio, lo accetta, lo apre, lo esamina e poi lo elabora.
Tuttavia è possibile anche utilizzare un approccio diverso, in cui l'arrivo di un messaggio causa la creazione e la conseguente gestione di un nuovo thread da parte del sistema. Questo thread è chiamato **thread pop-up**.

Un vantaggio chiave dei thread pop-up è che, visto che sono marcati come nuovi, essi non hanno alcuna storia che debba essere recuperata. Ogni thread viene avviato da zero e ogniuno è identico all'altro (perciò creare di così è molto semplice e rapido).

Un messaggio in arrivo viene passato al nuovo thread per essere elaborato, e grazie a questi thread pop-up la latenza tra l'arrivo del messaggio e l'inizio della sua elaborazione è breve.

Di solito è più veloce e esemplice far girare questi tipi di thread all'interno dello spazio kernel piuttosto che quello utente. In più dallo spazio kernel il thread può accedere facilmente a tutte le tabelle del kernel.
Dall'altra parte, un thread difettoso nel kernel può far più danni di un thread difettoso in modalità utente.

### 2.2.7 Trasformazione di codice a singolo thread in codice multithread

Molti programmi sono stati scritti per processi a singolo thread. Convertirli al multithreading può essere più completo di quanto non sembri.

cagasbura.

## 2.3 Scheduling

Quando un computer è multiprogrammato, ha molti processi o thread che competono per ottenere la CPU nello stesso istante.

Nel caso in cui solo una CPU sia disponibile deve essere effettuata una scelta, quale processo eseguire come successivo?
La parte del S.O. che svolge questa scelta è lo **scheduler**, e l'algoritmo è chiamato **algoritmo di scheduling**.

→ Lo scheduler è la parte di sistema operativo che sceglie quale processo eseguire.

### 2.3.1 Introduzione allo scheduling

All'epoca dei sistemi batch l'algoritmo di scheduling era molto semplice: eseguire il lavoro successivo sul nastro.

Con i sistemi a multiprogrammazione, l'algoritmo di scheduling divenne più complesso perchè c'erano generalmente molteplici utenti in attesa di servizio.

### 2.3.2 Comportamento dei processi

Si definiscono **compute-bound o CPU-bound** i processi che sfruttano pesantemente le risorse computazionali del processore, ma non richiedono servizi di ingresso/uscita dati al sistema operativo in quantità rilevanti.

**I/O-bound** si riferisce alla condizione in cui il tempo necessario a compiere un'elaborazione e determinato principalmente dall'attesa delle operazioni input/output. Ciò si contrappone ai processi CPU-bound.

### 2.3.3 Quando effettuare lo scheduling

### 2.3.4 Categorie di algoritmi di scheduling

Non sorprende che in determinati ambiti serva un diverso tipo di algoritmo di scheduling.
Questo si verifica perchè diverse aree di applicazione hanno obiettivi diversi.

Tre ambienti che vale la pena distinguere sono:

- batch
- interattivo
- real-time

I sistemi batch sono ancora utilizzati nel mondo degli affari per l'elaborazione di paghe, inentari, esecuzione di accrediti ecc. ecc. (tutte attività periodiche). Nei sistemi batch non ci sono utenti impazienti in attesa di una risposta rapida a una piccola richiesta. Di conseguenza sono spesso accettabili algoritmi di scheduling **non preemptive** o algoritmi **preemptive** con lunghi periodi per ciascun processo. Gli algoritmi batch sono spesso abbastanza generali e applicabili ugualmente ad altre situazioni.
Gli obiettivi dei sistemi batch sono:
- minimizzare il numero di job per ora (Throughput)
- ridurre al minimo il tempo da quando un lavoro è sottoposto alla sua fine (Tempo di turnaround)
- mantenere la CPU sempre impegnata (Utilizzo della CPU)

In un ambiente con utenti iterattivi, l'uso della **preemption** è essenziale per evitare che un processo monopolizzi la CPU e neghi il servizio agli altri. Il principale obiettivo dei sistemi interattivi è quello di minimizzare il **tempo di risposta**.
Gli obiettivi sono:
- risponderem alle richieste rapidamente (Tempo di risposta)
- far fronte alle aspettative dell'utente (Adeguatezza)

I sistemi con vincoli real-time invece stranamente non necessitano sempre della **preemptive**, poichè i processi sanno che non possono essere eseguiti per lunghi periodi di tempo e generalmente fanno il loro lavoro e si bloccano. La differenza tra sistemi real-time e interattivi è che i real-time eseguono solo programmi destinati a promuovere l'applicazione a portata di mano.
Gli obiettivi sono:
- evitare la perdita di dati (Rispetto delle scadenze)
- evitare il degrado della qualità nei sistemi multimediali (Prevedibilità)

Obiettivi di tutti i sistemi:
- dare a ogni processo una equa condivisa della CPU (Equità)

- assicurarsi che la policy dichiarata sia attuata(Applicazione della policy)
- tenere impegnate tutte le parti del sistema (Bilanciamento)

### 2.3.5 Scheduling nei sistemi batch

Alcuni algoritmi sono utilizzati sia nei sitemi batch che in in quelli interattivi.

- **Algoritmo First Come First Served**: (FIFO) Si utilizza nei processi **non-preemptive**. In questo algoritmo i processi vengono trattati in base al tempo di arrivo. Quindi c'è una singola coda di processi in stato di pronto, il primo di questa coda entra nel sistema viene eseguito e può continuare ad essere eseguito fin che vuole senza essere interrotto.
  Quando il processo in corso si blocca viene eseguito il successivo. Quando un processo bloccato ritorna printi viene messo in forndo alla coda come processo appena arrivato.
  - Questo è lo schema più semplice di tutti.
  - Senza prelazione
  - Utilizzat raramente come algoritmo principale di scheduling

- **Algoritmo Shortest job First**: (SJF) Si utilizza nei processi **non-preemptive**. In questo algoritmo i processi vengono trattati in base al minimo tempo per terminare. Quindi quando ci sono diversi lavori equivalenti in coda di input, lo scheduler preleva per primo il lavoro più breve. Il SJF è ottimale solo quando tutti i lavori sono disponibili contemporaneamente.
  - Alcune volte la stima non potrebbe essere corretta, però è comunque possibile correggerla.
  - Il tempo di attesa dalla terminazione di un processo e l'inizio di un nuovo processo è minore rispetto all'algoritmo FIFO.

- **Algoritmo Shortest Remaining Time First**: (SRT) Si utilizza nei processi **preemptive** (questo è una versione dell'algoritmo shortest job first). Lo scheduler sceglie sempre il processo il cui tempo che manca alla fine dell'esecuzione è più breve.
  All'arrivo di un nuovo lavoro il suo tempo totale è confrontato al tempo restante dei processi attuali, se il nuovo lavoro ha bisogno di minor tempo del processo attuale per terminare, il processo attuale viene sospeso ed è avviato il nuovo lavoro.
  - I processi molto lunghi spesso aspettano molto per finire di eseguire
  - Non è sempre ottimale: I processi in arrivo corti possono effettuare prelazione su processi quasi completi.

### 2.3.6 Scheduling nei sistemi interattivi

Frequenti nei personal computer, nei server e anche in altri tipi di sistema.

- **Algoritmo Round-Robin**: (RR) Uno degli algoritmi più semplici, vecchi, equilibrati e maggiormente utilizzati.
  A ogni processo è assegnato un intervallo di tempo, chiamato quanto. Se alla fine del quanto di tempo il processo è ancora in esecuzione, la CPU viene prelazionata e assegnata a un altro processo.
  I processi vengono attivati in modalità FIFO ma viene loro assegnato un tempo di processo (il **quanto**). Se un processo non viene completato nel tempo prestabilito viene messo in cosa alla lista dei processi.
  - Facile da implementare
  - Richiede al sistema di mantenere parecchi processi in memoria per minimizzare l'overhead
  - Spesso utilizzato come parte di algoritmi più complessi
  - Spesso usato per sistemi interattivi

- **Algoritmo di Scheduling di priorità**: (Priority Scheduling) A ciascun processo è assegnata una priorità e il processo eseguibile con la priorità più alta è quello a cui è consentita l'esecuzione.
  Per impedire che i processi ad alta priorità siano eseguiti indefinitivamente, lo scheduler, può abbassare la priorità del processo attualmente in esecuzione a ogni scatto del clock (cioè a ogni interrupt).
  Se questa azione fa si che la sua priorità vada al di sotto di quella del processo successivo, avviene uno scambio di processo. In alternativa a ciascun processo può essere assegnto un quanto di tempo massimo in cui essere eseguito (quando il quanto termina è data la possibilità di esecuzione al processo con priorità più alta successivo).
  Le priorità possono essere assegnate staticamente o dinamicamente:
  - **priorità statica:** attribuita ai processi nell'atto della loro creazione in base alle loro caratteristiche o politiche riferite al tipo di utente.
  - **priorità dinamica:** modificata durante l'esecuzione del processo.

  Spesso è conveniente raggruppare i processi in classi di priorità e usare lo scheduling a priorità fra le classi, ma all'interno di ciascuna classe usare lo scheduling RR.

- **Algoritmo Selfish Round Robin**: (SRR) Usa l'invecchiamento delle priorità dei processi per farle crescere gradualmente nel tempo. Ci sono due code, una in attesa della $a$ e una dei processi detta $b$. Le due velocità fanno si che lo scheduler SRR possa spaziare tra lo schema FIFO e RR.
  L'obiettivo del SRR è di fornire un servizio migliore ai processi in esecuzione da un po' rispetto ai nuovi arrivati.
  Come detto ci sono 2 code: a (Attivo), b (In attesa (holding)).
  Un processo entra nella coda dei processi nuovi (in attesa) e invecchiando

la sua priorità aumenta
Quando la sua priorità è uguale a quella dei processi pronti (attivi) entra nella coda pronti e si applica il RR.

- Favorisce i processi più anziani al fine di evitare ritardi irragionevoli.
- Possibili diverse velocità di crescita della priorità.

- **Algoritmo Highest-Response-Ratio-Next**: (HRRN) Politica di scheduling **non-preemptive**. Lo scheduler stima la priorità di ogni processo non solo in base al tempo di servizio, ma anche al tempo speso in attesa di quel servizio.
  - Migliora lo scheduling SJF
  - Ancora senza prelazione
  - Considera anche quanto a lungo un processo ha aspettato
  - Previene l'attesa infinita
  - Tempo di risposta = (Tempo di attesa + Tempo di esecuzione)/Tempo di esecuzione

- **Code Multilivello con Feedback**: Questa politica di scheduling utilizza le code multilivello per capire se un processo è I/O vound o CPU bound. Attraverso le code i processi interattivi escono immediatamente mentre quelli che hanno bisogno di diversi cicli per essere completati finiscono in code sempre più profonde.
  Diversi processi hanno diverse necessità:
  - Processi corti e I/O bound dovrebbero essere eseguiti prima dei processi CPU bound e batch.
  - I modelli di comportamento non sono immeduatamente evidenti allo scheduler.

  Code multilivello con feedback:
  - I processi che arrivano entrano nella coda di **più alto livello** e sono eseguiti con **priorità maggiore** rispetto ai processi nelle code inferiori.
  - I **processi lunghi scendono** a livelli più bassi più volte. Questo fornisce maggior priorità ai processi brevi e I/O bound. I processi lunghi sono eseguiti quando quelli brevi e I/O bound sono terminati.
  - I processi in ogni coda sono serviti utilizzando FIFO (livelli alti) o RR (sempre l'ultimo livello). I processi entrano in una coda ad alta priorità e forzano la prelazione sui processi in esecuzione.

  L'algoritmo deve rispondere ai cambiamenti dell'ambiente. Sposta i processi in altre code quando alternano tra il comportamento interattivo e batch.

  Meccanismi adattivo richiedono un maggior overhead che spesso viene compensato da una maggiore sensibilità ai cambiamenti di comportamento dei process.

- **Algoritmo Fair-Share**: (FSS) In questo algoritmo a ogni utente viene assegnata una frazione di CPU e lo scheduler raccoglie i processi in modo tale da farla rispettare. Quindi se a 2 utenti è stato assegnato il 50% della CPU la avranno (indipendentemente da quanti processi abbiano attivi).

### 2.3.7 Scheduling nei sistemi real-time

Un sistema real-time è un sistema in cui il tempo gioca un ruolo essenziale.
Tipicamente uno o più dispositivi fisici esterni al computer generano degli stimoli a cui il computer deve reagire appropriatamente in un intervallo di tempo determinato (i processi hanno quindi vincoli temporali).
I sistemi real-time sono generalmente divisi in due categorie:

- **Soft real-time**: Non garantisce che i vincoli temporali siano soddisfatti
- **Hard real-time**: I vincoli temporali devono essere sempre soddisfatti. Il mancato rispetto di scadenza potrebbe avere risultati catastrofici. Periodi o asincroni.

Entrambi i sistemi si ottengono dividendo il programma in un certo numero di processi, il comportamento di ognuno dei quali è prevedibile e conosciuto in anticipo. Gli eventi cui un sistema real-time può dover rispondere possono ulteriormente essere categorizzati in:

- **Periodici**: eventi che avvengono in modo regolare.
- **Non periodici**: eventi che avvengono non in modo regolare.

Gli algoritmi di scheduling real-time possono essere **statici e dinamici**:

- Scheduling real-time **statici**: prendono le loro decisioni di scheduling prima che il sistema inizi l'esecuzione (basso overhead). Algoritmo adatto per sistemi dove le condizione cambiano raramente, come nel caso dei sistemi hard real-time. La priorità può essere assegnata secondo due algoritmi:
  - Scheduling **Rate-monotonic**: consiste nell'assegnare ad ogni processo una priorità direttamente proporzionale alla propria frequenza, in modo che ai processi con periodi più brevi venga assegnata una priorità più elevata. Poichè i periodi dei processi sono assunti costanti, l'algoritmo RM è un algoritmo di tipo statico, nel senso che le priorità assegnate ai processi sono fissate all'inizio e rimangono invariate per tutta la durata dell'applicazione. Inoltre RM è un algoritmo intrinsecamente di tipo preemptive.
  - Scheduling Deadline RM: utile per un processo periodico che ha una scadenza diversa dal suo periodo.
- Scheduling real-time **dinamici**: prendono le loro decisioni di scheduling durante l'esecuzione. Questo algoritmo può portare ad un overhead significativo, e deve garantire che l'overhead non porti ad un aumento di

scandenze mancate. Le proritù dei processi sono di solito basate sulle scadenze dei processi:

- **Earliest-deadline-first**: lo scheduler seleziona come prossimo processo da eseguire quello con la distanza minore dalla sua deadline. Con questo algoritmo viene massimizzato il throughput e minimizzato il tempo di attesa. (con prelazione).
- Maximum-laxity-first: l'algoritmo MLF è simile all'algoritmo EDF, ma al contrario di quest'ultimo esso basa la priorità sulla lassità. La lassità misura il tempo alla scadenza del processo e il suo tempo rimanente di esecuzione(C).

  $$L = D - (T + C)$$

  dove D = deadline, T = tempo corrente.

**Scheduling deadline** Nello scheduling deadline (o *scadenza* ) il sistema deve pianificare la gestione dei processi in base al deadline ovvero al loro tempo di completamento predeterminato. Questo algoritmo viene utilizzato quando i risultati sarebbero inutili se non consegnati in tempo.
Uno dei difetti principali questo scheduling è la difficoltà di implementazione:

- lo scheduler deve conoscere i requisiti delle risorse dei processi in anticipo;
- comporta un notevole overhead;
- il servizio offerto ad altri processi può degradare.

Teoricamente dovrebbe allocare prima i processi a a scadenza più vicina.

### 2.3.8 Politica e meccanismi di scheduling

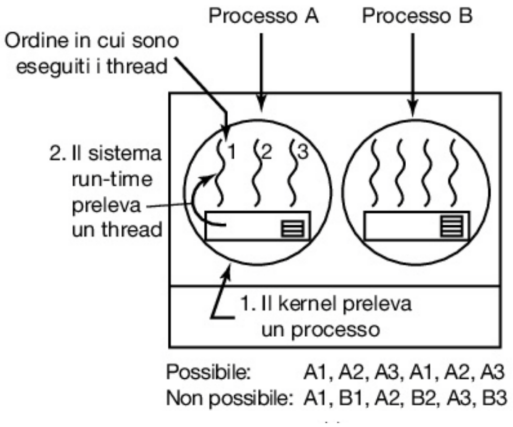
Un'importante distinzione che vale in generale è quella fra la politica e il meccanismo di scheduling. Da un lato ho il meccanismo di scheduling che mi dice come devo implementare, dall'altro ho la politica che mi dice le quali procedure seguire per schedulare.
Il meccanismo risiede solitamente nel nucleo, mentre la politica viene definita dai processi utente

### 2.3.9 Scheduling a thread

Quando molti processi hanno ognuno molteplici thread, abbiamo due livelli di parallelismo presenti: processi e thread. Lo scheduling in questi sistemi si differenzia sostanzialmente a seconda che siano supportati thread utente o a livello kernel (o entrambi).

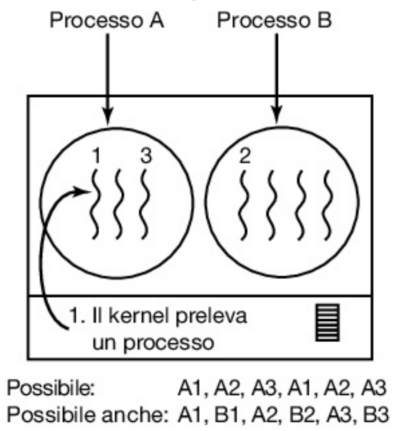
**Scheduling thread utente**: Supponiamo di avere un possibile scheduling di thread a livello utente con le seguenti caratteristiche:

- un quanto di processo della durata di 50msec.
- il thread viene eseguito per 5msec per ogni CPU burst (per ogni porzione di CPU assegnata al processo).




**Scheduling thread kernel**: Supponiamo di avere un possibile scheduling di thread a livello kernel con le seguenti caratteristiche:

- un quanto di processo della durata di 50msec.
- il thread viene eseguito per 5msec per ogni CPU burst (per ogni porzione di CPU assegnata al processo).



# 3 Capitolo: Gestione della memoria

In questa sezione andremo a toccare i seguenti punti, alcuni sono già stati introdotti e trattati in sezioni precedenti:

- Necessità di gestione memoria reale (fisica);
- Gerarchie di memoria;
- Allocazione di memoria contigua e non contigua;
- Multiprogrammazione con partizione fisse e variabile;
- Swapping;
- Strategia di posizionamento della memoria;
- Memoria virtuale
- Vantaggi e svantaggi della paginazione a previsione
- Vantaggi e svantaggi della paginazione a richiesta
- Problemi nella sostituzione delle pagine;
- Confronto tra strategie di sostituzione delle pagine e ottimizzazione;
- Impatto della dimensione di una pagina sulle prestazioni della memoria virtuale;
- Comportamento di un programma nella paginazione;

## 3.1 Organizzazione della memoria

La memoria pu‘o essere organizzata in vari modi:

- Possiamo avere che un processo utilizza tutto lo spazio di memoria
- In alternativa ogni processo ottiene una propria partizione in memoria, che pu‘o essere allocata in modo *statico* o *dinamico*.

In generale ogni processo avr‘a dei requisiti di memoria. ‘E corretto osservare che anche se le memorie sono diventate con il tempo sempre pi‘u veloci, efficienti ed economiche. Tutti i sistemi di elaborazione oramai tendono ad avere una memoria centrale pi‘u grande ma dall'altra parte i software sviluppati richiedono sempre pi‘u quantit‘a di memoria: ogni caso la memoria ‘e una risorsa critica che necessita in gestione accurata e attenta.
Gli utenti e i programmatori vorrebbero che le memorie fisse fossero: grandi, veloci e non volatili, ma non ‘e possibile avere tutte queste caratteristiche in un'unica memoria.
Grazie alle gerarchie di memoria siamo in grado di ottenere le caratteristiche prima elencate in modo intelligente: organizziamo le varie memorie presenti nel sistema secondo delle regole che ci permettono di utilizzarle nel miglior modo.
Solitamente le memorie presenti sono:

- Memoria cache: solitamente localizzata sul processore stesso. Essa memorizza i dati pi‘u usati in modo da avere un accesso pi‘u veloce.
  Una cache di piccole dimensioni è utile per migliore le prestazioni poichè sfrutta il principio della località temporale.
  Caratteristiche:
  - Di piccole dimensione
  - Veloce
  - Molto costosa
- Memoria principale (RAM): dovrebbe memorizzare solo i programmi e dati necessari dei processi al momento della loro esecuzione;
  Caratteristiche:
  - Velocit‘a media
  - Prezzo adeguato
- Memoria secondaria: memorizza dati e programmi che non sono necessari al momento;
  Caratteristiche:
  - Grande spazio di archiviazione, ormai nel ordine dei TB
  - Lenta
  - Economica

Per vedere la piramide delle gerarchie delle memorie vedere sezione Componenti hardware.

### 3.1.1 Gestione della memoria

Per ottimizzare le prestazione delle memoria è necessario implementare delle strategie che poi verranno eseguite dal gestore di memoria che considererà le gerarchie di memoria.
Per implementare queste strategie è il gestore della memoria deve rispondere alle seguenti domande:

- Quale processo rimarrà in memoria?
- A quanta memoria ogni processo ha accesso?
- Dove posizionare in memoria ogni processo?

### 3.1.2 Strategie di gestione della memoria

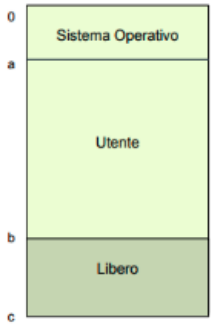
## 3.2 Allocazione di memoria contigua e non contigua

Esistono due modi per allocare i programmi in memoria: in maniera contigua e in maniera non contigua.

- Allocazione contigua: un programma deve essere memorizzato come un unico blocco di indirizzi contigui (basso overhead). Può essere impossibile trovare un blocco abbastanza grande in memoria per caricare il programma.
- Allocazione non contigua: in questa tipologia di allocazione divido lo spazio richiesto per allocare il programma in blocchi chiamati segmenti. Ogni segmento può essere allocato in diverse parti della memoria. In questo modo è più facile trovare "buchi" in cui un segmento possa essere memorizzato poichè la grandezza di un segmento non è molto grande. Questa metodologia di allocazione mi permette di avere più processi presenti contemporaneamente in memoria.

### 3.2.1 Allocazione di memoria contigua mono-utente

Nel caso più semplice abbiamo un utente che ha il controllo dell'intero sistema. Nel sistema non è presente nessun modello di astrazione della memoria. Non c'è alcun problema di nella condivisione di risorse fra utenti poichè ne è presente solo uno e non c'è alcun sistema operativo. Questo modello risale ai tempi in cui non c'era l'S.O. E il programmatore scriveva il codice per eseguire la gestione delle risorse includendo I/O a livello macchina. Successivamente è stato sviluppato il sistema di controllo dell'I/O chiamato **Input-Output Control System** (IOCS): sono delle librerie di codice già pronto per gestire i dispositivi I/O (siamo ancora nell'era del pre S.O.).
Visione della allocazione di memoria contigua mono utente:




Questa è la divisione più semplice possibile della allocazione di memoria contigua mono-utente dove ho: il sistema operativo nella parte bassa della memoria, lo spazio utilizzato dall'utente in modo contiguo (da $a$ a $b$) e una zona libera.

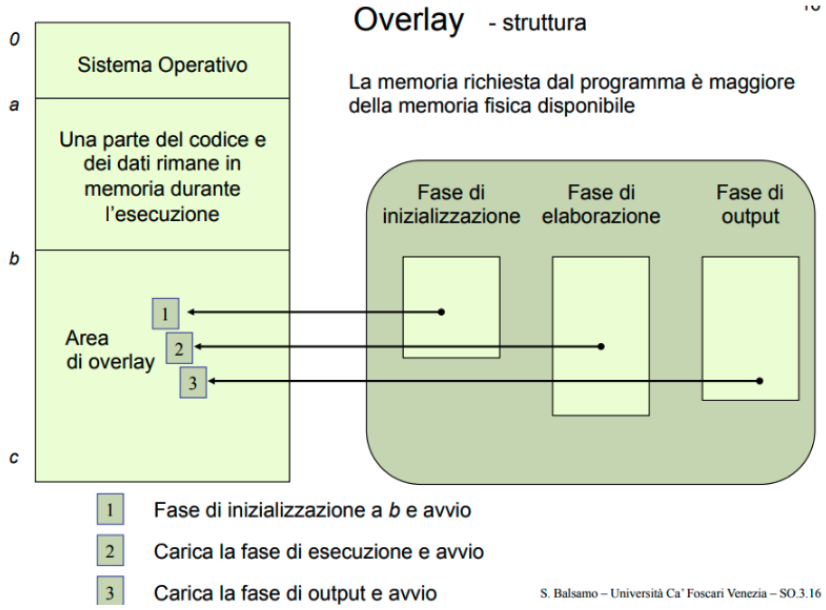
### 3.2.2 Tecnica di Overlay

L'**overlay** è una tecnica di programmazione sviluppata per superare i limiti dell'allocazione contigua: permette all'utente di richiede più dello spazio disponibile. Nell'overlay il programma è diviso in *sezioni logiche*; vengono memorizzate soltanto le *sezioni attive* al momento.
Questa tecnica ha diversi svantaggi:

- Difficile organizzare le sovrapposizioni per utilizzare in modo efficace la memoria principale;
- Complica la modifica ai programmi;

La memoria virtuale ha un obbiettivo simile all'overlay che è quello di proteggere i programmatori di questioni complesse come la gestione della memoria.

**Struttura overlay**




Il programma caricato in memoria comprende: una parte del codice e dati che rimane in memoria per tutta l'esecuzione e tutte le sue sezione logiche presenti nell'area di overlay. Nell'area di overlay vengono caricati in momenti diversi parti diverse (sezione logiche) del programma.
Inizialmente in memoria vengono caricate: la parte del codice e dati (da $a$ a $b$) e il primo segmento (*Fase di inizializzazione*) ed poi viene inziato il programma. Successivamente in ordine vengono caricati la *Fase di elaborazione* e infine la *Fase di output*, ogni volta sovrascrivendo la sezione logica già presente nell'area di overlay.

### 3.2.3 Protezione in ambiene mono-utente

Il sistema operativo non deve essere danneggiato da programmi quindi è necessario implementare una protezione:

- Il sistema non pu'o funzionare se il sistema operativo viene sovrascritto;
- E' sufficiente avere come protezione i registri limite (boundary) che indicano dove inizia lo spazio libero a disposizione del processo

## 3.3 Single-Stream Batch Processing

I primi sistemi richidevano un tempo di setup rilevante provocando uno spreco di tempo e risorse. Automatizzando il setup e teardowm viene aumentata l'efficienza.
Il batch processing è il processore con un flusso di job stream letti in job controllo language (linguaggi oche definisce ogni job e come configuralo).

## 3.4 Allocazione a partizioni fisse

La memoria principale viene divisa in un **numero fisso di** $n$ **di aree** (dove $n$ definisce il livello di multiprogrammazione) dette *partizioni* di dimensione possibilmente diverse. Ogni partizione contiene un processo attivo ed è identifica da coppia di registri *base-limite*, dove il primo contiene l'indirizzo in memoria dove comincia la partizione mentre il secondo contiene la sua dimensione.
Quando c'è una partizione libera, un processo viene caricato in essa ed è pronto per essere schedulato per l'esecuzione.
Prima di attivare un processo è necessario conosce la sua dimensione in modo da capire dove posizionarlo.
I *processi piccoli* causano spesso un spreco di spazio di memoria mentre quelli più *grandi* della partizione più grande i memoria non possono essere eseguiti (aumentando la dimensione delle partizioni avremo un diminuzione del grado di multiprogrammazione).
Il S.O. utilizza delle *code di input* per scegliere come allocare le partizioni del processi: **coda singola per partizione** oppure **coda singola per tutte le partizioni** Gli svantaggi della partizione fisse sono:

- **Frammentazione interna**
  - Il processo non occupa un'intera partizione, sprecando memoria;
  - Impossibilità di usare parte della memoria libera;
  - Possibilità di avere processi troppo grandi da non per poter essere inseriti in nessuna parte;
- **Maggior overhead**: compensati da *maggior utilizzo* delle risorse;

## 3.5 Allocazione a partizioni multiple variabili

Un'alternativa alle partizione fisse: progetto con **partizioni variabili**.
Il S.O. tiene traccia in una tabella di quali parti della memoria sono occupate e quali no come: il numero, la dimensione e le posizioni della partizioni allocate variano dinamicamente.
Quando un processo, arriva il gestore di memoria cerca nell'insieme una partizione libera abbastanza grande per contenrerlo completamente e la "ritaglia" a misura del processo. Il S.O. operativo oltre ad avere la tabella delle partizioni ha pure una coda dei processi pronti: al primo processo viene assegnata la memoria se esiste una partizione abbastanza grande, in caso contrario attende oppure procede al processo successivo.
La partizione da usare viene scelta fra quelle abbastanza grandi secondo una strategia di posizionamento che può essere: *first-fit*, *best-fit* oppure *worst-fit* (la peggiore in termini di tempi).
Alla lunga i processi vengono allocati lo spazio libero apparirà suddiviso in piccole aree: questo è il fenomeno della **frammentazione esterna**.
Esistono diversi modi per combattere questo fenomeno:

- **Coalescenza**: combinare i blocchi liberi adiacenti in un unico grande blocco; spesso non è sufficiente per recuperare quantità di memoria realmente significative.
- **Compattazione** (o *garbage collection*): riorganizzare la memoria in unico blocco contiguo di spazio libero e in un unico blocco contiguo di spazio occupato, rendendo tutto lo spazio libero disponibile. L'overhead in questo caso è significativo.

## 3.6 Gestione della memoria libera

L'allocazione dinamica della memoria libera richiede la *gestione della memoria libera* attraverso una **mappa di bit** oppure attraverso le **liste collegate**.

- **Mappa di bit**: la memoria viene organizzata in *unità* e ogni unità corrisponde ad un bit sulla mappa.
- **Liste collegate**: sono delle liste di segmenti di memoria ognuno dei quali è allocato ad un processo o è libero. Ogni elemento della lista indica:
  - se allocato ad un processo (P) o vuoto (H);
  - l'indirizzo da cui parte;
  - la lunghezza;
  - il puntatore all'elemento successivo.

## 3.7 Un po' di cagate

### 3.7.1 Concorrenza

caratteristica dei sistemi di elaborazione nei quali può verificarsi che un insieme di processi o sotto-processi (thread) sia in esecuzione nello stesso istante. Tale sistema prende il nome di sistema a concorrenza.

### 3.7.2 Process control block (PCB)

struttura dati di un processo, del nucleo del sistema operativo, che contiene tutte le informazioni essenziali per la gestione del processo stesso.

### 3.7.3 Nucleo

software che contiene le componenti fondamentali del sistema operativo che sono:

- Processor scheduler
- Gestore della memoria
- Gestore della Interprocess communication (IPC)
- Gestore del File system

### 3.7.4 Pseudoparallelismo

la CPU esegue un solo processo alla volta, nel corso di 1 secondo può lavorare su parecchi di loro, dando l'illusione di parallelismo. Mettendolo in contrapposizione con il vero parallelismo hardware dei sistemi multiprocessore che hanno in realtà due o più CPU che condividono la medesima memoria fisica permettendo quindi di eseguire realmente in parallelismo.

### 3.7.5 Multithreading

possibilità di molteplici thread nello stesso processo. Alcune CPU hanno il supporto hardware diretto per il multithreading e consentono che lo scambio fra i thread avvenga in tempi dell'ordine dei nano-secondi.

### 3.7.6 Multiprogrammazione vs time sharing

Un sistema che adotta la multiprogrammazione e un sistema che adotta il timesharing non implicano lo stesso concetto. Un sistema multiprogrammato permette l'esecuzione di più processi/thread contemporaneamente, dando l'impressione che ci siano più CPU anche se ne è presente solo una fisicamente. Difatti non è esattamente così: la CPU passa l'esecuzione da un processo ad un altro dando l'impressione di eseguirli contemporaneamente ma in realtà solo uno è in esecuzione. Un sistema che adotta timesharing invece è una variante di un sistema

multiprogrammato nel quale il/i processori eseguono più processi commutando le loro esecuzioni con una frequenza elevata da per permettere a ciascun utente di interagire con il proprio programma durante l'esecuzione.