Sistemi Operativi

A che servono i sistemi operativi

• **Cos'è?**

Definizione 1: Il sistema operativo è il primo programma che viene eseguito quando viene acceso il computer. Serve a fornirci un ambiente a finestre. Ci permette di installare ed eseguire le applicazioni che ci danno le funzionalità che ci interessano.

Definizione 2: È un insieme di programmi (software) che gestiscono gli elementi fisici di un computer (hardware). Gestisce l'HW del computer al posto dei servizi del sistema.

• **A cosa serve**

1. Fornisce una piattaforma di sviluppo per le applicazioni, che permette loro di condividere ed astrarre le risorse hardware (usare le risorse hardware in un modo più semplice che non con la programmazione a basso livello che ci offre il produttore dei dispositivi per gli sviluppatori di applicazioni).
2. Per gli utenti, invece, agisce a intermediario tra utenti e computer, permettendo agli utenti di controllare l'esecuzione dei programmi applicativi e l'assegnazione delle risorse hardware ad essi.
3. Protegge le risorse degli utenti (e dei loro programmi) dagli altri utenti (e dai loro programmi) e da eventuali fattori esterni.

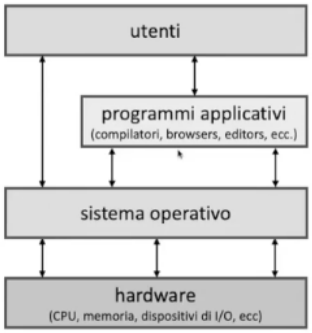
   Esempio: Se un programma non deve avere accesso ai dati sensibili di una certa regione del disco, tolgo quel permesso.

Definizione 3: Un sistema operativo è una piattaforma di sviluppo, ossia un insieme di funzionalità software che i programmi applicativi possono usare.
Ci sono API che le applicazioni possono invocare per fare diverse cose (Scrittura, lettura di fare, dare permessi, togliere permessi etc.).

- Quindi il sistema operativo **astrae le risorse hardware**, presentando agli sviluppatori dei programmi applicativi una visione delle risorse hardware più facile da usare e più potente rispetto alle risorse hardware "native" (basso livello).
- Inoltre il sistema operativo **condivide le risorse hw tra molti programmi contemporaneamente in esecuzione**, suddividendole tra i programmi in maniera equa ed efficiente e controllando che questi le usino correttamente.

Se seguissimo il modello della macchina di Von Neumann, avremmo un computer che esegue un processo alla volta e sarebbe uno spreco di risorse. Il tempo del computer in cui rimane fermo risulta essere molto di più di quello in cui viene utilizzato. L'obiettivo con l'informatica, infatti, è cercare di eliminare il più possibile i tempi morti, per questo nascono i computer potenti.

COMPONENTI DI UN SISTEMA DI ELABORAZIONE



- **Utenti**: persone, macchine, altri computer…
- **Programmi applicativi**: risolvono i problemi di calcolo degli utenti;
- **Sistema operativo**: coordina e controlla l'uso delle risorse hardware;
- **Hardware**: risorse di calcolo (CPU, periferiche, memorie di massa…)

Requisiti per i sistemi operativi

Vi sono molteplici tipologie dI computer e vengono usati in scenari applicativi molto diversi. In ogni scenario viene usato un computer che richiede che il sistema operativo che vi viene installato abbia caratteristiche ben determinate.

Esempio: un sistema embedded che serve per controllare le ali di un aereo non ha necessariamente bisogno di certe caratteristiche che ha un sistema operativo del laptop.

Esempi di scenari applicativi

- ***Server, mainframe***: massimizzazione la performance, rendere equa la condivisione delle risorse tra molti utenti. Gestiscono tante richieste, connessioni egualmente importanti.

(Esempio: La connessione deve essere equa per tutti, non può essere potente solo per una parte degli utenti e andare male per i restanti utenti).

- ***Laptop, PC, tablet***: massimizzare la facilità d'uso e la produttività della singola persona che lo usa.

(A differenza del server, non ho milioni di connessioni, ma viene utilizzato da una singola persona).

- ***Dispositivi mobili***: oltre alla facilità d'uso, è importantissimo ottimizzare i consumi energetici e la connettività.
- ***Sistemi embedded***: funzionare senza, o con minimo, intervento umano e reagire in tempo reale agli stimoli esterni (interrupt).

  Esempio: il computer non interagisce con l'umano, ma con l'ambiente tramite i sensori e deve funzionare con un minimo controllo umano per il maggior tempo possibile.

LA MALEDIZIONE DELLA GENERALITÀ

Definizione: La maledizione della generalità afferma che, se un sistema operativo deve supportare un insieme di scenari applicativi applicativi troppo ampio, non sarà in grado di supportare nessuno di tali scenari particolarmente bene.

Esempio: un sistema operativo che funziona sia per server che per laptop, pc, tablet etc. non funzionerà mai particolarmente bene.

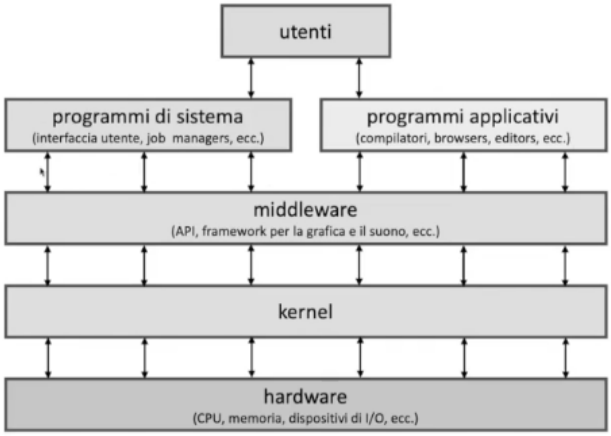
C'è un modo per superare la maledizione della generalità?

Mantenendo delle linee di codice diverse per quando si devono effettuare operazioni che sono per scenari applicativi diversi. C'è comunque questo rischio.

Esempio: Se il SO deve effettuare delle operazioni per un certo scenario applicativo, rimuovo delle parti necessarie per un altro scenario applicativo per poi aggiungerli nuovamente in caso si debbano effettuare operazioni con questi. È un metodo che può funzionare.

Struttura e servizi dei sistemi operativi

Non c'è una definizione universale dei programmi che fanno parte di un sistema operativo.



In generale un sistema operativo comprende almeno:

I. **Kernel**: programma sempre presente, si impadronisce del hw, lo gestisce ed offre ai programmi i servizi per poterlo usare in maniera condivisa ed astratta. È il sistema operativo vero e proprio.

II. **Middleware**: servizi di alto livello che astraggono ulteriormente i servizi del kernel. Di solito il kernel offre dei servizi chiamati "**chiamate di sistema**" che sono un po' più astratti dal hw ma ancora hanno delle caratteristiche che non li rendono ancora usabili dalle applicazioni. Di solito viene messo un ulteriore strato, il middleware, che rendono ancora più astratti i servizi del kernel, standardizzandoli perchè il kernel le offre fuori standard, quindi il middleware li deve standardizzare.
   Esempio: Standard POSIX.

III. **Programmi di sistema**: non sempre in esecuzione e offrono ulteriori funzionalità di supporto e di interazione utente con il sistema
   Esempio: Shell dei comandi.

Alcuni sistemi operativi forniscono anche dei programmi applicativi (editor, word processor, fogli di calcolo…), ma non li considereremo parti del sistema operativo stesso.

SERVIZI OFFERTI DA UN SISTEMA OPERATIVO

Un sistema operativo offre un certo numero di servizi:

- Per i **programmi applicativi**: perchè possano eseguire sul sistema di elaborazione usando le risorse astratte esposte dal sistema operativo;
- Per gli **utenti**: per gestire l'esecuzione dei programmi e stabilire a quali risorse hw i programmi (e gli altri utenti) hanno diritto.
- Per garantire che il sistema di elaborazione funzioni in maniera efficiente.

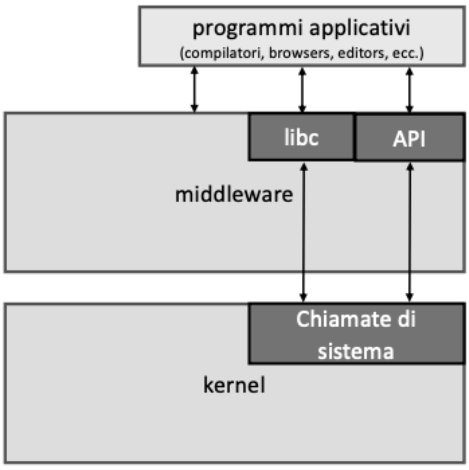
Gli utenti però interagiscono con il sistema operativo attraverso i programmi di sistema i quali utilizzano gli stessi servizi dei programmi applicativi. Quindi, in definitiva, il sistema operativo ha bisogno di esporre i suoi servizi esclusivamente ai programmi (applicativi o di sistema).

PRINCIPALI SERVIZI

I. **Controllo processi**: questi servizi permettono di caricare in memoria un programma, eseguirlo, identificare la sua terminazione e registrarne la condizione di terminazione (normale o erronea)
II. **Gestione file**: questi servizi permettono di leggere, scrivere, e manipolare files e directory
III. **Gestione dispositivi**: questi servizi permettono ai programmi di effettuare operazioni di input/output, ad esempio leggere da/scrivere su un terminale
IV. **Comunicazione tra processi**: i programmi in esecuzione possono collaborare tra di loro scambiandosi informazioni: questi servizi permettono ai programmi in esecuzione di comunicare (Esempio: multithreading).
V. **Protezione e sicurezza**: permette ai proprietari delle informazioni in un sistema multiutente o in rete di controllarne l'uso da parte di altri utenti e di difendere il sistema dagli accessi illegali
VI. **Allocazione delle risorse**: alloca le risorse hardware (CPU, memoria, dispositivi di I/O) ai programmi in esecuzione in maniera equa ed efficiente
VII. **Rilevamento errori**: gli errori possono avvenire nell'hardware o nel software (es. divisione per zero); quando avvengono il sistema operativo deve intraprendere opportune azioni (recupero, terminazione del programma o segnalazione della condizione di errore al programma).
VIII. **Logging**: mantiene traccia di quali programmi usano quali risorse, allo scopo di contabilizzarle

Chiamate di sistema ed API

Il kernel offre tutto un insieme di servizi chiamate "**chiamate di sistema**", ossia di funzioni invocabili in un determinato linguaggio di programmazione (C, C++).



Di solito i programmi non tendono a invocare direttamente il kernel (chiamare di sistema), ma a frapporre il middleware che mette a disposizione delle librerie chiamate **API (Application Program Interface)** implementate invocando le chiamate di sistema.

Perchè c'è questa distinzione?

1. Ogni linguaggio di programmazione ha delle librerie standard. Il problema è che spesso le API sono fortemente legate con le librerie standard al punto che anche queste diventano una parte implicita dell'API. Si cerca quindi di separare i due che utilizzano entrambi le chiamate di sistema.
2. Nell'ingegneria del software dove voglio un'interfaccia pubblica dei programmi applicativi standard e semplici da usare, ma una implementazione privata che posso modificare come mi pare (kernel di Windows è totalmente riservato, le chiamate di sistema sono riservate solo agli sviluppatori di Windows).

DIFFERENZA TRA CHIAMATE DI SISTEMA E API

- Le API sono esposte dal middleware, le chiamate di sistema dal kernel;
- Le API usano le chiamate di sistema nella loro implementazione;
- Le API sono standardizzate (esempio: standard POSIX e Win32), le chiamate di sistema no (ogni kernel ha le sue);
- Le API sono stabili, le chiamate di sistema possono variare al variare della versione del sistema operativo;
- Le API offrono funzionalità più ad alto livello e più semplici da usare, le chiamate di sistema offrono funzionalità più elementari e più complesse da usare.

I programmi di sistema

La maggior parte degli utenti utilizza i servizi del sistema operativo attraverso i programmi di sistema. Questi permettono agli utenti di avere un ambiente più conveniente per l'esecuzione dei programmi, il loro sviluppo, e la gestione delle risorse del sistema.

- **Interfaccia utente (UI**): permette agli utenti di interagire con il sistema stesso; può essere grafica (GUI) o a riga di comando (CLI); i sistemi mobili hanno un'interfaccia touch;
  Esempio:
  - **interprete dei comandi** permette agli utenti di impartire in maniera testuale delle istruzioni al sistema operativo. È possibile configurare quale interprete dei comandi usare, in questo caso è chiamato Shell. Ci sono due modi per implementare un comando:
    - Built-in: l'interprete esegue direttamente il comando, tipico nell'interprete di comandi di Windows;
    - Come programma di sistema: l'interprete esegue il programma, tipico delle Shell Unix e Unix-Like.

    Spesso riconosce un vero e proprio linguaggio di programmazione con variabili, condizionali, cicli etc.

  - **Interfacce grafiche (GUI):** l'interfaccia grafica è di solito basata sulla metafora della scrivania, delle icone e delle cartelle (corrispondenti alle directory).

  - **Interfacce touch-screen:** i dispositivi mobili richiedono interfacce di nuovo tipo senza dispositivo di puntamento, utilizzo delle gesture, tastiere vocali e comandi vocali.

- **Gestione file**: creazione, modifica, e cancellazione file e directory;

- **Modifica dei file**: editor di testo, programmi per la manipolazione del contenuto dei file;

- **Visualizzazione e modifica informazioni di stato**: data, ora, memoria disponibile, processi, utenti etc. fino informazioni complesse su prestazioni, accessi al sistema e debug. Alcuni sistemi implementano un **registry**, ossia un database delle informazioni di configurazione;

- **Caricamento ed esecuzione dei programmi**: loader assoluti e rilocabili, linker, debugger;

- **Ambienti di supporto alla programmazione**: compilatori, assemblatori, debugger, interpreti per diversi linguaggi di programmazione;

- **Comunicazione**: forniscono i meccanismi per creare connessioni tra utenti, programmi e sistemi; permettono di inviare messaggi agli schermi di un altro utente, di navigare il web, di inviare messaggi di posta elettronica, di accedere remotamente ad un altro computer, di trasferire file etc.

- **Servizi in background**: lanciati all'avvio, alcuni terminano, altri continuano l'esecuzione fino allo shutdown. Forniscono servizi quali verifica dello stato dei dischi, scheduling di jobs, logging etc.

IMPLEMENTAZIONE DEI PROGRAMMI DI SISTEMA

Sono implementati utilizzando le API, come i programmi applicativi.

Una possibile struttura del codice è riportata sulla destra, dove le invocazioni delle API sono riportate in grassetto.

- **Apri** in.txt in lettura
- Se non esiste
  - **Scrivi** un messaggio di errore su termimale
  - **Termina** il programma con codice errore
- **Apri** out.txt in scrittura
- Se non esiste, **crea** out.txt
- Loop
  - **Leggi** da in.txt
  - **Scrivi** su out.txt
- End loop
- **Chiudi** in.txt
- **Chiudi** out.txt
- **Termina** normalmente

Processi e thread

Concetto di processo

Un sistema operativo esegue un certo numero di programmi sullo stesso sistema di elaborazione e il numero di programmi da eseguire può essere arbitrariamente elevato, di solito molto maggiore del numero di CPU del sistema.

A tale scopo il sistema operativo realizza e mette a disposizione un'astrazione detta **processo**.

Definizione di processo: Un **processo** è un'entità attiva astratta definita dal sistema operativo allo scopo di eseguire un programma.

Per il momento supporremo che l'esecuzione di un processo sia sequenziale.

DIFFERENZA TRA PROGRAMMA E PROCESSO

PROGRAMMA	PROCESSO
Un'entità passiva (un insieme di istruzioni, tipicamente contenuto in un file sorgente o eseguibile)	Un'entità attiva (un esecutore di un programma o un programma in esecuzione)
Un programma può dare origine a diversi processi	Attivazioni di uno stesso programma all'interno della memoria e su i processori virtuali

STRUTTURA DI UN PROCESSO

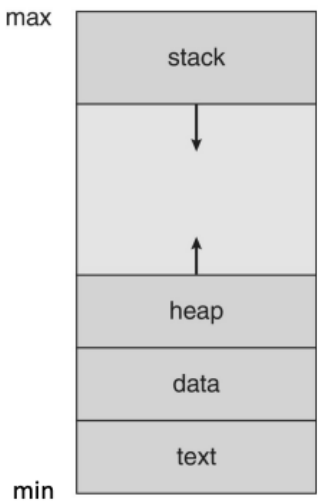
Un processo è composto da diverse parti:

- **Stato dei registri del processore** che esegue il programma, incluso il program counter (l'istruzione corrente che il mio processore sta eseguendo del programma);
- **Stato dell'immagine del processo** (Il programma in esecuzione userà una regione di memoria centrale. L'immagine del processo infatti è la regione di memoria centrale usata dal programma);
  Osservazione: Due processi dello stesso programma utilizzeranno la stessa regione di memoria? Ci sono delle tecniche che fanno si che due processi dello stesso programma non vadano a collidere e utilizzino delle zone di memoria centrale separate. (Da non confondere con i thread che invece utilizzano la stessa regione di memoria);
- **Risorse del sistema operativo in uso al programma** (possono essere condivise, esempio i file);
- **Informazioni di contabilità** (informazioni sullo stato del processo per il sistema operativo).

Due processi distinti hanno immagini distinte! Due processi operano su zone di memoria centrale separate!
Sono scatole chiuse dove ogni processo non c'è interferenza tra un processo all'altro. Ogni processo, pur essendo dello stesso programma, non devono interferire tra di loro in memoria

perchè voglio che i miei programmi eseguano come se ognuno avesse un proprio processore e una propria memoria.
Le risorse del sistema operativo, invece, possono essere condivise tra processi (a seconda del tipo di risorsa).

IMMAGINE DI UN PROCESSO



L'immagine di un processo è compresa nell'intervallo di indirizzi di memoria minima e massima chiamato **spazio di indirizzamento (Address space)** del processo. (Ogni processo lavora su intervalli di indirizzi diversi, ma ci sono casi in cui può succedere che più processi lavorino su indirizzi uguali -> spazio di indirizzamento virtuale. Lo affronteremo più avanti).

L'immagine di un processo di norma contiene:

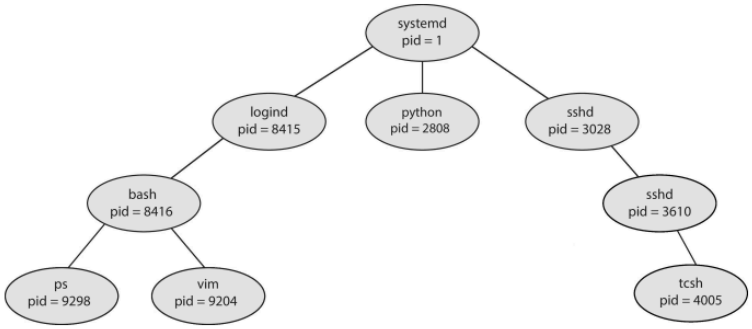
1. **Text section** dove si trova il programma (codice macchina del programma);
2. **Data section** dove si trovano le costanti e le variabili globali;
3. **Stack delle chiamate** che serve per le invocazioni di procedure e chiamate di funzioni (può crescere e diminuire). Contiene parametri, variabili locali e indirizzo di ritorno delle varie procedure che vengono invocate durante l'esecuzione del programma;
4. **Heap** contenente la memoria allocata dinamicamente durante l'esecuzione (può crescere e diminuire).

Operazioni sui processi

I SO forniscono delle chiamate di sistema con le quali un processo può creare/terminare/ manipolare altri processi.

Se un processo può creare un altro processo, alloca c'è bisogno di un processo primordiale dai quali tutti i processi utente e di sistema vengono progressivamente creati.
Infatti i processi sono organizzati in maniera gerarchica:

- Processo padre che può creare altri processi figli.
- Questi a loro volta possono essere padri di altri processi figli, creando un **albero di processi**.



CREAZIONE DI PROCESSI - RELAZIONE PADRE FIGLIO
La relazione padre/figlio è di norma importante per le politiche di condivisione risorse e di coordinazione tra processi.

I. **Politiche di *condivisione di risorse***:
   A. Padre e figlio condividono *tutte* le risorse;
   B. Padre e figlio condividono un *opportuno sottoinsieme* di risorse;
   C. Padre e figlio *non condividono* risorse

II. **Politiche di creazione *spazio di indirizzi*:**
   A. Il figlio è un *duplicato del padre* (stessa memoria e programma)
   B. Il figlio *non è un duplicato del padre* e bisogna specificare quale programma deve eseguire

III. Politiche di *coordinazione padre/figli*:
   A. Il padre è sospeso finché i figli non terminano
   B. Padre e figlio eseguono in maniera concorrenti

TERMINAZIONE DI PROCESSI
I processi di regola richiedono esplicitamente la propria terminazione al sistema operativo.

- Un processo padre può attendere o meno la terminazione di un figlio;
- Un processo padre può forzare la terminazione di un figlio (non è una buona cosa, il figlio termina in modo asincrono. Non permette di terminare il processo in modo ordinato).
  Possibili ragioni:
  - Il figlio sta usando risorse in eccesso (tempo, memoria…);
  - Le funzionalità del figlio non sono più richieste (ma è meglio terminarlo in maniera «ordinata» tramite IPC)
- Il padre termina prima che il figlio termini (in alcuni sistemi operativi)

Riguardo all'ultimo punto, alcuni sistemi operativi non permettono ai processi figli di esistere dopo la terminazione del padre.

Terminazione in cascata: anche i nipoti, pronipoti… devono essere terminati. La terminazione viene iniziata dal sistema operativo.

Definizione di processo zombie: se un processo termina ma il suo padre non lo sta aspettando (non ha invocato wait()), il processo è detto essere **zombie**. Le sue risorse non possono essere completamente deallocate (il padre potrebbe prima o poi invocare wait()).
Definizione di processo orfano: un **processo orfano** è uno dei figli ancora attivi di un processo padre che termina prima di loro e quindi non vi è una terminazione a cascata.

Comunicazione interprocesso

Un processo potrebbe cooperare con uno o altri processi se il suo comportamento influenza o è influenzato dal comportamento di questi ultimi.

Motivazioni:

- Condivisione informazioni;
- Accelerazione computazioni;
- Modularità ed isolamento.

Per permettere ai processi di cooperare il sistema operativo mette a disposizioni delle API per la comunicazione interprocessi (IPC).

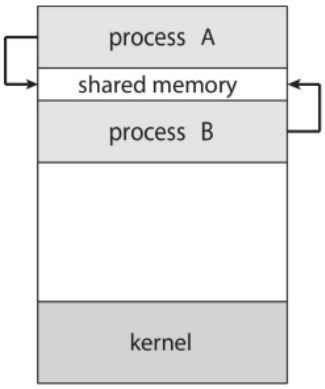
MODELLI TIPI DI PRIMITIVE PER IPC

Memoria condivisa	Message passing
Le immagini di due processi condividono una certa regione di memoria.	All'interno della memoria del sistema operativo viene messa una coda di messaggi nella quale un processo può inserire messaggi e un altro processo può estrarre i messaggi dalla coda.

MEMORIA CONDIVISA

Nella memoria condivisa viene stabilita una zona di memoria condivisa tra i processi che intendono comunicare.

- **La comunicazione è controllata dai processi che comunicano** (i processi si devono coordinare e un processo deve decidere quando scrivere e l'altro quando leggere. Non possono scrivere entrambi nella stessa cella di memoria), **non dal sistema operativo**. Serve un protocollo di coordinazione ed è totalmente a carico del programmatore applicativo.



Memoria condivisa

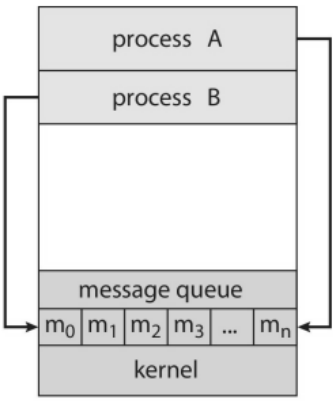
Un problema importante è **permettere ai processi che comunicano tramite memoria condivisa di sincronizzarsi** (un processo non deve leggere la memoria condivisa mentre l'altro la sta scrivendo). Allo scopo i sistemi operativi mettono a disposizione ulteriori primitive per la sincronizzazione

MESSAGE PASSING

Permettono ai processi *sia* di comunicare *che* di sincronizzarsi (stabilire chi legge e chi scrive). L'API vengono utilizzati anche per la sincronizzazione.
I processi comunicano tra di loro senza condividere memoria, attraverso la mediazione del sistema operativo.

Il SO mette a disposizione:

- Un'operazione `send(message)` con la quale un processo può inviare un messaggio ad un altro processo;
- Un'operazione `receive(message)` con la quale un processo può (mettersi in attesa fino a) ricevere un messaggio da un altro processo.



Message passing

Per comunicare due processi devono:

- Stabilire un **link di comunicazione** tra di loro (Aprire un canale di comunicazione tra i due processi);
- Scambiarsi messaggi usando `send(message)` e `receive(message)`.

PIPE

Sono canali di comunicazione tra i processi (una forma variante di message passing).

Varianti:

- Unidirezionale (processo manda messaggi in una direzione) o bidirezionale
- (se bidirezionale) Half-duplex o full-duplex (solo un processo parla oppure entrambi)
- Relazione tra i processi comunicanti (sono padre-figlio o no)
- Usabili o meno in rete

Definizione di pipe convenzionali: le **pipe convenzionali** sono unidirezionali, non accessibili al di fuori del processo creatore e quindi di solito condivise con un processo figlio attraverso una `fork()` (In Windows sono chiamate «pipe anonime»).

Definizione di named pipes: le **named pipes** sono bidirezionali, esistono anche dopo la terminazione del processo che le ha create e non richiedono una relazione padre-figlio tra i processi che le usano

- In Unix sono Half-duplex, solo sulla stessa macchina, solo dati byte-oriented.
- In Windows: sono Full-duplex e anche tra macchine diverse e anche dati message-oriented.

NOTIFICHE CON CALLBACK

In alcuni sistemi operativi (es. API POSIX e Win32) un processo può **notificare un altro processo in maniera da causare l'esecuzione di un blocco di codice** («callback»), similmente ad un interrupt (notifica che causano l'interruzione di quello che un processo sta facendo e causano l'esecuzione di un codice chiamato callback).

- **UNIX-LIKE:** Nei sistemi Unix-like (POSIX, Linux) tale notifiche vengono dette **segnali**, ed interrompono in maniera *asincrona* la computazione del processo corrente

causando un salto brusco alla callback di gestione, al termine della quale la computazione ritorna al punto di interruzione.

- **WINDOWS**: Nelle API Win32 esiste un meccanismo simile, detto **Asynchronous Procedure Call** (APC), che però richiede che il ricevente si metta esplicitamente in uno stato di attesa, e che esponga un servizio che il mittente possa invocare.

Multithreading

Se supponiamo che un processo possa avere molti processi virtuali (condividono lo stesso spazio di memoria). Più istruzioni quindi possono eseguire concorrentemente, e quindi il processo può avere più percorsi (thread) di esecuzione concorrenti.

Definizione di multithreading: il **multithreading** è un concetto nella programmazione e nell'informatica che si riferisce alla capacità di un sistema o di un'applicazione di eseguire più thread di esecuzione contemporaneamente.

Definizione di thread: Un **thread** è una sequenza di istruzioni che rappresenta l'unità di base di esecuzione di un programma.

In un ambiente multithreading, un'applicazione può suddividere le sue operazioni in più thread, consentendo loro di essere eseguiti in parallelo, o in modo concorrente, su una CPU o su più CPU se disponibili. Il multithreading è ampiamente utilizzato per migliorare le prestazioni e l'efficienza delle applicazioni in quanto consente di sfruttare appieno le risorse hardware disponibili.

PROCESSI SINGLE	MULTITHREADED
Ogni thread di uno stesso processo deve avere un proprio: 1. Stack: le chiamate a subroutine di un thread non devono interferire con quelle di un altro thread concorrente; 2. Processore (Program Counter distinto); 3. Registri del suo processore.	I thread di uno stesso processo condividono tra loro: 1. La memoria globale (data); 2. La memoria contenente il codice (code): eseguono lo stesso codice; 3. Risorse ottenute dal sistema operativo (Esempio: File aperti) 4. Heap

(Paragrafo discorsivo: In un processo multithreaded hanno parte della memoria condivisa e un proprio processore virtuale (un proprio PC distinto) e ha i suoi registri del suo processore distinti. Per quanto riguarda il codice (text) è condivisa tra tutti i thread perchè eseguono lo stesso codice. Stessa cosa vale per i data section e risorse del sistema operativo (Esempio: File). Anche l'heap è condiviso. Per quanto riguarda lo Stack, ogni

thread ha un proprio Stack delle chiamate. Se ogni thread chiamata delle subroutine, se non avessero Stack divisi interferirebbero tra chiamate).

Come si sfruttano i thread? Tramite le librerie di thread.

Definizione di librerie di thread: le **librerie di thread** sono API fornite al programmatore per creare e gestire thread.

(Librerie più in uso: POSIX thread e Windows thread).

GESTIONE DEI SEGNALI

Quando un processo è single-threaded, un segnale interrompe l'unico thread del processo.
Quando ci sono più thread, bisogna capire quale thread deve ricevere un segnale. Ci sono vari modi:

- Il thread a cui si applica il segnale viene interrotto e riceve il segnale;
- Ogni thread del processo;
- Alcuni thread del processo;
- Un thread speciale del processo deputato esclusivamente alla ricezione dei segnali.

CANCELLAZIONE DEI THREAD

L'operazione di cancellazione di un thread determina la terminazione prematura del thread. Può essere invocata da un altro thread.

Ci sono due tipi di approcci:

- **Cancellazione asincrona**: il thread che riceve la cancellazione viene terminato immediatamente.
  Vantaggio: Dal momento che un thread controlla il momento della propria cancellazione, può effettuare una terminazione ordinata.

- **Cancellazione differita**: un thread che supporta la cancellazione differita deve controllare periodicamente se esiste una richiesta di cancellazione pendente, e in tal caso terminare la propria esecuzione.
  Vantaggio: Nessuna necessità di controllare periodicamente se ci sono richieste di cancellazione pendenti.

Si può attivare o disattivare la cancellazione ed avere sia cancellazione differita che asincrona. Se la cancellazione è inattiva, le richieste di cancellazione rimangono in attesa fino a quando non viene nuovamente riattivata. In caso di cancellazione differita, questa avviene solo quando l'esecuzione del thread raggiunge un punto di cancellazione. Il

thread può aggiungere un punto di cancellazione controllando l'esistenza di richieste di cancellazione con la funzione `pthread_testcancel()`.

DATI LOCALI DEI THREAD

In alcuni casi è utile assegnare ad un thread dei dati locali (**Thread Local Storage, TLS**) che non sono condivisi con gli altri thread dello stesso processo. La TLS è diverta dalle variabili locali, infatti viene creata nella data section, globale. È tipo memoria globale, ma unica per ciascun thread (Esempio: dati static del linguaggio C, ma unica per ciascun thread). Utile quando il programma non ha un controllo diretto sul momento di creazione dei thread.

Gestione della memoria

Il problema dell'allocazione della memoria

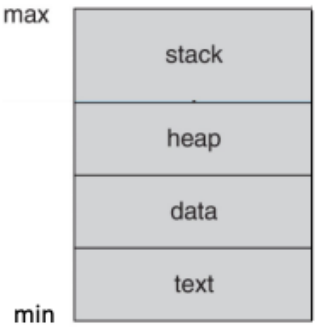
Perchè un programma possa andare in esecuzione, esso deve avere a disposizione:
- Il processore, per eseguire il codice
- La memoria centrale, per memorizzare il codice e i dati sul quale il codice opera.

Solo nei SO più semplici un solo programma alla volta è in memoria.
Nei SO moderni molti programmi sono contemporaneamente in memoria in uno stesso istante ("più immagini di più processi sono presenti contemporaneamente nella memoria centrale").

SO deve quindi allocare porzioni di memoria centrale ai diversi processi in funzione delle necessità di tali processi.

Spazio di indirizzamento

- Questo range minimo massimo spazio di spazio di indirizzamento del processo contiene gli indirizzi di memoria che il codice del programma utilizza.



Ogni processo ha a disposizione uno spazio di indirizzamento che può usare per le proprie operazioni.

Nei primi sistemi operativi tale spazio di indirizzamento era il range di memoria centrale che veniva assegnato al processo, era estremamente rigido. Se mi capitava che mi si aprisse dello spazio da un'altra parte, non potevo usarlo.

> *Esempio: se l'immagine di un certo processo avesse avuto dimensione 1 MB e fosse stata caricata in memoria centrale dall'inizio 001B:0000, il suo spazio di indirizzamento sarebbe stato 001B:0000 fino a 002B:0000.*

Oppure se due immagini hanno spazi di indirizzamento che si sovrappongono, non posso utilizzarli contemporaneamente. Non permette quindi di caricare lo stesso programma in zone diverse di memoria.

Associazione degli indirizzi

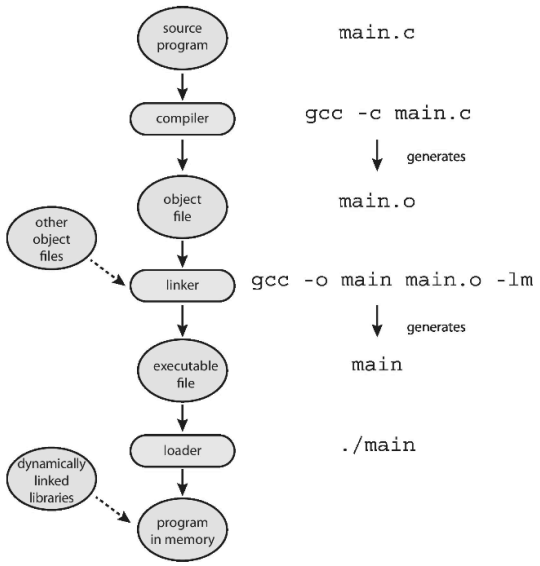
In presenza di molti programmi in memoria, il SO carica uno stesso programma in momenti diversi in diverse aree di memoria (dove trova spazio).

Ma come fa un istruzione del codice a fare riferimento ad una certa locazione di memoria senza conoscere a priori il suo indirizzo?

Una possibile soluzione è che il **codice del programma deve essere indipendente e questo range possa essere qualsiasi**. Il testo del programma deve essere configurato per adattarsi a qualsiasi indirizzo minimo, senza saperlo a priori.

1. **Position-independent code (PIC)**, prodotto dal compilatore. Tutti i processi hanno l'indirizzamento relativo (indirizzo che viene espresso a partire da una certa base solo in funzione di spiazzamento). Quindi se il codice utilizza indirizzi relativi, funziona purché in un certo registro venga caricato il valore della base.
   Formale: Il compilatore produce codice indipendente dalla posizione, ossia codice macchina che usi solo indirizzi di memoria relativi e che quindi funzioni correttamente in qualsiasi locazione di memoria venga caricato.
2. **Associazione degli indirizzi (binding)**, riscrittura del codice (produzione di codice dipendente dalla posizione) e tradurre gli indirizzi dipendenti dalla posizione negli indirizzi correttiLoader e Linker

Un programma sorgente è compilato in un file oggetto che deve poter essere caricato a partire da qualsiasi locazione di memoria fisica (file oggetto rilevabile).



- Il linker combinano più file oggetto (diversi file sorgente + librerie) per formare un file eseguibile.

- I loader si occupano di caricare in memoria i file eseguibili nel momento in cui devono essere eseguiti. Effettuano anche il linking delle librerie dinamiche.

(Le librerie dinamiche sono librerie linkate dal loader quando vengono caricare oppure direttamente dal programma quando vengono utilizzate).

Librerie dinamiche

Definizione di librerie dinamiche: Le librerie dinamiche sono componenti software autonomi che contengono funzioni e codice che possono essere utilizzati da più programmi o applicazioni. Queste librerie sono chiamate "dinamiche" perchè vengono caricate dinamicamente n memoria durante l'esecuzione del programma che le utilizza, invece di essere incorporate direttamente nel codice sorgente dell'applicazione.

Vantaggi:
1. Condivisione di codice: diverse applicazioni possono utilizzare la stessa libreria dinamica, risparmiando spazio su disco e memoria;
2. Aggiornamenti più facili: se è necessario apportare modifiche o correzioni a una libreria, è possibile farlo senza dover ricompilare tutte le applicazioni che la utilizzano. Gli aggiornamenti possono essere applicati in modo centralizzato;
3. Risparmio di memoria: poiché le librerie dinamiche vengono caricare in memoria solo quando necessario, possono contribuire a ridurre il consumo di memoria complessivo del sistema.

Quando un programma utilizza una libreria dinamica, il sistema operativo o un caricatore dinamico si occupa di caricare la libreria in memoria e risolvere i riferimenti alle funzioni in essa contenute in

fase di esecuzione. Questo permette una maggiore flessibilità e modularità nell'implementazione delle applicazioni software.
Sono librerie linkate dinamicamente sono un ulteriore complicatezza perchè il linking di alcune librerie API viene fatto durante l'esecuzione del programma (non tutto durante la compilazione).

Associazione (binding) degli indirizzi

Supponiamo che il loader debba fare il binding degli indirizzi:

1. **In compilazione**: il linker sa già l'indirizzo di memoria in cui verrà caricato il codice, generando il codice assoluto (Esempio kernel del SO, usa sempre lo stesso indirizzo di memoria senza essere ritoccato).

Svantaggio: Semplice, ma se cambia l'indirizzo di caricamento, il codice deve essere ricompilato.

2. **In caricamento**: il linker genera un eseguibile rilevabile (non position dependent ma con un insieme di tabelle che dicono ove stanno tutte le istruzioni macchina dentro il testo che contengono indirizzi di memoria. Dopo di che il loader consulta tutte delle tabelle e in fase di caricamento sostituisce nelle istruzioni macchina gli indirizzi di memoria scorretti con gli indirizzi di memoria corretti -> questo processo si chiama rilocazione. Questo nel momento in cui il programma viene caricato.

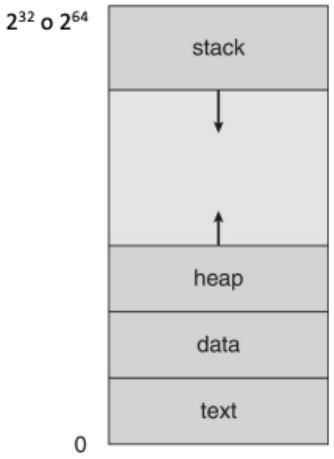
Svantaggio: Flessibile, ma soluzione lenta perchè rallenta molto il tempo di caricamento del codice. Il problema è il tempo di avvio ed è un tempo molto lungo quindi non è una soluzione così flessibile. Inoltre non permette di spostare l'immagine di un processo (rilocazione) durante la sua esecuzione. Un'altra cosa che fanno infatti i SO non è solo spostare l'immagine di un processo in memoria all'inizio dell'esecuzione e lasciarlo lì, ma durante l'esecuzione si ferma ogni tanto, si toglie l'immagine, si sposta in un'altra zona della memoria e si riprende dal punto dove si era interrotto (molto complicata). Fare ciò con il binding di caricamento sarebbe impossibile.

- Non permette di spostare l'immagine di un processo in un altro punto della memoria centrale durante l'esecuzione;
- Non permette di utilizzare le librerie dinamiche;
- Non permette di realizzare uno spazio di indirizzamento virtuale per i processi;
- Perché sia fattibile l'eseguibile deve essere rilocabile.
- È il più lento tra i tre.

3. **In esecuzione**: il binding viene effettuato direttamente dall'HW. Esiste un'unità HW dedicata al binding (Memory Management Unit). È la più flessibile di tutte: permette di spostare anche l'immagine del processo durante la sua esecuzione (come abbiamo spiegato prima). Questo tipo di binding richiede però un supporto HW.

Nei sistemi operativi moderni è usato il binding in esecuzione perchè tutte le CPU moderne hanno la Memory Management Unit e quelle che non lo hanno sono solo in CPU di sistemi embedded. Il binding in compilazione è usato per alcuni eseguibili speciali, come il kernel, di cui si sa a priori l'indirizzo di caricamento.

Spazio di Indirizzamento Virtuale

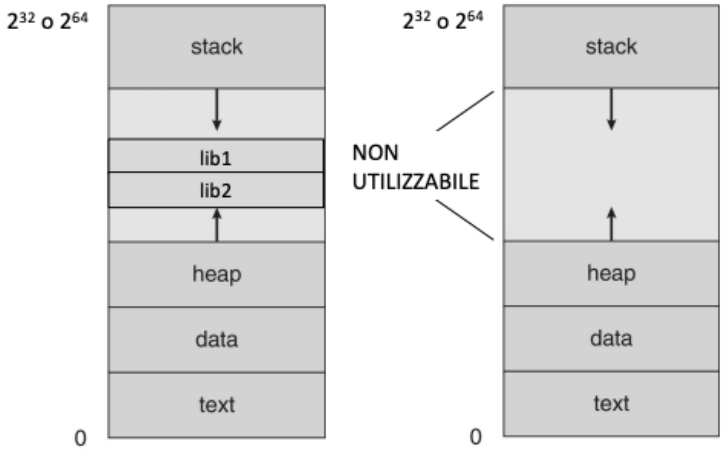


Nei SO ogni processo ha uno **spazio di indirizzamento virtuale** grazie a questi sistemi di rilocazione (binding), possono essere compilate assumendo all'interno del codice del programma che il programma possa utilizzare uno spazio di indirizzamento che ha un range di indirizzi da 0 a 2^32 (32 bit) oppure 2^64 (64 bit) (dipende dall'architetture del processore).

Questo spazio di indirizzamento in cui l'indirizzo minimo è sempre 0 per tutti gli spazi di indirizzamento di tutte le immagini di tutti i processi viene detto spazio di indirizzamento virtuale. Ogni processo ne ha uno.

Totalmente indipendente dagli indirizzi fisici in cui la sua immagine verrà caricata (l'immagine non deve essere caricata in memoria fisica per forza da 0 (esempio da indirizzo 1000), può essere caricato dove vuole. Inoltre non deve occupare per forza fino a 2^32 o 2^64 la sua immagine nella memoria fisica, sono tantissimi).

Esistono tecniche di associazione degli indirizzi in esecuzione che fanno corrispondere lo spazio di indirizzamento virtuale del processo con la regione (o le regioni) di memoria centrale che la sua immagine occupa. Permettono di eseguire un processo anche se solo una parte della sua immagine è in memoria. Quindi posso avere uno spazio di indirizzamento virtuale molto grande perchè tanto non devo caricare tutta l'immagine in memoria.



Esiste una parte del VAS che è inutilizzato (perchè è molto grande, anche quando è molto utilizzato è impossibile occupare tutta la memoria). Se lo utilizzassimo tutto non avemmo abbastanza memoria fisica per contenere l'immagine. Solitamente lo spazio inutilizzato si trova tra Stack e heap e non viene mappata su nessuna memoria fisica.

Di solito in questo spazio vengono caricate le librerie dinamiche (possono essere caricate in qualsiasi ordine e non hanno un indirizzamento virtuale, vengono caricate all'interno dello spazio di indirizzamento virtuale ma gli indirizzi che utilizzano sono direttamente indirizzi virtuali. Devono essere compilati come PIC, così possono capitare in qualsiasi posto dello spazio e non si sovrappongono).

MEMORY MAPPING
In generale i sistemi operativi mettono a disposizione API per mappare una regione inutilizzabile del VAS su memoria centrale, così diventi utilizzabile. Esistono delle API che permettono di mappare una regione del VAS che è libera (mappato su niente) e di mapparla o su della memoria

fisica oppure su un file e quindi si può implementare la memoria condivisa oppure i file mappati in memoria.
In tal modo l'accesso al file può avvenire utilizzando le istruzioni macchina per accedere alla memoria, anziché le API del filesystem.

INFORMAZIONI SULLE API PER LA GESTIONE DELLA MEMORIA
Di norma non dobbiamo usare le API per gestire Stack e Heap:
- Lo Stack è gestito autonomamente dal sistema operativo, non tramite API;
- Lo Heap è gestito di norma dal supporto runtime del linguaggio (new in C++) o dalla sua libreria (malloc in C), che invocano le API per ridurre/espandere lo heap in funzione delle necessità del processo.

Le API per la gestione della memoria ci permettono di:
- Avere regioni di memoria con permessi particolari (sola lettura, eseguibili etc.);
- Implementare componenti quali allocutori di memoria, compilatori just-in-time etc. qualora volessimo implementare il nostro nuovo linguaggio di programmazione;
- Utilizzare i file mappati in memoria e la memoria condivisa.

Guardare API POSIX per gestione della memoria.

File System

File e File System

- **Definizione di File System**: Il **File System** è il modo attraverso il quale SO memorizza in linea i dati e i programmi.
- **Definizione di File**: Il **File** è una unità di memorizzazione logica, un insieme di informazioni correlate, registrate in memoria secondaria, alle quali è stato dato un nome.

Attributi

- **Nome**: unica informazione umanamente leggibile
- **Identificatore**: utilizzato dal File System per distinguere i file.
- **Tipo**: tipo di dati contenuti nel file
- **Locazione**: identifica su quale dispositivo di memoria secondaria e posizione nel dispositivo dove l'informazione del file è memorizzata.
- Dimensione
- **Protezione**: informazione su chi può modificare/leggere file
- **Ora**, **data** e **utente** che ha creato, letto o modificato per ultimo il file
- **Attributi estesi**: verifica se il contenuto del file è corrotto o no

Le informazioni del file sono memorizzate in una **directory**.

Operazioni dei processi sui file

- **Creazione**: viene riservato spazio nel File System per i dati, e viene aggiunto un elemento nella directory.
- **Apertura**: annuncia all'SO che voglio effettuare operazioni su file.
- **Lettura**: lettura a partire dalla posizione determinata da un puntatore di lettura.
- **Scrittura**: scrittura a partire dalla posizione determinata da un puntatore di scrittura (che coincide con quello di lettura).
- **Riposizionamento** (**seek**): spostamento del puntatore all'interno del file.
- **Chiusura**: effettuata alla fine dell'utilizzo di un file.
- **Cancellazione** e **troncamento**: il troncamento cancella i dati ma non il file con i suoi attributi.

LOCK DEI FILE

Uno stesso file può essere aperto contemporaneamente da più processi che operano in concorrenza. Alcuni sistemi operativi permettono di associale ai file (o a porzioni di esso) dei lock per coordinare i processi che operano sullo stesso file.

Cosa succede se due processi operano sullo stesso file? Il problema è come se due processi lavorassero su una stessa zona di memoria. I SO mettono a disposizione delle API di coordinazione. Per i file vediamo quali sono le primitive di sincronizzazione (locking). In alcuni casi sono necessari, in altri no.
Di regola i SO permettono di associare dei lock, possibilità di bloccare porzioni o un file intero. I lock sono delle primitive molto primitive, sono dei modi per coordinarsi in cui è facilissimo sbagliare ad usarli. Il classico esempio di errore è il dead lock (problema che si verifica quando un

thread o processo vuole prendere due lock e un altro thread o processo vuole prendere gli stessi lock).

Ci sono nei sistemi operativi tipicamente due/quattro tipi di lock:
1. **Lock condiviso**: lock di lettura, permette ad altri processi anche loro il lock condiviso ma non esclusivo (Se uno ha il lock condiviso,, gli altri processi possono fare solo lock condiviso). Serve come lock di lettura perchè se voglio solo leggere è sicuro condividere il file con altri processi che vogliono solo leggere.
2. **Lock esclusivo**: lock di scrittura. Se un processo vuole scrivere, prende il lock escluso ed esclude tutti gli altri lock.
3. **Lock consultivi**: il SO non obbliga il lock, ma non regola l'accesso al file. Lo mette solo a disposizione dei thread e lascia che siano i thread o processi che devono evitare di accedere al file se non hanno il lock (Unix-Like).
4. **Lock obbligatori**: il SO impedisce l'accesso al file ai processi che non detengono il lock. Impedisce l'accesso concorrente (Windows).

TIPI DI FILE

Esistono tanti tipi di file. Il SO può essere più o meno consapevole del tipo di file, ma deve almeno riconoscere il tipo di file eseguibile (per creare l'immagine del processo, altrimenti non riesce ad eseguire i programmi).

Per riconoscere il tipo di file:
- Schema del nome (estensione);
- Attributi nel file (In MacOs viene registrato il programma che ha creato il file);
- Magic Number all'inizio del file (Ci sono standard).

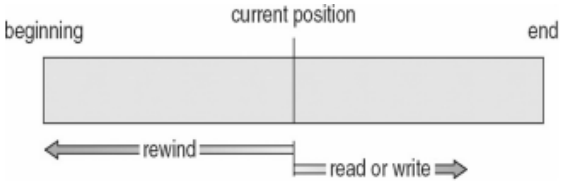
Possono essere:
- **Non strutturati** (nei sistemi Unix-like un file è una sequenza di byte);
- **Sequenza di record** (righe di testo o record binari, a struttura e lunghezza fissa o variabile).
- Strutture più **complesse e standardizzate**, sopratutto per file eseguibili.

Più il sistema operativo supporta direttamente diverse strutture di file, più diventa complesso. Inoltre se il sistema operativo è troppo «rigido» sulle possibili strutture, potrebbe non supportare nuovi tipi, o tipi ibridi,

METODI DI ACCESSO

Accesso sequenziale (il più semplice, praticamente non più usato):

1. Il file è sequenza di record a lunghezza fissa.
2. Può effettuare le operazioni di `read_next()` e `write_next()`, i quali leggono/scrivono il successivo record dalla posizione corrente, e l'operazione di riavvolgimento.
   (Essenzialmente puoi solo andare avanti e in caso ritorni all'inizio del file - Puntatore alla posizione corrente).



Accesso diretto:

1. Permette di muoversi nel file in qualsiasi punto;
2. Operazioni: read(n) e write (n) dove n è un offset (per accedere direttamente all'n-esimo record), read_next(), write_next() (leggo byte dopo) e position(n) (vado all'n-esimo record).

Accesso indicizzato:
1. Nei mainframe, i file possono essere sequenze di record ordinate secondo un determinato campo chiave del record. In tali sistemi l'accesso si basa sulla ricerca di una chiave e il sistema operativo mantiene un indice per velocizzare l'accesso.

Guardare API POSIX per operazioni su file.

Directory

Definizione di directory: una **directory** è, in pratica, un elenco di file presenti nel file System. È una struttura organizzativa utilizzata per archiviare e gestire file e altre directory all'interno di un sistema di file. Una directory può contenere una varietà di elementi, tra cui file, sottodirectory e collegamenti simbolici. Le directory sono utilizzate per organizzare i file in una struttura gerarchica, consentendo agli utenti di raggruppare e categorizzare i dati in modo logico.

(**Digressione sul volume**: In informatica, un volume è una porzione di uno spazio di archiviazione, come un disco rigido, un'unità a stato solido (SSD) o un altro dispositivo di archiviazione, che è trattato come un'entità separata e gestibile. I volumi consentono di organizzare e gestire i dati in modo più flessibile all'interno di un dispositivo di archiviazione. I volumi consentono agli utenti di organizzare e separare i dati in modo più efficace, rendendo più semplice la gestione delle informazioni su dispositivi di archiviazione complessi. Questi volumi possono anche essere utilizzati per scopi specifici, come la creazione di partizioni per il sistema operativo, i dati utente o i backup, o per la separazione dei dati in base alle categorie o alle esigenze specifiche dell'utente.)

Un volume, che è una zona di un disco o di un'unità di memorizzazione secondaria nella quale c'è dentro una zona di dati (file) e una struttura della directory. Il directory infatti è una parte del volume.
File System, invece, un pezzo di sistema operativo che gestire il file, un pezzo di software.

- Più formalmente: è una tabella che permette di associare il nome di un file ai dati (e metadati) contenuti nel file stesso.

- Sia i file che le directory risiedono su disco: deve esservi almeno una directory nel file system (altrimenti non sarebbe possibile ritrovare i file!).

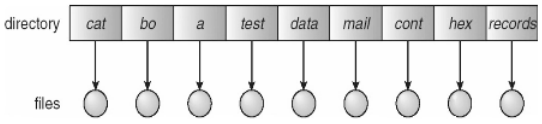
OPERAZIONI DEI PROCESSI SULLE DIRECTORY

- **Creazione** di un file in una directory;
- **Cancellazione** di un file in una directory;
- **Ridenominazione** di un file o **spostamento** da una directory ad un'altra;
- **Elenco** dei file in una directory;
- **Ricerca** di un file: basata sul nome, o su uno schema di possibili nomi;
- **Attraversamento** del file system, ad esempio per backup (Attraverso l'albero della directory).

STRUTTURA DELLE DIRECTORY

Ad un livello

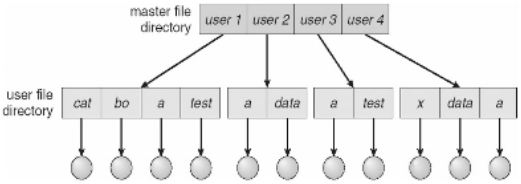
Una sola directory per tutti i file. Un unico grande enorme elenco di file. Ogni file deve avere un nome diverso.



Vantaggi: semplicità
Svantaggi:
• Difficoltà naming file quando sono molti (non si possono avere due file con lo stesso nome)
• Difficoltà raggruppamento file di utenti diversi (di solito sono in ordine alfabetico, quindi come si fa a riconoscere che sia di due utenti diversi?).

A due livelli

La directory principale contiene delle sottodirectory, una per ogni utente.
La sotto directory utente contiene i file dell'utente.
Utenti diversi possono dare lo stesso nome a file diversi.

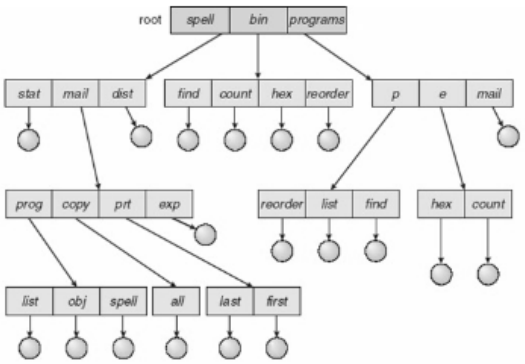


Occorre quindi usare dei nomi di percorso (path name) per identificare un file univocamente
• /user2/data(separatori Unix-like)
• \user2\data(separatori Windows-like)
• >user2>data (separatori MULTICS-like)

Un pathname è quindi (nomedirectory + separatore + nomeFile).
I file di sistema di solito sono posti in una o più directory speciali, e occorre che il sistema conosca un **percorso di ricerca** per trovarli.

Ad albero

Generalizzazione della struttura a due livelli. Ogni directory contiene ricorsivamente file se altre directory, formando un albero di directory.

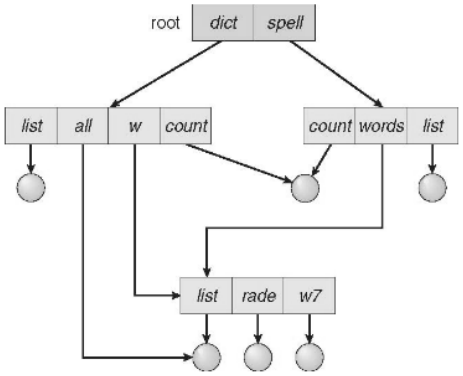


Permette agli utenti di raggruppare i propri file in sottodirectory. Viene più facile anche organizzare i file eseguibili.

Per semplificare l'accesso ad ogni programma è assegnata una directory corrente, dalla quale si possono specificare path relativi
Esempio: se la directory corrente è /programs/mail, un path relativo potrebbe essere prt/first

Cancellazione directory:
• Con la directory cancello tutto il suo contenuto;
• Oppure permetto di cancellare una directory solo se è vuota
• La seconda soluzione è più sicura

A grafo aciclico
Permette l'aliasing (più di un nome per lo stesso file)

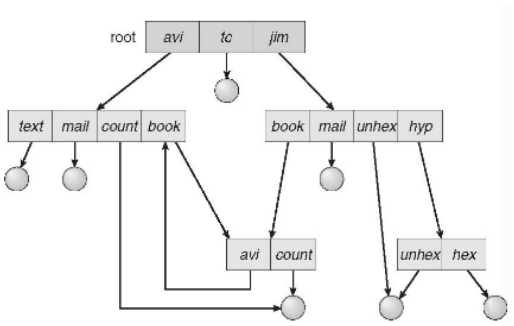


Cosa succede se cancello un file/directory con un alias?
• Hard links: duplicazione voci di directory; viene introdotto un contatore ai riferimenti, quando è a zero viene cancellato il file (Se elimino dict/count, spell/count non viene eliminato. Verrà liberato spazio in memoria solo quando entrambi sono eliminati);

• Link simbolici: riferimenti simbolici ad un path assoluto, quando questo è cancellato restano dangling; non sono aggiunti al contatore dei riferimenti (Collegamenti ad un file, non è una voce di directory, ma un file che contiene l'indirizzo dove si trova il vero file. Se il file originale viene cancellato, il collegamento rimane ma rimane pendente).

Problema attraversamento (Se io attraverso l'albero delle directory per elencare tuti i file, se ho degli hard link devo attraversali due volte o una volta sola? Non c'è una soluzione esatta).

A grafo generale
Possibilità di hard link anche a directory a livelli superiori, persino che contengono ricorsivamente il link stesso. (Contiene dei cicli, non è più un albero che è un grafo aciclico).
Per determinare se un file non è più referenziato un contatore al numero dei riferimenti non basta, occorre un autentico algoritmo di garbage collection.



Attraversamento del file system ancora più complicato

Guardare API POSIX per operazioni su directory.

Protezione

Tutti i sistemi operativi sono stati pensati per essere usati da più di una persona contemporaneamente (Unix) oppure non contemporaneamente (Windows). Le informazioni devono essere preservate dai danni fisici (affidabilità) e dagli accessi impropri (protezione).
Un sistema multiutente permette un accesso controllato ai file di un certo utente da parte degli altri utenti.

Operazioni controllabili (perchè un utente possa essere sicuro che gli altri utenti abbiano un accesso mediato da un SO ai file e quindi che ci sia protezione):
• **Lettura** (Controlla che non possa leggere)
• **Scrittura**
• **Esecuzione**
• **Aggiunta** (scrittura in coda ad un file)
• **Cancellazione**
• **Elencazione** (elenco contenuto directory)

LISTE DI CONTROLLO DEGLI ACCESSI

- Ad ogni file/directory è associata una **lista di controllo degli accessi** (access control list, ACL);
- L'ACL specifica gli utenti che possono accedere al file/directory, con i relativi permessi di accesso;
- Il file system controlla l'ACL prima di ogni accesso al file.

Principale svantaggio: le ACL possono diventare molto lunghe (Tanti file, tante directory, tanti utenti).

Approccio usato nei sistemi Unix-like: raggruppare gli utenti in classi distinte:
1. Proprietario: l'utente che possiede il file/directory;
2. Gruppo: il gruppo di utenti che condivide il file/directory;
3. Pubblico: tutti gli altri utenti.

Se un certo file ha come diritti di accesso 7 (proprietario può leggere, scrivere ed esecuzione), 6 (utenti del gruppo possono leggere e scrivere ma non eseguirlo) 4 (gli altri possono solo leggere il file). I numeri sono convertiti in binario.

- Esempio:

			r w x
• Accesso proprietario	7	⇒	1 1 1
• Accesso gruppo	6	⇒	1 1 0
• Accesso pubblico	4	⇒	1 0 0

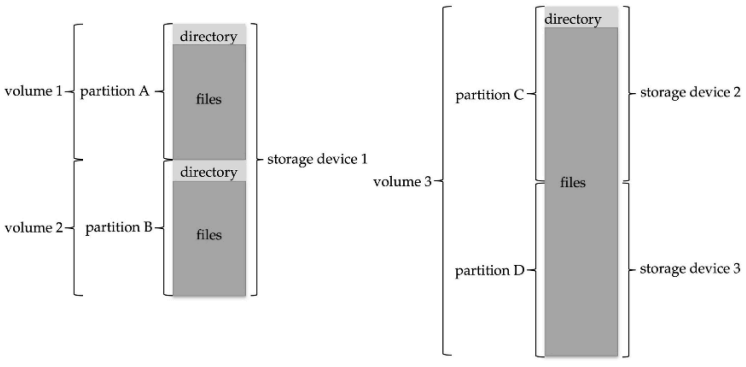
- Supponiamo di avere un file `game`, di volergli assegnare un gruppo G, e di volergli attribuire i permessi di cui sopra:
  - `chgrp G game   #cambia gruppo al file game`
  - `chmod 764 game #cambia i permessi al file game`

Guardare API POSIX per la protezione

Volumi e montaggio

VOLUMI

- Un sistema operativo deve permettere di aggiungere e rimuovere dinamicamente unità di memorizzazione dati
- Un dispositivo di archiviazione può essere suddiviso in **partizioni (più volumi);**

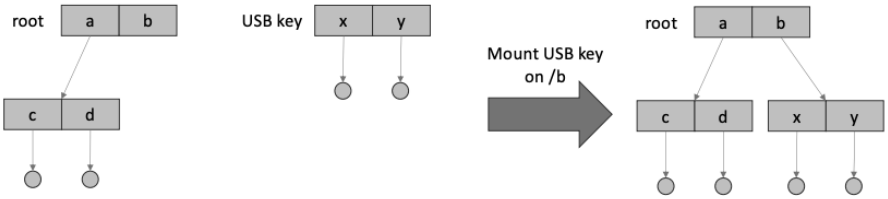


Definizione di volume: Un **volume** è una zona di un dispositivo di archiviazione contenente un file system. Un volume di solito è contenuto in una partizione, su un solo dispositivo. Ma per alcuni file system particolari un volume si può estendere su più partizioni/dispositivi

MONTAGGIO E SMONTAGGIO
Il sistema operativo deve permettere di **montare** e **smontare** un volume all'interno dello spazio dei nomi del filesystem.

Di solito per montare un volume occorre fornire al sistema operativo:

- L'identificazione del dispositivo/partizione dove risiede il volume;
- Il punto di montaggio, ossia la locazione nella struttura di file e directory alla quale «agganciare» il filesystem contenuto nel volume (tipicamente una directory vuota).



Supponiamo che sulla mia chiavetta usb ho un albero della directory con un paio di file senza directory e sul mio pc ho la directory a e b e sulla a ho i file c e d. Un'idea che posso avere è la mia directory b è completamente vuota e lo posso scegliere come punto di montaggio per la mia chiavetta usb. Gli dico "Monta la mia chiavetta usb sul mio punto di montaggio b". Alla fine ottengo è l'albero della directory a destra. Vedo tutto integrato in un unico albero della directory.

Varianti:

- Punto di montaggio: directory vuota oppure no
- Montaggio automatico o manuale
- Utilizzo identificatori (lettere) di unità anziché punti di montaggio
- Montaggio ripetuto di uno stesso volume permesso oppure no

Guardare le API POSIX per operazioni su file