Sistemi operativi

Introduzione

Un computer, appena avviato, inizia ad eseguire un **programma**; questo programma è il sistema operativo. I principali sono Windows/macOS/Linux, mentre I principali OS per piattaforme mobili sono Android e iOS.

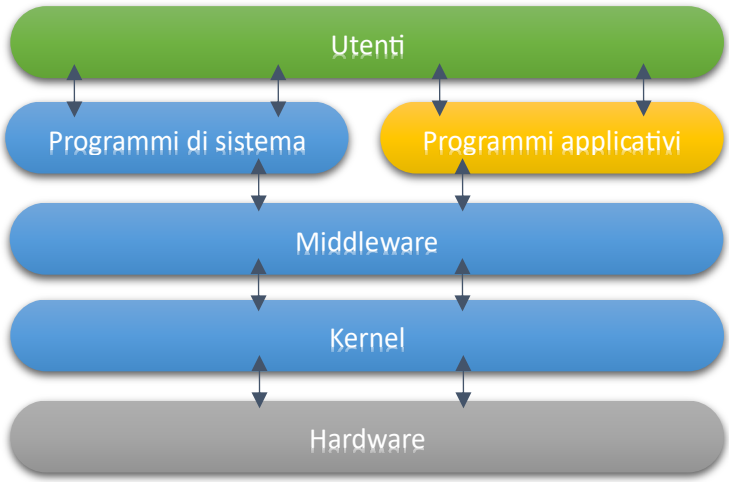
Il sistema operativo è un insieme di programmi che gestiscono gli elementi fisici di un computer; agisce da intermediario tra utenti e computer, permettendo agli utenti di controllare l'esecuzione dei programmi applicativi e l'assegnazione delle risorse hardware ad essi. Il sistema operativo **controlla di per sé inoltre l'hardware**: un esempio di questo è il fatto che, mentre il computer aspetta l'esecuzione di un programma, viene eseguito un altro programma perché così si può minimizzare il tempo in cui il computer non fa nulla, e massimizzare il tempo in cui il computer esegue qualcosa che è veramente utile.

Fornisce inoltre un **ambiente** nel quale noi possiamo fare delle operazioni per gestire le nostre **applicazioni**. Ci permette inoltre di eseguire tante applicazioni contemporaneamente, e mantiene ed organizza i nostri dati sotto forma di file e cartelle. Un altro aspetto importante è la sicurezza, perché appunto protegge le risorse degli utenti (e dei loro programmi) dagli altri utenti (e dai loro programmi), e da eventuali attori esterni.

Le basi

Un sistema operativo, in primo luogo, è una **piattaforma di sviluppo**, ossia un insieme di funzionalità software che i programmi applicativi possono usare.
Da un lato, il sistema operativo **astrae le risorse hardware**, presentando agli sviluppatori dei programmi applicativi (**API**) una visione delle risorse hardware più facile e più potente rispetto alle risorse hardware native. Per esempio, *il sistema operativo permette di scrivere e leggere file su disco senza dover conoscere i dettagli fisici del disco stesso*. In questo modo, gli sviluppatori possono concentrarsi sulla logica dei loro programmi applicativi senza dover gestire direttamente le risorse hardware.
Dall'altro lato, il sistema operativo **condivide le risorse hardware** tra molti programmi contemporaneamente in esecuzione, suddividendole tra i programmi in maniera equa ed efficiente, e controllando che questi le usino correttamente.

I componenti



- **Utenti**: persone, macchine o altri computer
- **Programmi applicativi**: usano le risorse hardware per svolgere operazioni di calcolo (*compilatori, browser, editor, etc)*
- **Sistema operativo**
  - **Kernel**: programma che rimane sempre in esecuzione, e gestisce le risorse hardware in modo da permettere ai programmi applicativi di usarle in maniera condivisa e astratta.
  - **Middleware**: programmi che offrono dei servizi di alto livello ai programmi applicativi. Il middleware astrae ulteriormente i servizi del kernel, e semplifica la programmazione. Per esempio, può fornire API o dei framework.
  - **Programmi di sistema**: non sono sempre in esecuzione, ma offrono delle funzionalità di supporto e di interazione utente con il sistema. Per esempio, gestiscono i jobs e i processi, oltre a permettere di interfacciarsi con il sistema operativo tramite una shell, configurare il sistema, etc.
- **Risorse hardware**: CPU, periferiche, memorie di massa, etc…

Servizi offerti dall'OS

Un sistema operativo offre un certo numero di servizi per i programmi applicativi (perché 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 hardware i programmi hanno diritto), e 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; in definitiva, **il sistema operativo ha bisogno di esporre i suoi servizi esclusivamente ai programmi**.

I principali servizi sono:

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

La generalità

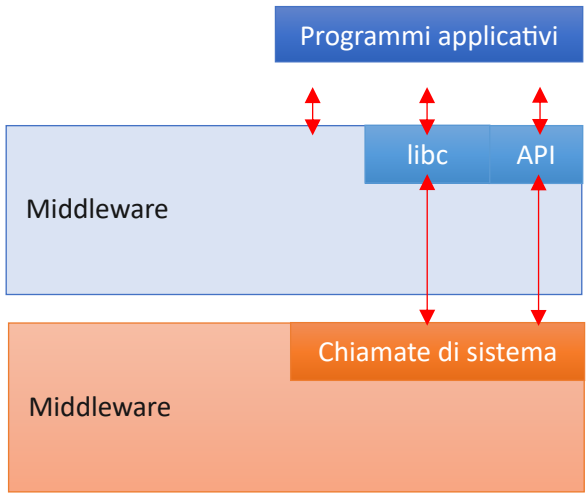
Oggigiorno, vi sono molteplici tipologie di computer utilizzati in scenari applicativi diversi. In quasi tutti i tipi di computer, si tende ad installare un sistema operativo allo scopo di gestire l'hardware e semplificare la programmazione. Ma ogni scenario applicativo in cui viene usato un computer richiede che **il sistema operativo che vi viene installato abbia caratteristiche ben determinate**, come ad esempio:

- **Server, mainframe**: è importante massimizzare la performance e rendere equa la condivisione delle risorse tra molti utenti.
- **Laptop, PC, tablet**: è importante massimizzare la facilità d'uso e la produttività della singola persona che lo usa.
- **Dispositivi mobili**: è importante ottimizzare i consumi energetici e la connettività.
- **Sistemi embedded**: è importante che funzionino senza, o con minimo intervento umano, e che reagiscono in tempo reale agli stimoli esterni (interrupt)

Se un sistema operativo deve supportare un insieme di scenari applicativi troppo ampio, non sarà in grado di supportare nessuno di tali scenari particolarmente bene; questo è chiamato **maledizione della generalità**.

Le API e le chiamate di sistema

Il kernel offre i propri servizi ai programmi come chiamate di sistema, ossia funzioni invocabili. I programmi, invece, usano le API implementate invocando le chiamate di sistema. Spesso le API sono fortemente legate con le librerie standard del linguaggio di implementazione, al punto che anche queste diventano una parte implicita dell'API.
Le API seguono degli standard di architettura, come win32 o POSIX.



Le differenze tra le due "strutture" sono le seguenti:

Topic	**API**	**Chiamate di sistema**
Esposizione	Dal middleware	Dal kernel
Usano…	Le chiamate di sistema nella loro implementazione	-
Standardizzazione	Si (POSIX/Win32)	No
Stabili?	Si, una volta che sono pubblicate, ci vogliono anni prima che vengano modificate (e quelle vecchie non lo sono)	Possono variare da versione a versione
Funzionalità	Alto livello e più semplici da usare	Più elementari e più complesse da usare

I programmi di sistema

Sono il modo con cui gli utenti interagiscono con il sistema operativo, per lanciare programmi, installare, ecc…. Abbiamo varie tipologie:

- **Gestione file**: creazione, modifica, cancellazione file e directory
- **Modifica dei file**: editor di testo, programmi per la manipolazione del contenuto del file

- **Visualizzazione e modifica informazioni di stato**: data, ora, memoria disponibile, processi, ecc…. E informazioni complesse su prestazioni, debug.
    - Alcuni sistemi implementano un **registro**, 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
- **Comunicazione**: forniscono i meccanismi per creare connessioni tra utenti, programmi e sistemi; permettono di inviare messaggi agli schermi di un altro utente, navigare il web, inviare messaggi ecc….
- **Servizi in background**: lanciati all'avvio, alcuni terminano e altri continuano l'esecuzione fino allo shutdown. Forniscono servizi quali la verifica dello stato dei dischi, scheduling di jobs, logging, ecc….

Implementazione

I programmi di sistema sono implementati usando le API, come i programmi applicativi. Possiamo rappresentare questo concetto attraverso un esempio:

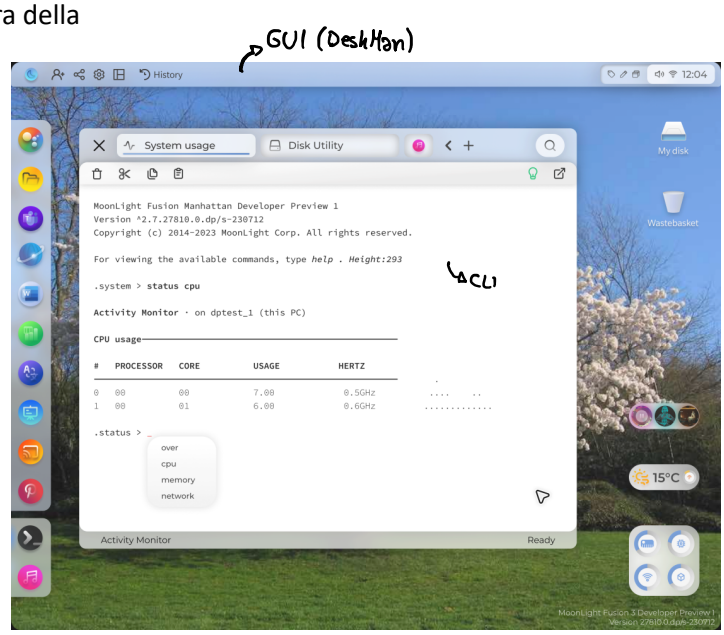
`cp in.txt out.txt`

- Copia il contenuto del file in.txt in un file out.txt
- Se il file out.txt esiste, il contenuto precedente viene cancellato, altrimenti out.txt viene creato.

- **Apri** in.txt in lettura
    - Se non esiste:
        - **Scrivi** un messaggio di errore sul terminale
        - **Termina** il programma con codice di 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

GUI

L'interfaccia grafica (GUI) è di solito basata sulla metafora della scrivania, icone e delle cartelle, corrispondenti alle directory. Nate dalla ricerca presso lo XEROX PARC lab negli anni 70, è stata poi popolarizzata dal Macintosh. I dispositivi mobili usano una variante di quest'interfaccia, specificatamente fatta per i touchscreen: non hanno dispositivi di puntamento, si usano le gestures, comandi vocali e tastiere virtuali.



CLI

L'interprete dei comandi permette agli utenti di impartire in maniera testuale delle istruzioni al sistema operativo. In molti, è possibile configurare quale interprete dei comandi usare; in questo caso è chiamato **shell**. Esistono due modi per implementare un comando:

- **Built-in**: l'interprete esegue direttamente il comando
- **Come programma di sistema**: l'interprete manda in esecuzione il programma

Inoltre, spesso il CLI risconosce un vero e proprio linguaggio di programmazione.

Processi

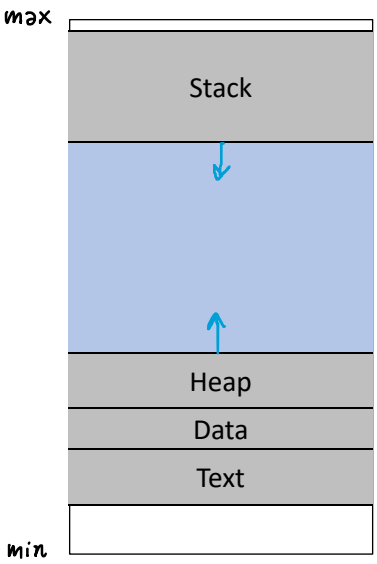
Un sistema operativo esegue un certo numero di programmi sullo stesso sistema di elaborazione. Il numero di programmi da eseguire può essere arbitrariamente elevato; per questo il sistema realizza e mette a disposizione un'astrazione detta **processo**, il quale è un'entità attiva astratta definita dal sistema operativo allo scopo di eseguire un programma. Il Process Control Block è una struttura dati che contiene tutte le info di un processo attivo.

Il processo è un **esecutore di un programma**, o un programma in esecuzione. È composto da diverse parti:

- Lo stato dei registri del processore che esegue il programma, incluso il program counter
- Lo stato dell'**immagine** del processo
- Le risorse del sistema operativo in uso al programma
- Diverse informazioni sullo stato del processo del sistema operativo

Ogni processo ha immagini distinte, poiché operano su zone di memoria centrale separate. Le risorse del sistema operativo invece possono essere condivise tra processi. Inoltre, sono organizzati in maniera gerarchica (vedi sezione "Creazione").

Ogni processo ha un **PID** (Process Identifier), che è un numero che identifica univocamente un processo in esecuzione in un sistema operativo.



Immagine

L'immagine mostra la struttura di un processo e le sue relazioni con il sistema operativo e le risorse hardware.

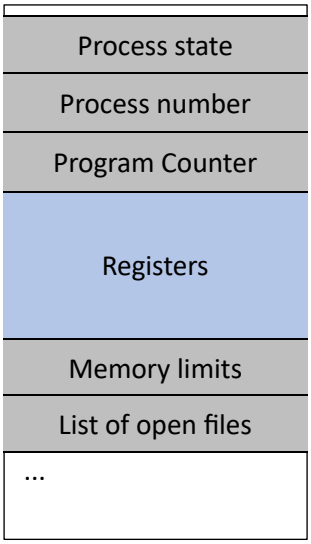
L'intervallo di indirizzi di memoria *min…max* in cui è contenuta l'immagine di un processo è detto **address space**, mentre di norma contiene:

- Dimensioni costanti
  - **Text section**: contiene il codice macchina del programma, indirizzo più basso
  - **Data section**: contiene le variabili globali
    - **Dati statici:** dati che non cambiano durante l'esecuzione del programma
    - **Data section**: variabili globali
- Dimensioni dinamiche
  - **Heap**: contiene la memoria allocata dinamicamente durante l'esecuzione
  - **Stack** delle chiamate: contiene parametri, variabili locali, indirizzi di ritorno delle varie procedure

Process Control Block

È una struttura dati del kernel, che contiene tutte le informazioni relative ad un processo:

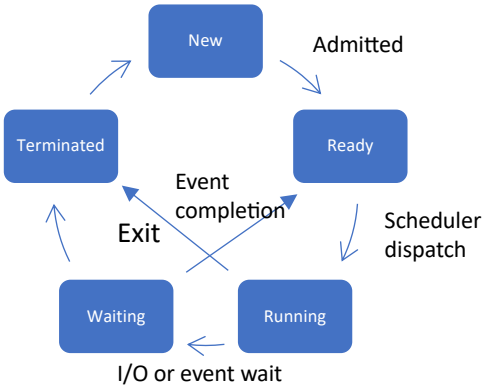
- **Process state**: ready, running, etc.
- **Process number (PID)**: identifica il processo
- **Program counter**: contenuto del registro "istruzione successiva"
- **Registers**: contenuto dei registri del processore
- **Informazioni relative alla gestione della memoria**: memoria allocata al processo
- **Informazioni sull'I/O:** dispositivi assegnati al processo, elenco file aperti, ecc….
- **Informazioni di scheduling:** priorità, puntatori a code di scheduling…
- **Informazioni di accounting:** CPU utilizzata, tempo trascorso..



Stato di un processo

Durante l'esecuzione, un processo cambia più volte il suo stato. Gli stati possibili sono:

- **New (nuovo)**: il processo è creato, ma non è ancora ammesso all'esecuzione
- **Pronto (ready)**: il processo può essere eseguito; è in attesa che gli sia assegnata una CPU
- **In esecuzione (running)**: le sue istruzioni vengono eseguite da qualche CPU
- **In attesa (waiting)**: il processo non può essere eseguito perché è in attesa che si verifichi qualche evento, come per esempio il completamento di un'operazione di I/O
- **Terminato (terminated)**: il processo ha terminato l'esecuzione

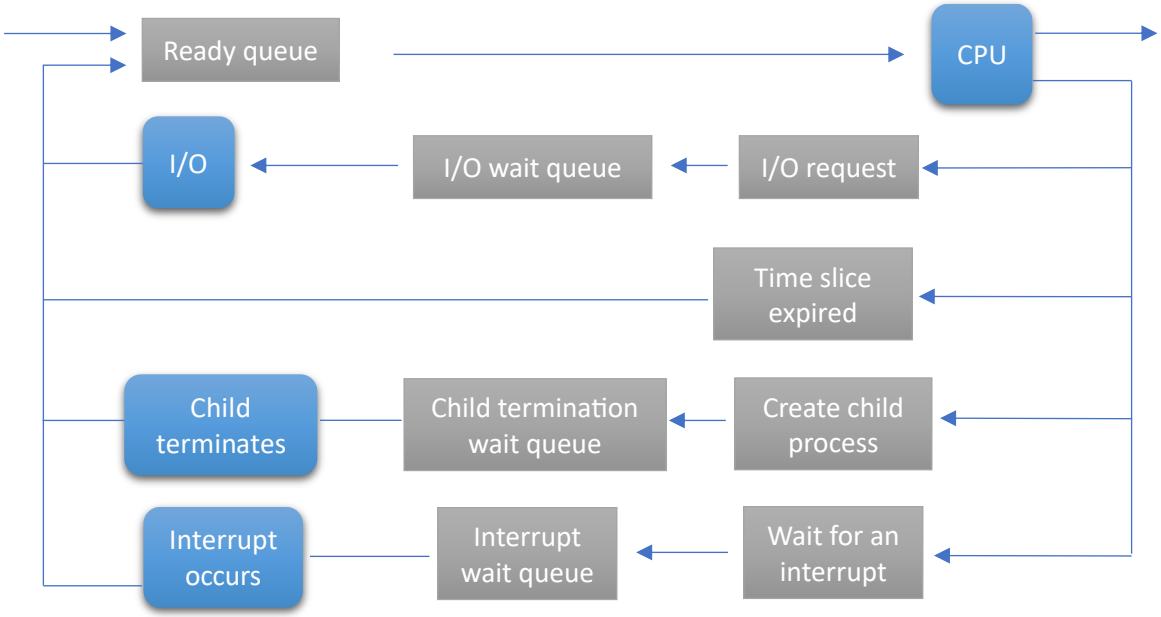


Scheduler della CPU

Lo scheduler della CPU, o scheduler a breve termine, seleziona il prossimo processo da eseguire tra quelli pronti per l'esecuzione, e assegna un core libero ad esso. Gestisce diverse code di processi, tra cui:

- **Ready queue:** contiene i processi residenti in memoria e pronti per l'esecuzione
- **Wait queues:** code destinate ai processi presenti in memoria, ma in attesa di un particolare evento. Mantenuta coda separata per ciascun tipo di evento di attesa.

Durante il ciclo di vita di un processo, rappresentato dal suo PCB, il processo può spostarsi da una coda all'altra in base al suo stato corrente.



Schemi di scheduling

Lo scheduler della CPU è responsabile di decidere quale processo nella **ready queue** assegnare a un core libero. Le decisioni di scheduling possono essere prese in diversi momenti, in corrispondenza dei cambiamenti di stato dei processi:

1. Quando un processo passa dallo stato *running* a *waiting*
2. Quando un processo passa dallo stato *running* a *ready*
3. Quando un processo passa dallo stato *waiting* a *ready*
4. Quando un processo termina

Se il riassegnamento viene effettuato solo nelle situazioni **1 e 4**, lo schema di scheduling è detto **cooperativo**; in questo caso, un core viene liberato solo quando un processo volontariamente rinuncia ad esso.

Altrimenti, se il riassegnamento può avvenire anche nelle situazioni **2 e 3**, lo schema è definito **preemptive**; un core può essere liberato anche perché il kernel forza la rimozione del core da un processo che lo sta utilizzando. Questo è più complicato da implementare, e solleva alcune questioni, come la gestione di processi che condividono dati, e l a gestione di un processo che esegue in modalità kernel.

Commutazione di contesto e dispatcher

Quando il controllo della CPU passa da un processo a un altro, scelto dallo scheduler a breve termine, si verifica una **commutazione di contesto**, nota anche come **context switch**. Questa operazione è eseguita dal dispatcher, il quale:

- **Salva il contesto** (stato, registri, ecc….) del processo da interrompere nel suo PCB.
- **Carica** il contesto del processo da eseguire dal suo PCB
- **Passa** in modalità utente.
- **Salta** al punto corretto del programma del processo selezionato, dove era stato precedentemente interrotto.

È importante notare che il dispatcher implementa un meccanismo tipico, mentre lo scheduler implementa una politica tipica.

Latenza di dispatch

La latenza è il tempo impiegato sul dispatcher per fermare un processo, e avviarne un altro. Questa latenza rappresenta un **overhead puro**, poiché durante questo intervallo, non viene eseguito alcun lavoro utile. Maggiore è la complessità dell'OS e del PCB, maggiore sarà la latenza di dispatch. Alcuni processori offrono supporto per minimizzare la latenza di dispatch.

Algoritmi di scheduling

Confrontare gli algoritmi di scheduling è una sfida, poiché praticamente ogni OS ha le sue politiche specifiche. L'assenza di una politica di scheduling universalmente migliore suggerisce che la scelta dipende dai requisiti specifici e dal carico del sistema.

Per valutare gli algoritmi, dobbiamo tenere in considerazione dei dati criteri:

- **Utilizzo della CPU**: percentuale di tempo in cui la CPU è attiva nell'esecuzione dei processi utente
- **Throughput**: numero di processi completati in un'unità di tempo, dipendente dalla durata dei processi
- **Tempo di completamento**: tempo necessario per completare l'esecuzione di un processo, influenzato dalla durata del processo, carico totale, durata operazioni I/O
- **Tempo di attesa**: tempo trascorso dal processo nella coda di attesa, meno dipendente dalla durata del processo, e dalle operazioni di I/O rispetto al tempo di completamento
- **Tempo di risposta**: è il tempo trascorso tra l'arrivo di una richiesta al processo e la produzione della prima risposta, escludendo l'emissione nell'output

Scheduling in ordine di arrivo

Con questo algoritmo, la CPU viene assegnata al primo processo che la richiede. L'implementazione richiede una semplice coda FIFO con nessuna prelazione, ma può verificarsi un **effetto convoglio**, ovvero dove il tempo di attesa medio può essere lungo.

Processo	Durata burst CPU	Tempo attesa
P_2	3	0
P_3	3	3
P_1	24	6

Ordine di arrivo

Tempo di attesa medio = (0 + 3 + 6) / 3 = 3 **(ridotto a circa 1/6)**
Tempo di completamento medio = (3 + 6 + 30) / 3 = 13 **(ridotto a circa 1/2)**

Scheduling per brevità

La CPU viene assegnata al processo che ha il successivo CPU burst più breve. Ha un'implementazione quasi identica a FCFS, ma **minimizza il tempo di attesa medio**. Di solito, però, non si sa in anticipo quale sia il processo che avrà il CPU burst più breve.

Processo	Durata burst CPU	Tempo attesa
P_4	3	0
P_1	6	3
P_3	7	9
P_2	8	16

Ordine per brevità

Tempo di attesa medio SJF = (0 + 3 + 9 + 16) / 4 = 7
Tempo di completamento medio SJF = (3 + 9 + 16 + 24) / 4 = 13

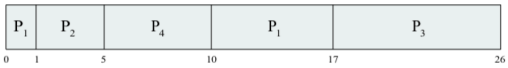
Shortest-remaining-time-first

Utilizza la prelazione per gestire il caso in cui i processi non arrivino tutti nello stesso istante; se nella ready queue arriva un processo con un burst più corto di quello running, quest'ultimo viene prelazionato dal nuovo processo.

- Con preemption e tempo di arrivo
- Il tempo di attesa di un processo è:
  istante terminazione processo - (tempo di arrivo + durata burst)
- Il tempo di completamento di un processo invece è:
  istante terminazione processo - tempo di arrivo

Processo	Tempo di arrivo	Durata burst CPU
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Determina l'ordine di arrivo



Tempo di attesa medio = ((17-8) + (5-5) + (26-11) +(10-8)) / 4 = 6,5
Tempo di completamento medio = ((17-0) + (5-1) + (26-2) +(10-3)) / 4 = 13

Scheduling circolare

Ogni processo ottiene una piccola quantità fissata di tempo di CPU, di solito 10-100 millisecondi, per il quale può essere in esecuzione.
Trascorso tale tempo, il processo in esecuzione viene interrotto e messo in fondo alla ready queue, che è gestita in maniera FIFO.
In tal modo, la ready queue funziona essenzialmente come un buffer circolare, e i processi vengono scanditi dal primo all'ultimo, per poi ripartire dal primo nello stesso ordine. Esiste anche un **timer** che genera un interrupt periodico con periodo q, per effettuare la prelazione del processo corrente.

- Ricordiamo che il tempo di attesa di un processo è:
  istante terminazione processo - (tempo di arrivo + durata burst)
- E il tempo di completamento di un processo è:
  istante terminazione processo - tempo di arrivo
- In questo esempio il tempo di arrivo è 0 per tutti i processi

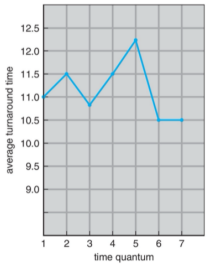
Processo	Durata burst CPU
P_1	24
P_2	3
P_3	3

Ordine di arrivo

q = 4
Tempo di attesa medio = ((30-24) + (7-3) + (10-3)) / 3 = 5,67
Tempo di completamento medio = (30 + 7 + 10) / 3 = 15,67

Se ci sono n processi nella ready queue, e il quanto temporale è q:

- Nessun processo attende più di **q(n-1)** unità di tempo nella ready queue prima di ridiventare running per un altro quanto di tempo.
- Ogni processo ottiene **1/n** del tempo totale di CPU, in maniera equa



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Il suo comportamento rispetto a q varia:

- **Q è elevato**: se q è di regola maggiore della durata di tutti i burst, RR tende al FCFS.
- **Q è basso**: deve comunque essere molto più lungo della latenza di dispatch, altrimenti questa si mangia un tempo comparabile al tempo di esecuzione dei processi utente, e l'utilizzo della CPU diventa in accettabilmente basso.

La performance di questo algoritmo:

- Rispetto a SJF, tipicamente ha un tempo di completamento medio più alto, ma un tempo di risposta medio più basso.
- Il tempo di completamento medio non necessariamente migliora con l'aumento di q.

Scheduling con priorità

Ad ogni processo è associato un numero intero, che indica la sua priorità. Viene eseguito il processo con priorità più alta, mentre gli altri aspettano (nei sistemi Unix, più è basso è il numero, più è alta la priorità. In Windows, invece, più è alto il numero, più è alta la priorità). Ha la possibilità di essere pre-emotive, o no.

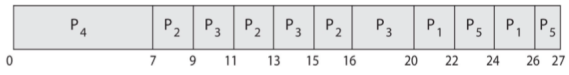
Possono essere permessi più processi con pari priorità o no; in caso positivo, occorre stabilire un **secondo algoritmo di scheduling**, per arbitrare tra i processi a pari priorità; di solito, si utilizza RR.

SJF è scheduling con priorità, dove la priorità è l'inverno della durata del CPU burst. C'è un problema però dell'**attesa indefinita**: un processo a priorità troppo bassa potrebbe non venir mai schedulato. La soluzione rimane nell'**invecchiamento**, ossia l'aumento automatico di priorità di un processo al crescere del tempo di permanenza nella ready queue.

Scheduling con priorità + RR: esempio

Processo	Durata burst CPU	Priorità (UNIX)
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Ordine di arrivo



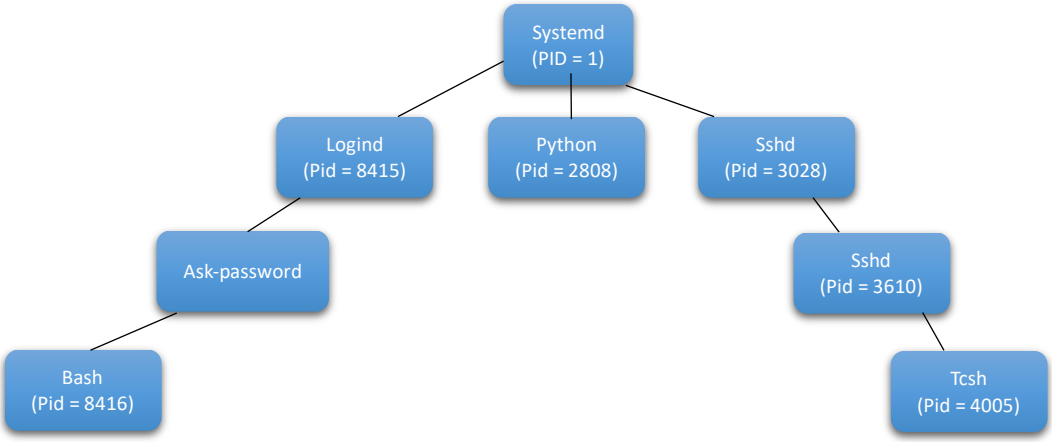
q = 2 per processi con stessa priorità
Tempo di attesa medio = ((26 - 4) + (16 - 5) + (20 - 8) + (7 - 7) + (27 - 3)) / 5 = 13,8
Tempo di completamento medio = (26 + 16 + 20 + 7 + 27) / 5 = 19,2

Operazioni sui processi

I sistemi operativi forniscono delle chiamate di sistema con le quali un processo può creare/terminare/manipolare altri processi. Dal momento che solo un processo può creare un altro processo, all'avvio il sistema operativo crea dei **processi primordiali** dai quali tutti i processi utente e di sistema vengono progressivamente creati.

Creazione dei processi

I processi sono organizzati in maniera gerarchica, il che vuol dire che ogni processo ha un processo padre e un processo figlio, a parte l'iniziale che viene inizializzato **dal kernel stesso** (per Windows: wininit.exe/per Unix: init/systemd). Quindi quello che si crea è effettivamente un **albero di processi**:



La relazione padre/figlio è di norma importante per le politiche di condivisione delle risorse e di coordinazione tra processi. Di seguito alcune possibili politiche:

- Possibili politiche di **condivisione di risorse**: padre e figlio condividono tutte le risorse, o un opportuno sottoinsieme, o nessuna
- Possibili politiche di **creazione di spazio di indirizzi**: il figlio è un duplicato del padre oppure no, e bisogna specificare quale programma deve eseguire il figlio
- Possibili politiche di **coordinazione padri/figli**: il padre è sospeso finchè i figli non terminano, oppure eseguono in maniera concorrente.

Primo processo in Unix

In Unix, i processi sono creati tramite la funzione *fork()*, che restituisce al processo padre il PID del processo figlio appena creato. Il PID 0 è riservato allo swapper, che si occupa della gestione della memoria, mentre il PID 1 è riservato al processo **init**, che avvia e spegne il sistema. Una volta raggiunto il limite di assegnazione, l'assegnazione ricomincia da un valore minimo. Per conoscere il proprio PID, si possono usare le funzioni *getpid()* e *getppid()*.

Primo processo in Windows

In Windows, i processi sono creati tramite la funzione *CreateProcess()*, che restituisce al processo padre un handle al processo figlio appena creato. I processi **sono indipendenti tra loro**, e non ha un valore speciale per il primo processo processo avviato dal sistema; invece, è assegnato casualmente da un algoritmo che cerca di evitare le collisioni. I processi possono conoscere il proprio PID tramite la funzione *GetCurrentProcessId()* e *GetProcessId()*.

Terminazione dei processi

I processi, di regola, richiedono esplicitamente la propria terminazione al sistema operativo. Ecco qua alcuni casi:

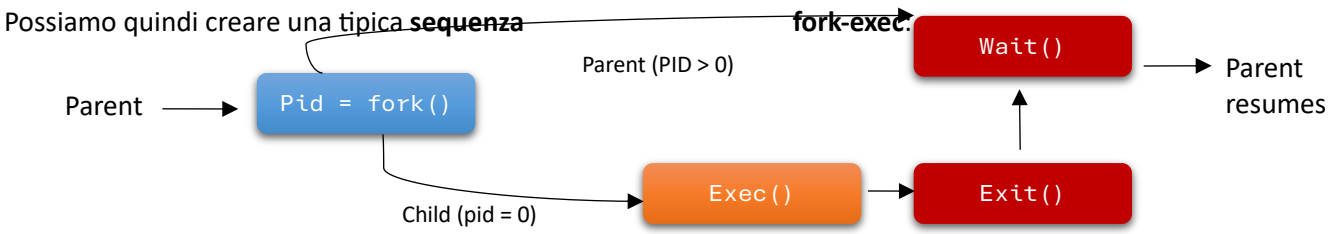
- Un processo padre può attendere o meno la terminazione di un figlio
- Un processo padre può forzare la terminazione di un figlio (se il figlio sta usando risorse in eccesso, le funzionalità del figlio non sono più richieste, il padre termina prima che il figlio termini)

Alcuni sistemi operativi non permettono ai processi figli di esistere dopo la terminazione del padre: abbiamo quindi una **terminazione in cascata**, o una terminazione forzata dal sistema operativo.

Le API POSIX per operazioni su processi

- `fork()` crea un nuovo processo figlio
  - il figlio è un **duplicato** del padre, ed esegue concorrentemente ad esso
  - Ritorna al padre un PID del processo figlio, e al figlio il PID 0

- `exec()` sostituisce il programma in esecuzione da un processo con un altro programma che viene eseguito dall'inizio
    - viene tipicamente usata dopo una fork() dal figlio per iniziare ad eseguire un programma diverso dal padre
- `wait()` è chiamata dal padre per attendere la fine dell'esecuzione di un figlio
    - ritorna il PID del figlio che è terminato, e il codice di ritorno del figlio
- `exit()` fa terminare il processo che la invoca
    - accetta come parametro un codice di ritorno numerico
    - Il sistema operativo elimina il processo e recupera le sue risorse
    - Restituisce al processo padre il codice di ritorno
    - Viene implicitamente invocata se il processo esce dalla funzione *main*
- `abort()` fa terminare forzatamente un processo figlio



Comunicazione interprocesso

Più processi possono essere indipendenti o cooperare; quando cooperano, vuol dire che il comportamento di un processo influenza o è influenzato dal comportamento di un altro processo. I motivi per i quali può essere utile avere più processi cooperanti sono una condivisione di informazioni, l'accelerazione di computazioni, e modularità ed isolamento.
Per permettere ai processi di cooperare, il sistema operativo deve mettere a disposizione **primitive di comunicazione interprocesso**, le quali sono di due tipi: *memoria condivisa* e *message passing*.

Memoria condivisa

Viene stabilita una zona di memoria condivisa tra i processi che intendono comunicare. La comunicazione è controllata dai processi che comunicano, non dal sistema operativo.
Un problema importante è permettere ai processi che comunicano tramite memoria condivisa di sincronizzarli, e di non permettere che un processo legga la memoria condivisa mentre l'altro la sta scrivendo. Per questo, i sistemi operativi mettono a disposizione ulteriori primitive per la sincronizzazione.

Dopo l'esecuzione della primitiva fork(), il processo padre e figlio non condividono lo spazio di memoria.

Pipe

Sono canali di comunicazione tra i processi, e possono essere considerati una forma di message passing.
Esistono varie varianti*: unidirezionale, bidirezionale (Half-duplex o full-duplex), relazione tra processi comunicanti, usabili o meno in rete*.

Pipe convenzionali

Le pipe convenzionali sono unidirezionali; questi non sono accessibili al di fuori del processo creatore, quindi di solito sono condivise con un processo figlio attraverso una *fork()*. In Windows, queste sono chiamate pipe anonime.

Named pipes

Le named pipes sono bidirezionali. Esistono anche dopo la terminazione del processo che le ha create, e dunque non richiedono una relazione padre-figlio tra i processi che le usano.

Un processo deve aprire una named pipe in lettura, il cui nome è /usr/local/mypipe. Quindi, si usa *open("/usr/local/mypipe", 0_RDONLY);*.

In Unix e in Windows

In Unix le pipes sono solo half-duplex, sulla stessa macchina, e sono solo dati byte-oriented. In Windows invece, le pipes sono full-duplex, anche tra macchine diverse, ed include anche dati message-oriented.

Memoria condivisa in POSIX

Un processo crea un segmento di memoria condivisa con la funzione `shm_open`:

```
int shm_fd = shm_open(name, 0_CREAT | 0_RDWR, 0666);
```

È anche usato per aprire un segmento precedentemente creato; quindi, imposta la dimensione del segmento con la funzione `ftruncate`:

```
ftruncate(shm_fd, 4096);
```

Infine, la funzione `mmap` mappa la memoria condivisa nello spazio di memoria del processo:

```
void *shm_ptr = mmap(0, 4096, PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

Da questo momento, si può usare il puntatore `shm_ptr` ritornato da `mmap` per leggere o scrivere la memoria condivisa. mmap serve per mappare parte degli indirizzi di uno spazio di indirizzamento virtuale su un file, su memoria condivisa o su memoria centrale (estensione non POSIX).

Pipe anonime in POSIX

Vengono create con la funzione `pipe`, che ritorna due descrittori, uno per il punto di lettura e uno per il punto di scrittura.

```
int p_fd[2];
int res = pipe(p_fd);
```

Le funzioni `read` e `write` permettono di leggere e scrivere:

```
ssize_t n_wr = write(p_fd[1], "Hello, World!", 14);
char buffer[256];
ssize_t n_rd = read(p_fd[0], buffer, sizeof(buffer) -1);
```

È possibile utilizzare la funzione `fdopen` per fare il wrapping di un punto della pipe in un file, ed utilizzare le funzioni C `stdio` con esso.

Named pipes in POSIX

Le named pipes vengono anche chiamate **FIFO** nei sistemi POSIX. Per creare una FIFO si utilizza l'API `mkfifo`:

```
int res = mkfifo("/home/e/myfifo", 0640);
```

La FIFO si utilizza come un normale file:

```
int fd = open("/home/e/myfifo", 0_RDONLY);
char buffer[256];
ssize_t n_rd = read(fd, buffer, sizeof(buffer) -1);
```

Al termine dell'utilizzo, è importante ricordarsi di chiuderla usando `close(fd)`. Per eliminare la FIFO, si usa l'API `unlink`:

```
unlink("/home/e/myfifo");
```

Segnali POSIX

Per invitare un segnale ad un processo bisogna usare l'API `kill`:

```
int ok = kill(1000, SIGTERM); //terminazione al processo 1000
```

Per registrare una callback per un determinato segnale, è opportuno utilizzare l'API `sigaction`:

```
struct sigaction act;
sigemptyset(&act.sa_mask); //non bloccare altri segnali
acct.sa_flags = SA_SIGINFO; //callback in act.sa_sigaction
act.sa_sigaction = sigterm_handler //callback
int ok = sigaction(SIGTERM, &act, NULL);
```

Si usano dei **puntatori a delle funzioni**, che punta all'indirizzo di memoria della prima operazione di eseguire di una certa funzione. Prendo il nome della funzione, e se ci metto la parentesi la invoco, se non ce le metto allora prende il primo indirizzo di memoria.

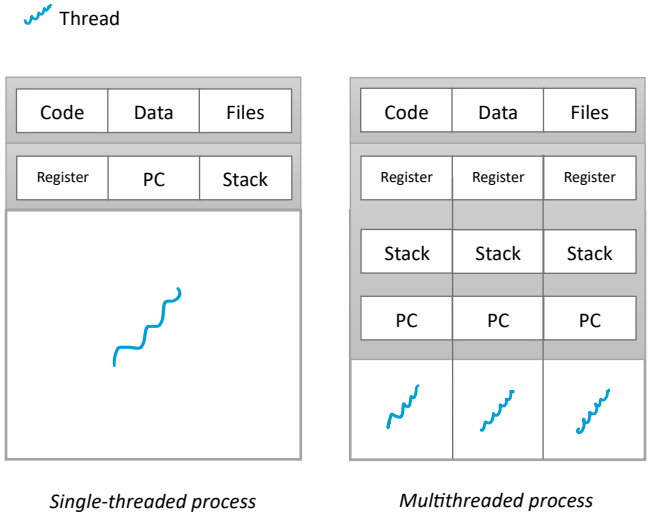
Notifiche con callback

In alcuni sistemi operativi, un processo può notificare un altro processo in maniera da causare l'esecuzione di un blocco di codice, similmente ad un interrupt.

In un sistema Unix, tali 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.
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

Fino ad ora, abbiamo assunto che un processo abbia un singolo flusso di esecuzione sequenziale, ossia, un singolo processore virtuale. Se supponiamo che un processo possa avere molti processori virtuali, più istruzioni possono eseguire concorrentemente, e quindi il processo può avere più percorsi (**thread**) di esecuzione concorrenti.



Single-threaded process *Multithreaded process*

I thread di uno stesso processo condividono la memoria globale (**data**), la memoria contenente il codice (**code**) e le risorse ottenute dal sistema operativo (ad esempio i file aperti). Ogni thread di uno stesso processo però deve avere un proprio stack, altrimenti le chiamate a subroutine di un thread interferirebbero con quelle di un altro thread concorrente.

Nei sistemi Unix, questi thread sono chiamati **posix phtreads**, che non sono un'implementazione, ma una specifica. L'API per creare un nuovo thread è **clone()**.

Un programmatore può accedervi se queste funzionalità sono incorporate nel linguaggio che usa, sono accessibili tramite una API specifica del sistema operativo, o tramite una libreria implementata su diversi sistemi operativi.

I thread possono essere:

- **A livello utente**: i thread disponibili nello spazio utente dei processi, e sono quelli offerti dalle librerie di thread ai processi.
- **A livello kernel**: i thread implementati nativamente dal kernel, e sono utilizzati per strutturare il kernel stesso in maniera concorrente.

Modelli di supporto al multithreading

I thread a livello del kernel vengono utilizzati dalle librerie di thread per implementare i thread a livello utente di un certo processo; a tale scopo, possono essere adottate diverse strategie:

- **Molti-a-uno**: i thread a livello utente di un processo sono implementati su un unico thread a livello del kernel.
    - **Vantaggio**: usabile su ogni sistema operativo
    - **Svantaggio**: se un thread utente fa una chiamata di sistema bloccante, blocca tutti i thread utente dello stesso processo
        - Non sfrutta la presenza di più core.
- **Uno-a-uno**: ogni thread a livello utente è implementato su un singolo e distintivo thread a livello del kernel. Questo modello offre un maggiore parallelismo, ma può generare un sovraccarico considerevole.
    - **Vantaggio**: permette un maggior grado di concorrenza
        - Permette di sfruttare il parallelismo nei sistemi multi core
    - **Svantaggio**: minore performance rispetto al modello molti-a-uno
        - Stress del kernel
- **Molti-a-molti**: i thread a livello utente di un processo sono implementati su un insieme di thread a livello del kernel, che potrebbero essere in numero inferiore. L'associazione tra thread utente e thread kernel è dinamica, e stabilita da uno scheduler interno alla libreria di thread. Cerca di combinare i vantaggi.
    - **Svantaggio**: complesso da implementare (la libreria di thread deve dinamicamente alternare l'esecuzione dei thread a livello utente sui thread a livello del kernel disponibili
    - **Modello a due livelli**: permette agli utenti di creare dei thread che hanno un mapping uno-a-uno con un thread a livello del kernel

Thread control blocks

Nei sistemi kernel multithreaded, vengono utilizzate strutture dati simili ai PCB chiamate **Thread Control Blocks (TCB)**; ogni TCB memorizza il contesto e le informazioni di contabilizzazione di un thread. A differenza del PCB, che contiene solo le informazioni comuni di contesto e contabilizzazione, come lo spazio di memoria, il TCB è specifico per ciascun thread kernel utilizzato dal processo.

Il PCB è solitamente associato ai TCB dei thread kernel utilizzati da un dato processo. I TCB dei thread kernel associati a un processo sono collegati al PCB del processo stesso. Questa organizzazione facilita la gestione e la sincronizzazione dei thread all'interno del kernel multithreaded.

Creare un nuovo thread

All'inizio, un processo viene creato con un singolo thread. Per creare un nuovo thread si utilizza l'API **`pthread_create`**, per attendere la fine dell'esecuzione di un thread si utilizza invece l'API **`pthread_join`**:

```
void *thread_code(void *name) { … }
…
Pthread_id tid1, tid2;
int ok1 = pthread_create(&tid1, NULL, thread_code, “thread 1”);
int ok2 = pthread_create(&tid2, NULL, thread_code, “thread 2”);
…
```

```
void *ret1, *ret2;
ok1 = pthread_join(tid1, &ret1);
ok2 = pthread_join(tid2, &ret2);
```

Aspetti particolari nelle API per il multi threading

Chiamate di sistema fork() ed exec()

Una `fork()` dovrebbe duplicare soltanto il thread chiamante o tutti i thread? Alcuni sistemi operativi Unix-like hanno due diverse `fork()`. **`exec()`** è invece invocata da un thread che ha effetto sugli altri thread? Di solito, termina tutti i thread del processo precedentemente in esecuzione.

Gestione dei segnali

Quando un processo è single-threaded, un segnale interrompe l'unico thread del processo. Quando vi sono quindi più thread, quale thread riceve il segnale? Abbiamo varie soluzioni:

- il thread a cui si applica 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, e può essere invocata da un altro thread. Abbiamo due approcci:

- **Cancellazione asincrona**: il thread che riceve la cancellazione viene terminato immediatamente
    - Non c'è nessuna necessità di controllare periodicamente se ci sono richieste di cancellazione pendenti
- **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.
    - Dal momento che un thread controlla il momento della propria cancellazione, può effettuare una terminazione ordinata.

All'interno dei pthreads, si può attivare/disattivare la cancellazione, ed avere entrambi i tipi di cancellazione. Se la cancellazione è inattiva, le richieste di cancellazione rimangono in attesa fino a quando è attivata. Nel caso della 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()`.

```
void *thread_code(void *name) {
     int oldtype, oldstate;
     pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);
     pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &old state);
     while(true){
          pthread_testcancel()
          }
}
Pthread_id tid;
int ok = pthread_create(&tid, NULL, thread_code, “thread 1”);
ok = pthread_cancel(tid);
void *ret;
ok = pthread_join(tid, &ret);
```

Dati locali dei thread

In alcuni casi, è utile assegnare ad un thread dei stati locali (**thread local storage**) non condivisi con gli altri thread dello stesso processo. La TLS è diversa dalle variabili locali; infatti, è visibile a tutte le funzioni. È simile ai dati static del linguaggio C, ma unica per ciascun thread; è utile quando il programma non ha un controllo diretto sul momento di creazione dei thread.

Multiprogrammazione e multitasking

Nei sistemi operativi, i programmatori cercano due obiettivi:

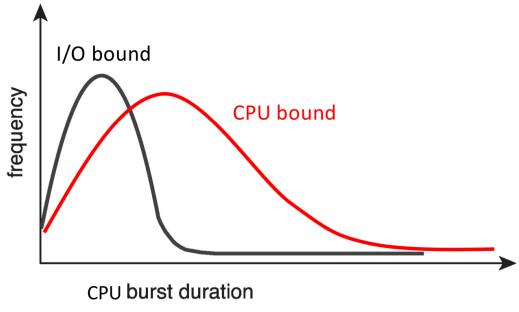
- **Efficienza**: mantenere impegnata la CPU il maggior tempo possibile nell'esecuzione dei programmi, se ci sono programmi da eseguire.
- **Reattività**: dare l'illusione che ogni processo progredisca continuamente nella propria esecuzione, come se avesse una CPU dedicata; particolarmente importante per i programmi interattivi.

Le due tecniche adottate nei sistemi operativi per ottenere questi due obiettivi sono:

- **Multiprogrammazione**, il quale obiettivo è impedire che un programma che non è in condizione di proseguire l'esecuzione mantenga la CPU
    - Il sistema operativo mantiene in memoria le immagini dei processi da eseguire. Questo richiede che tutte le immagini di tutti i processi siano in memoria perché questi possano essere eseguibili.
        - Se i processi sono troppi, non possono essere contenuti tutti in memoria; quindi, si può usare la tecnica dello **swapping** per spostare le immagini dentro/fuori dalla memoria.
        - La **memoria virtuale** è un'altra tecnica che permette di eseguire un processo la cui immagine non è completamente in memoria.
    - Quando una CPU non è impegnata ad eseguire un processo, il sistema operativo seleziona un processo non in esecuzione e gli assegna la CPU
    - Quando un processo non può proseguire l'esecuzione, la sua CPU viene assegnata ad un altro processo non in esecuzione
    - Se i processi sono più delle CPU, queste saranno impegnate nell'esecuzione di qualche processo per la maggior parte del tempo.
- **Multitasking**, il quale obiettivo è che un programma interattivo possa reagire agli input utente in un tempo accettabile.
    - La CPU viene sottratta periodicamente al programma in esecuzione, ed assegnata ad un altro programma.
    - Tutti i programmi progrediscono in maniera continuativa nella propria esecuzione, anziché solo nei momenti in cui il programma che detiene la CPU si mette in attesa.

Burst CPU e I/O

L'obiettivo della multiprogrammazione è di **massimizzare l'utilizzo della CPU**; gli algoritmi di scheduling sfruttano il fatto che, di norma, l'esecuzione di un processo è una sequenza di *burst della CPU* e poi di *I/O* (attesa completamento dell'operazione di I/O).



Un programma con prevalenza di I/O si chiama **I/O bound**, e ha un: elevato numero di burst CPU brevi, ridotto numero di burst CPU lunghi, ed è tipico dei programmi interattivi. Nella curva, il massimo sta più a sinistra.

Un programma con prevalenza di CPU si chiama **CPU bound**, e ha un: elevato numero di burst CPU lunghi, ridotto numero di burst CPU brevi, ed è invece tipico dei programmi batch. Nella curva, il massimo sta più a destra.

Gestione della memoria

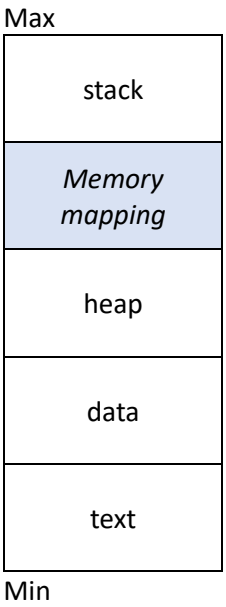
Il problema dell'allocazione della memoria

Perché un programma possa andare in esecuzione, esso deve avere a disposizione il **processore** per eseguire il codice, e la **memoria centrale** per memorizzare il codice e i dati sul quale il codice opera.

Solo nei sistemi operativi più semplici un solo programma alla volta è in memoria: nei moderni sistemi operativi, molti programmi sono contemporaneamente in memoria in uno stesso istante. **Più immagini di più processi sono presenti contemporaneamente nella memoria centrale**; quindi, il sistema operativo deve allocare porzioni di memoria centrale ai diversi processi in funzione delle necessità di tali processi.

Lo spazio di indirizzamento

Ogni processo ha a disposizione uno spazio di indirizzamento che può usare per le proprie operazioni. Nei primi sistemi operativi, tale spazio di indirizza mento era il range di indirizzi di memoria centrale che veniva assegnato al processo: se l'immagine di un certo processo avesse avuto dimensione 1MB, e fosse caricata in memoria centrale dall'indirizzo *001B:0000*, il suo spazio di indirizzamento sarebbe stato *002B:0000*. Questo però non permette di caricare lo stesso programma in zone diverse di memoria!



Spazio di indirizzamento virtuale

Nei moderni sistemi operativi, ciascun processo è dotato di uno **spazio di indirizzamento virtuale**, chiamato anche VAS, che è indipendente dagli indirizzi fisici della memoria centrale in cui il programma è caricato.
Si estende da un indirizzo base a un massimo consentito, in base all'architettura del processore utilizzata. Per far corrispondere il VAS del processo con la porzione di memoria in cui il programma è effettivamente caricato, vengono usate **tecniche di associazione degli indirizzi durante l'esecuzione**.

Di solito, lo spazio è molto più ampio rispetto alla quantità di memoria centrale fisica disponibile, e questo vuol dire che gran parte del VAS non è direttamente utilizzabile dal processo, poiché **non è associato a nessuna regione di memoria centrale**. Questa parte è di solito posizionata tra lo stack e lo heap, e questa zona è solitamente inaccessibile; per rendere questo possibile, gli OS forniscono API che consentono di mappare una regione centrale su quella zona, effettuando quindi **memory mapping**. Forniscono anche la possibilità di mappare una regione del VAS direttamente sul contenuto di un file, creando i cosiddetti **file mappati in memoria**; questo approccio consente l'accesso ai dati contenuti nel file utilizzando le istruzioni di accesso alla memoria, anziché le API del filesystem.

Strategia di allocazione continua

È la strategia più semplice di allocazione della memoria in un sistema multi-programmato; la memoria centrale è partizionata in due zone, una per il sistema operativo, e una per i processi utente. Ogni processo utente occupa un'area contigua di memoria nella partizione dei processi utente, e in quell'area viene caricata la sua immagine.

Protezione con registro base e limite

È il più semplice metodo di protezione utilizzabile con l'allocazione contigua. Il processore possiede due registri: un registro **base**, ed un registro **limite**. Il registro base contiene il più piccolo indirizzo della memoria fisica che il processo corrente ha il permesso di accedere, mentre il registro limite determina la dimensione dell'intervallo ammesso.

I registri base e limite **possono essere impostati solo in kernel mode**; in modalità utente, il processore proibisce tutte le operazioni di lettura/scrittura fuori dall'intervallo individuato dai registri base e limite.

Nel caso in cui venga generato un indirizzo fuori dall'intervallo, l'indirizzo non viene messo sul bus, e viene generata un'eccezione. In modalità di sistema, il processore può accedere a tutta la memoria.

Gli **svantaggi** sono che non fornisce uno spazio di indirizzamento virtuale ai processi. L'unico modo per implementare uno spazio di indirizzamento virtuale con tale soluzione sarebbe il **binding in fase di caricamento**, ma questo è molto lento; pertanto, si preferiscono soluzioni basate sul binding in fase di esecuzione.

Partizioni variabili

Ogni processo riceve una partizione di memoria distinta; l'approccio a **partizioni variabili** assegna a ciascun processo una partizione della dimensione esatta, necessaria per la sua esecuzione.

Immaginiamo quindi la memoria come un puzzle, in cui i buchi rappresentano *regioni di memoria libera*. Ogni processo ottiene la sua porzione di memoria da un buco abbastanza grande per contenerlo. Quando un processo termina, rilascia la sua partizione, creando un nuovo gioco.

L'OS mantiene una lista dei buchi disponibili nella memoria centrale; quando si crea un nuovo processo, il sistema operativo deve scegliere un buco strategico per allocare la memoria necessaria al nuovo arrivato. Per fare questo, si usano diverse strategie di allocazione di buchi: *first fit*, *best fit, worst fit.*

Frammentazione

Suddividiamo la frammentazione in due categorie:

1. **Frammentazione esterna**: si verifica quando la memoria libera è sufficiente per creare un nuovo processo, ma *dispersa tra buchi non contigui troppo piccoli*
    a. Potrebbe esserci spazio disponibile, ma è frammentato in modo che non sia utilizzabile per soddisfare le esigenze di un nuovo processo.
2. **Frammentazione interna**: quando la memoria allocata per un processo è più grande di quanto effettivamente necessario
    a. Una partizione può contenere memoria inutilizzata, contribuendo allo spreco di risorse.

Un possibile rimedio è la **compattazione**: sposta le partizioni in modo da poter unire buchi separati, e ridurre la frammentazione. Tuttavia, è importante notare che la compattazione richiede il binding in fase di esecuzione, ovvero la **rilocazione dinamica**, la quale è computationazionalmente oneroso, e può presentare sfide quando i processi coinvolgono operazioni di input/output.
Inoltre, se un processo sta eseguendo questo genere di operazioni, potrebbe non essere rilocato, o in alternativa, l'I/O potrebbe essere gestito solo in buffer interni al sistema operativo.

Paging

In questo approccio, la memoria centrale è suddivisa in **frames**, i quali sono blocchi di dimensione fissa, mentre lo spazio di indirizzamento virtuale di ogni processo è diviso in **pagine**, blocchi delle stesse dimensioni dei frames. Una tabella delle pagine associa le pagine di un processo ai frames in memoria centrale, facilitando la traduzione degli indirizzi logici in fisici attraverso la MMU.

I vantaggi della paginazione includono l'assenza di frammentazione esterna, e la piena indipendenza tra indirizzi logici e fisici.

Per mitigare la frammentazione interna, si cerca di ridurre lo spreco di memoria con pagine di dimensione ridotte. Tuttavia, ciò può aumentare il numero totale di pagine e, di conseguenza, la dimensione della tabella delle pagine.

L'OS deve gestire la tabella delle pagine di ciascun processo, insieme a una **tabella dei frame** che indica lo stato di ogni frame (libero o assegnato). Se il kernel deve accedere alla memoria di un processo, deve

effettuare manualmente la traduzione degli indirizzi logici del processo in indirizzi fisici, poiché la MMU non è attiva quando la CPU è in modalità di sistema.

La tabella delle pagine è mantenuta in memoria centrale, e la MMU utilizza il **Page Table Base Register (PTBR)** per indicare l'indirizzo fisico dell'inizio della tabella delle pagine, e il **Page Table Length Register (PTLR)** per specificarne la dimensione. Per migliorare le prestazioni, si ricorre a una cache dedicata per la tabella delle pagine (**Translation Lookaside Buffer – TLB**), riducendo così la necessità di continui accessi in memoria durante l'accesso a dati o istruzioni.

Con la paginazione è facile condividere memoria fisica tra più processi: è sufficiente che i frame siano nelle tabelle delle pagine dei processi che li condividono.

Translation Lookaside Buffer

È una cache dedicata per la tabella delle pagine, di solito piccolo (da 64 a 1024 entries).
Un problema principale era che *ad ogni cambio di contesto, devo effettuare il flush del TLB, il che comporta una notevole riduzione di prestazioni*. Per tale motivi, alcuni TLB memorizzano un **Address Space Identifier (ASID)** nelle loro entry, che identificano univocamente uno spazio di indirizzi, così da poter mantenere le entry di più tabelle delle pagine.

Dato questo, possiamo calcolare anche il tempo effettivo di accesso alla memoria: ho un **TLB hit** se il numero di pagina di un indirizzo logico si trova nel TLB, altrimenti ho un **TLB miss**.
Il **tasso di successi** è dato dalla percentuale di TLB hit sul totale degli accessi; supponendo che il tempo di accesso alla memoria sia chiamato *ma*, e lo hit ratio sia *hr*, il **tempo medio di accesso alla memoria** è dato da:

$$\text{EAT} = \text{ma hr} + 2\ \text{ma}\ (1 - \text{hr}) = \text{ma}\ (2 - \text{hr})$$

Politiche di sostituzione

Quando si verifica un TLB miss, e il TLB è al completo, è necessario sostituire una voce; per fare questo, usiamo tre diverse strategie:

1. **Last Recently Used**: si sostituisce ll'entry che non è stata utilizzata per un periodo più lungo. Questo approccio mira a mantenere in memoria le pagine utilizzate di recente
2. **Round-robin**: si sostituisce l'entry seguente nella sequenza. Questa politica è basata sull'ordine circolare, assegnando l'entry successiva a ogni miss.
3. **Casuale**: si sceglie casualmente un'entry da sostituire. Questo metodo non segue uno schema specifico, e si basa sulla casualità.

In alcuni processori, è possibile generare un **interrupt** quando si verifica un TLB miss, consentendo al sistema operativo di partecipare alla decisione su quale entry sostituire. Alcuni processori offrono anche la possibilità all'OS di **vincolare alcune TLB entry**, garantendo che non vengano mai sostituite.

Protezione della memoria

La protezione della memoria è gestita attraverso il **registro PTLR**, che consente di limitare gli indirizzi logici disponibili per un processo. Questa restrizione offre vantaggi significativi, quali una *riduzione delle dimensioni della tabella delle pagine* (il registro PTLR consente di tagliare la tabella, ottimizzando l'uso della memoria e semplificando la gestione degli indirizzi logici).

Un'alternativa a questo approccio è l'**uso di un bit di validità in ogni entry della tabella delle pagine**. Questo bit indica se l'entry è valida, cioè se la pagina è effettivamente mappata su un frame. I vantaggi includono una *maggiore flessibilità* (si può creare intervalli di indirizzi logici inaccessibili in qualsiasi punto dello spazio di indirizzi logici).

Altri bit presenti nella tabella consentono di specificare ulteriori attributi del frame associato, come:

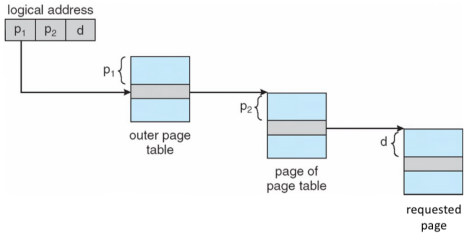
- **Read-only o read-write**: attraverso un bit di protezione, si può definire se la pagina è accessibile in modalità di sola lettura o di lettura/scrittura
- **Eseguibile o meno**: un bit di esecuzione indica se la pagina può essere eseguita come codice eseguibile o meno

Strutture delle tabelle

Quando ci confrontiamo con gli spazi di indirizzamento virtuali di dimensioni considerevoli, le tabelle delle pagine si trovano ad espandersi notevolmente.
Se le pagine hanno una dimensione di 2^n, e lo spazio di indirizzamento virtuale è di 2^m, il totale delle pagine sarà pari a 2^{m-n}.
Facendo l'ipotesi che ogni entry occupy un numero *e* di byte, la dimensione complessiva della tabella delle pagine sarà di $e * 2^{m-n}$.

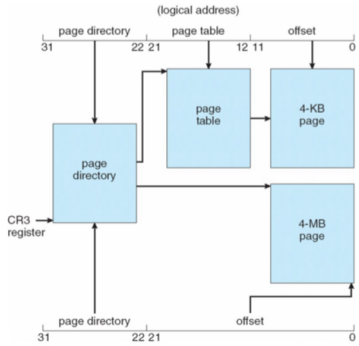
Per mitigare a tabelle troppo grandi, si usano varie soluzioni per strutturare delle tabelle in maniera che siano sufficientemente efficienti.

Tabelle delle pagine gerarchiche



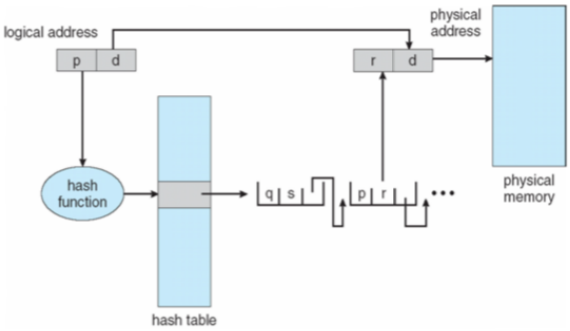
L'idea chiave è la **paginazione della tabella delle pagine stessa**, realizzata attraverso un modello a due livelli.

Ciascun numero di pagina è suddiviso ulteriormente in un **numero di pagina e offset**; questi dettagli sono utilizzati per recuperare, da una tabella esterna, *l'indirizzo della pagina della tabella delle pagine*. Quest'ultima, a sua volta, è fondamentale per la costruzione dell'indirizzo fisico completo.



Un beneficio significativo è la capacità di supportare contemporaneamente pagine di dimensioni diverse; basta marcare un'entry nella tabella delle pagine esterne perché sia considerata un'entry di ultimo livello.
Uno svantaggio principale è l'**aumento del numero di accessi in memoria** necessari per recuperare dati o istruzioni nel caso pessimo.

Tabelle delle pagine di tipo hash

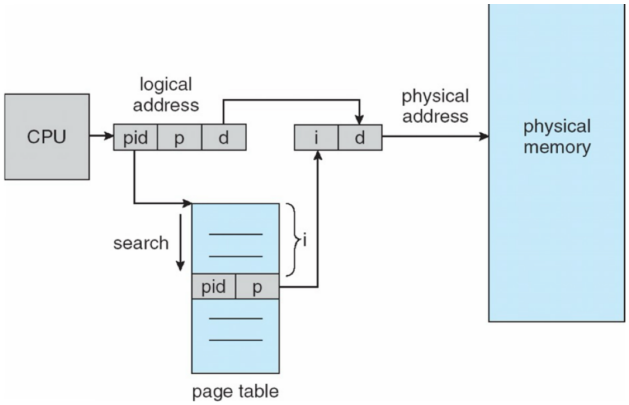


In questo approccio, viene applicata una **funzione hash** al numero di pagina. Ogni entry della tabella contiene una lista concatenata di elementi, progettata per gestire eventuali collisioni che potrebbero verificarsi.

Questo modello offre una prospettiva interessante, introducendo una struttura dinamica che si adatta alle esigenze del sistema. La funzione contribuisce alla distribuzione efficiente delle pagine, mentre la gestione delle collisioni attraverso liste concatenate rappresenta un modo flessibile di gestire situazioni complesse. Come sempre, va considerato il trade-off tra complessità ed efficienza.

Tabelle delle pagine invertite

Si adotta **un'entry per ciascun frame** anziché per ogni pagina, seguendo il tracciamento dei frame invece delle pagine. Ogni entry riporta il numero di pagina del frame corrispondente, insieme ad altri informazioni tra cui l'**ASID** (Address Space ID), che è essenziale.



Con questo approccio è possibile usare una sola tabella per tutti i processi; se lo spazio di indirizzi fisici è più limitato rispetto allo spazio virtuale, si ottiene un ulteriore risparmio. Tuttavia, ci sono anche svantaggi, come un **aumento del tempo di accesso** dovuto alla necessità di effettuare una ricerca per individuare l'entry: questo inconveniente può essere mitigato con l'implementazione di una tabella hash, e l'uso di un TLB. Inoltre, non è possibile condividere pagine tra processi diversi.

Paginazione su richiesta

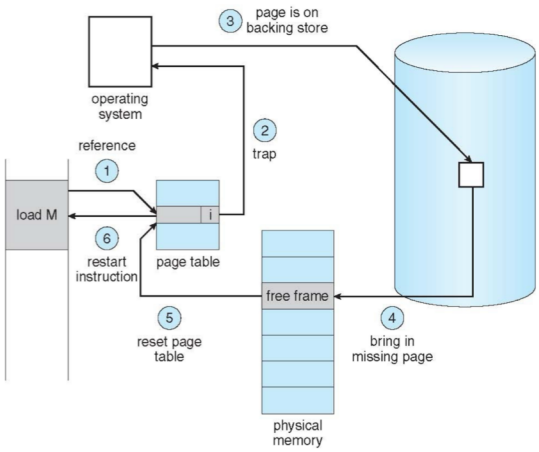
La paginazione su richiesta, basata sull'hardware di paginazione, opera secondo un principio chiave: non si porta l'intero processo in memoria alla creazione, ma solo le pagine che vengono effettivamente utilizzate durante l'esecuzione del programma.
Con questo, una *pagina viene caricata in memoria solo quando viene richiesta durante l'esecuzione* del programma, mentre il resto del processo risiede nella backing store. Questa metodologia è simile allo swapping con paginazione, ma si differenzia poiché individua predittivamente le pagine da scaricare o caricare in base all'uso.

L'hardware di paginazione deve fornire il supporto necessario per implementare la paginazione su richiesta. La tabella delle pagine deve includere un **bit di validità** per indicare se una determinata pagina è valida o assente dalla memoria.

Quando il programma tenta di accedere a una pagina con il bit di validità impostato a zero, la MMU genera un'interruzione di page fault. A questo punto, il sistema operativo gestisce il page fault, e se è stato generato perché la pagina è assente dalla memoria, la porta in memoria.
L'hardware deve consentire inoltre la **riesecuzione trasparente dell'istruzione interrotta da un page fault**, garantendo così una gestione fluida dell'evento. Infine, è necessaria un'area adeguata (**swap space**) nella backing store, per memorizzare le pagine fuori dalla memoria.

Gestione di un page fault

Inizialmente, il sistema operativo deve determinare se **la pagina che ha generato il fault è non valida o assente dalla memoria**.



Se non è valida, l'OS può decidere di abortire il processo coinvolto. Nel secondo caso, se la pagina è effettivamente assente dalla memoria, il sistema operativo cerca un **frame libero**. Una volta individuato, viene schedulata l'operazione di caricamento delle pagine dalla backing store (con un possibile cambio di contesto).
Al termine dell'operazione, la tabella viene **aggiornata con il frame caricato**, e **viene impostato il bit di validità**. L'OS ritorna dall'interruzione di page fault, consentendo al processore di rieseguire l'istruzione precedentemente interrotta.

Modello di località

La paginazione su richiesta si rivela efficiente quando i page fault sono **limitati** rispetto alle istruzioni eseguite, un aspetto spesso garantito dal modello di località.
Questo modello suggerisce che, in un dato momento, **un processo si concentra su una specifica località**, con i processi che nel tempo si spostano tra diverse località.
La possibilità di mantenere in memoria la località attuale di un processo contribuisce a minimizzare i page fault, concentrati nei momenti in cui avviene la migrazione di località.

Gestione dei frame liberi

I frame liberi sono gestiti tramite una lista mantenuta dal sistema operativo. Prima dell'assegnazione a un processo, **i frame vengono azzerati per evitare potenziali falle di informazioni**; la lista di frame liberi si riduce man mano che i processi richiedono frame, e se raggiunge una dimensione minima o scende a zero, è necessario riempirla nuovamente.

Prestazioni

Per valutare le prestazioni, vengono definite costanti come **p** (probabilità di un page fault), **m** (tempo medio di accesso alla memoria), e **pf** (tempo medio di gestione del page fault).

Il tempo medio di accesso effettivo (EAT) è calcolato come:

$$\mathrm{EAT} = (1-\mathrm{p})\ \mathrm{ma} + \mathrm{ppf}$$

Dove pf è dominato dal tempo di caricamento pagina dalla backing store.

Swapping

Quando la quantità di memoria è limitata rispetto al numero di processi, solo un numero limitato di processi può essere eseguito contemporaneamente; una tecnica che consente di gestire questo è lo **swapping**.

L'idea principale è di spostare temporaneamente l'immagine di un processo, che può essere in stato di attesa o pronto, dalla memoria centrale a una memoria secondaria. Questo processo di sospensione consente di fare spazio in memoria per altri processi, permettendo loro di essere eseguiti.

Swapping standard

L'intero processo viene spostato dalla memoria centrale alla memoria secondaria (**swap out).** Questo approccio richiede anche il trasferimento di tutte le strutture dati del sistema operativo associate al processo e ai thread: questa pratica è considerata molto onerosa.

Swapping con paginazione

Sposta solo un sottoinsieme delle pagine di un processo dalla memoria centrale alla memoria secondaria, finchè non si libera abbastanza spazio per caricare l'immagine di un nuovo processo: è notevolmente meno oneroso.
Le operazioni di scaricamento e caricamento di pagine dalla memoria centrale sono rispettivamente denominate **page out** e **page in**.

Grado di multiprogrammazione

Il grado di multiprogrammazione è definito come il **numero massimo di processi che lo scheduler della CPU può far avanzare nell'esecuzione**, considerando quelli pronti, in esecuzione e in attesa.
Questo grado svolge un ruolo cruciale nella gestione dell'occupazione della memoria fisica da parte dei processi. Senza l'uso di memoria virtuale, il grado di multiprogrammazione è identico al numero di immagini di processo completamente presenti in memoria.

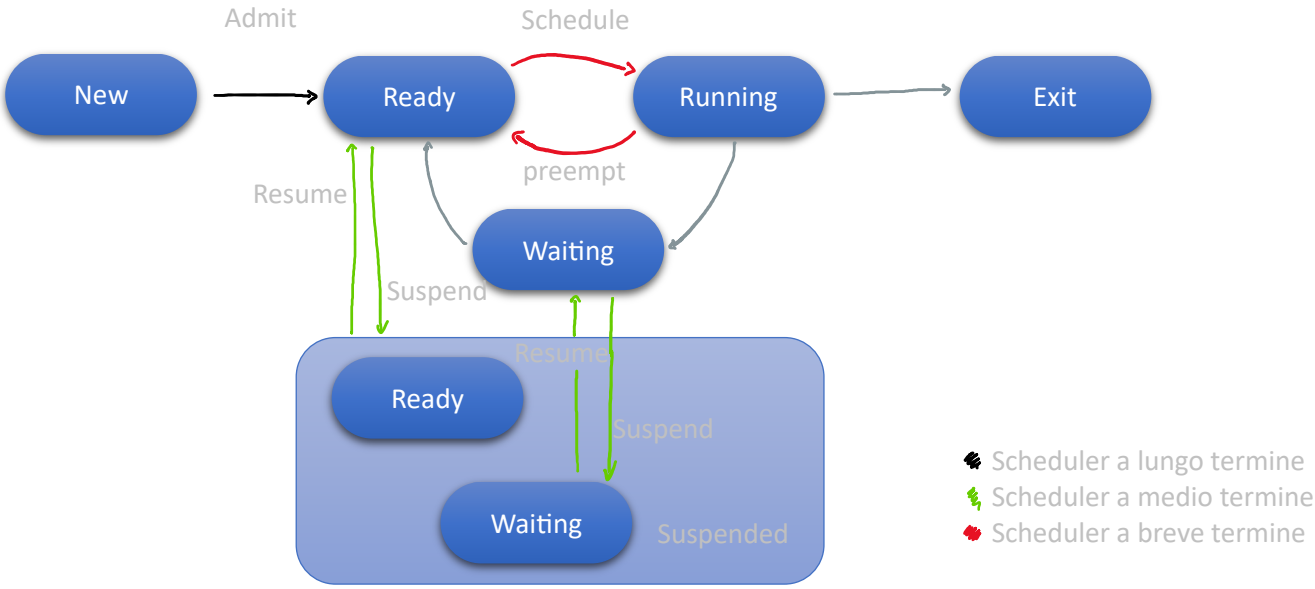
In un contesto più ampio, i processi in memoria includono quelli *pronti*, *in esecuzione* e *in attesa*, mentre quelli fuori memoria sono gli altri processi. Il grado di multiprogrammazione è essenzialmente il numero totale di processi in memoria.

Esiste una relazione tra il grado di multiprogrammazione e l'utilizzazione delle CPU. Se il grado è **basso**, l'utilizzo tende ad essere basso, poiché la mancanza di processi pronti impedisce di mantenere occupate le CPU quando alcuni processi sono in attesa. All'**aumentare** del grado, cresce il numero medio di processi pronti, portando a un aumento dell'utilizzo delle CPU.

Controllo e scheduler

Il controllo coinvolge diversi scheduler:

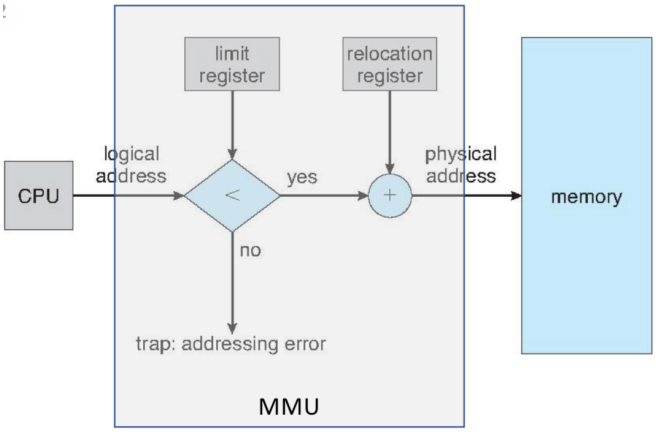
- **Lo scheduler a lungo termine** decide se ammettere un nuovo processo nella ready queue dopo la sua creazione, influenzando così il grado di multiprogrammazione a lungo termine
- **Lo scheduler a medio termine** sospende o riprende un processo per il suo swapping, modificando il grado di multiprogrammazione a medio termine
- **Lo scheduler a breve termine** (della CPU) non ha un impatto diretto sul grado di multiprogrammazione.



Indirizzi logici e fisici/MMU

Quando un programma viene eseguito, la CPU genera gli **indirizzi logici**; tuttavia, questi indirizzi logici devono essere tradotti in indirizzi fisici per interagire con la memoria centrale; questa traduzione avviene durante l'esecuzione del programma.

La chiave di questa conversione risiede nella **Memory Management Unit** (MMU), un componente hardware dedicato che entra in azione solo in modalità utente. Quando il sistema operativo opera in modalità kernel, gli indirizzi generati direttamente dalla CPU diventano indirizzi fisici.



La MMU più semplice, utilizzabile con lo schema di allocazione continua, è la **MMU con registro di rilocazione**. È una variazione dello schema con registri base e limite, dove il registro base è ora chiamato *di rilocazione*. In tal modo, i programmi hanno l'illusione di avere uno spazio di indirizzamento virtuale che va dall'indirizzo 0, a un indirizzo massimo pari al valore contenuto nel registro limite.

Nel registro base viene caricato *l'indirizzo più basso dell'area contigua di memoria assegnata al processo*; nel registro limite, viene caricata la dimensione di tale area di memoria. Se l'indirizzo logico supera tale valore, viene generata un'interruzione.

Traduzione

La traduzione degli indirizzi logici avviene **dividendo l'indirizzo logico in numero di pagina (p) e offset di pagina (d).** La MMU utilizza il numero di pagina per ottenere dalla tabella delle pagine il corrispondente numero di frame, concatenando poi il numero di frame all'offset di pagina per ottenere l'indirizzo fisico.

Associazione degli indirizzi

In presenza di molti programmi in memoria, il sistema operativo di regola carica uno stesso programma, in momenti diversi, in diverse aree di memoria.

Come fa quindi un'istruzione macchina di un programma a far riferimento ad una certa locazione di memoria, se il suo indirizzo non è noto a priori ma dipende da dove il programma viene caricato? Una prima possibilità è che il compilatore produca codice indipendente dalla posizione, ossia **position-indipendent code**, ovvero codice macchina che usi solo indirizzi di memoria relativi, e che quindi funzioni correttamente in qualsiasi locazione di memoria venga caricato. Una seconda possibilità è produrre codice dipendente dalla posizione e tradurre gli indirizzi dipendenti dalla posizione negli indirizzi corretti. Questa operazione è detta **binding degli indirizzi**.

Questa operazione può essere fatta in tre momenti diversi:

- **In compilazione**: il linker, a partire dall'indirizzo di caricamento, effettua il binding e genera codice assoluto
    - È una soluzione semplice, ma se cambia l'indirizzo di caricamento, il codice va ricompilato
- **In caricamento**: il linker genera codice rilevabile e il loader, a partire dall'indirizzo di caricamento, effettua il binding al momento del caricamento in memoria del codice
    - Permette di variare liberamente l'indirizzo di caricamento da esecuzione ad esecuzione, ma è una soluzione lenta che non permette di ritoccare l'immagine di un processo durante la sua esecuzione
- **In esecuzione**: il binding viene effettuato dall'hardware dinamicamente mentre il codice viene eseguito

Il binding degli indirizzi in fase di caricamento è una soluzione più rapida dell'associazione in fase di compilazione e dell'associazione in fase di esecuzione.

Memoria virtuale

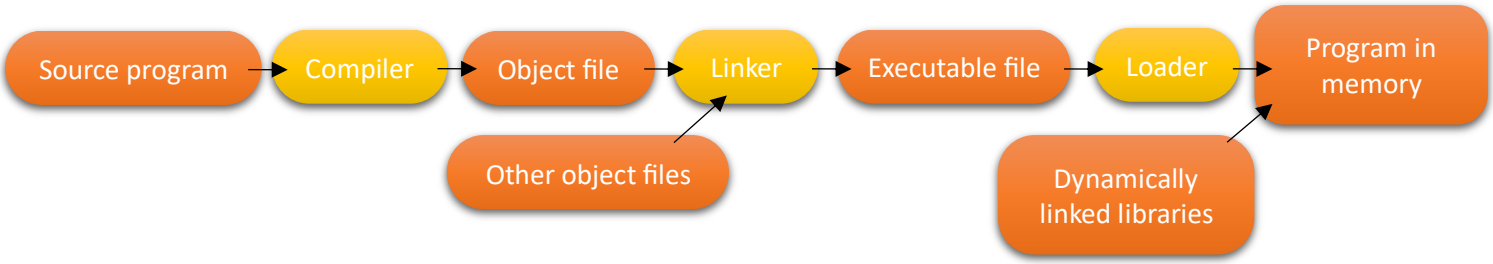
La necessità di utilizzare la memoria virtuale nasce dal fatto che, in pratica, un processo non deve avere la sua immagine completa in memoria per essere eseguito. Questo perché, raramente, tutta la memoria di un processo viene utilizzata integralmente e allo stesso istante.

Emergono diversi vantaggi con questo approccio: un processo potrebbe avere un'immagine più grande della memoria fisica disponibile (diminuendo il bisogno di memoria fisica per ogni processo, è possibile aumentare il numero di processi incrementando il grado, e anche l'uso del processore senza aumentare i tempi di risposta o di turnaround); un maggior numero di processi in memoria può anche significare una *riduzione delle operazioni di I/O necessarie per lo swapping*.

La memoria virtuale implica una **completa separazione tra la memoria logica e quella fisica** di un programma; ciò consente a un processo di essere eseguito anche se solo una parte di esso è presente in memoria fisica.
Lo spazio di indirizzamento virtuale può essere notevolmente più grande dello spazio di indirizzamento fisico.

Loader e linker

Un programma sorgente è compilato in un file oggetto, che deve poter essere caricato a partire da qualsiasi locazione di memoria fisica, ovvero un **file oggetto rilocabile**.
I **linker** combinano più file oggetto per formare un file eseguibile.
I **loader** si occupano di caricare in memoria i file eseguibili nel momento in cui devono essere eseguiti.
Inoltre, i loader effettuano il linking delle librerie dinamiche.



Librerie dinamiche

Nei sistemi operativi odierni, non tutto il linking viene fatto a compile time: le librerie dinamiche vengono collegate quando il programma è caricato o durante l'esecuzione del programma stesso. Il vantaggio delle librerie dinamiche è che queste possono essere **condivise tra diversi programmi**, riducendo le dimensioni dei programmi stessi e risparmiando memoria.

Mappare dei file

Per mappare i file, dobbiamo effettuare un'invocazione di API:

```
int fd = fopen("/foo/baz", 0_RDWR); mmap(0xa0000000, 4096, PROT_READ|PROT_WRITE,
                                    MAP_PRIVATE, 0);
```

- **fopen**: apre un file specificato dal percorso con la modalità di apertura **0_RDWR** (apertura in lettura e scrittura).
- **mmap:** mappa file o dispositivi nei processi; mappa 4096 byte di memoria a partire dall'indirizzo **0xa0000000** con diritti di lettura e scrittura (PROT_READ e PROT_WRITE). Il flag **MAP_PRIVATE** indica che è una mappatura privata (non condivisa), e l'ultimo parametro è 0, che indica l'offset nel file.

Le API POSIX

Normalmente, in un ambiente di sviluppo, non è necessario utilizzare direttamente le API per gestire lo stack e l'heap, ovvero per alcuni motivi precisi:

- **Gestione automatica dello stack**: lo stack di un processo è solitamente gestito automaticamente dall'OS, quindi non è necessario specificare alcunchè
- **Gestione dello heap tramite il supporto runtime o librerie del linguaggio**: lo heap è solitamente gestito dal supporto runtime del linguaggio di programmazione utilizzato, o dalle librerie standard (es. *malloc* in C). Questi strumenti usano le API di sistema per espandere o ridurre l'heap in base alle esigenze del processo.

Poiché allora è importante imparare le API per gestire la memoria?

- **Permessi speciali**: le API di gestione della memoria consentono di assegnare permessi speciali a ragioni specifiche di memoria. È possibile per esempio *dichiarare una regione di memoria come sola lettura o eseguibile*.
- **Sviluppo di strumenti personalizzate**: se si vuole sviluppare allocators di memoria, compilatori just-in-time, etc.
- **Gestione avanzata della memoria**: le API di gestione della memoria consentono di sfruttare funzionalità avanzate, come l'utilizzo di file mappati in memoria e la memoria condivisa.

Mmap

Nel contesto dei sistemi Unix legacy, esistono API come **brk** e **sbrk**, che permettono di cambiare la dimensione del segmento dati, che include regioni di dati e heap. Queste API però sono state deprecate, e incompatibili con l'API **mmap**, che è diventata uno standard più moderno e potente. Mmap offre la possibilità di mappare regioni di memoria su memoria centrale, file o memoria condivisa.

```
void *mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

- `addr`: indirizzo virtuale da cui iniziare il mapping
- `length`: lunghezza della regione di memoria da mappare. Normalmente allineata su multipli di 4K (4096 byte) su processori x86.
- `prot`: indica i permessi associati a questa regione di memoria, come *lettura, scrittura, esecuzione, o nessun accesso.*
- `flags`: specifica opzioni di mapping, ad esempio se la memoria deve essere allocata a un indirizzo specifico o altre opzioni
- `fd`: questo parametro è un descrittore di file, o in caso di memoria anonima, possiamo usare MAP_ANONYMOUS come flag.
- `offset`: usato in combinazione con `fd` per indicare la posizione all'interno del file da mappare.
- **Return** indirizzo della memoria mappata, oppure la costante `MAP_FAILED` se la chiamata fallisce

Abbiamo qualche esempio:

Per mappare **4KB** di memoria a partire dall'indirizzo virtuale **0xa0000000**

```
void *ptr = mmap(0xa000000, 4096, PROT_READ|PROT_WRITE, MAP_ANONYMOUS, 0, 0);
```

- MAP_ANONYMOUS specifica che la memoria mappata non è associata a un file
- La regione di memoria è resa leggibile e scrivibile
- La funzione restituisce un puntatore a questa regione di memoria mappata

Per mappare a partire dall'indirizzo virtuale **0xb0000000 8192 bytes** del file **/usr/foo** a partire dal **byte 100**

```
int fd = open("/usr/foo", 0_RDWR);
```

- Apre un file situato in `/usr/foo` in modalità lettura e scrittura
- Restituisce un descrittore di file che rappresenta il file appena aperto

```
void *ptr = mmap(0xb0000000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 100);
```

- La regione di memoria è resa leggibile e scrivibile
- Specifica che la memoria mappata è privata, il che significa che le modifiche apportate qua non verranno riflesse nel file originale

- `fd` è il descrittore del file restituito dalla chiamata open. La memoria verrà mappata al contenuto del file specificato

Per sincronizzare le modifiche in memoria con il file

```
int ok = msync(0xb0000000, 8192, MS_SYNC|MS_INVALIDATE);
```

- viene specificato l'indirizzo iniziale della regione di memoria da sincronizzare, e la dimensione della regione da sincronizzare
- `MS_SYNC` indica che la sincronizzazione deve essere aspettata e completa, mentre `MS_INVALIDATE` indica che la memoria mappata dovrebbe essere invalidata dopo la sincronizzazione

Il file system

Il file system è uno degli aspetti fondamentali dei sistemi operativi moderni. È responsabile della **gestione e dell'organizzazione dei dati su supporti di archiviazione**, come dischi rigidi, unità flash e altro. Grazie al file system, i dati possono essere memorizzati, recuperati, organizzati e protetti in modo efficace.

Un file system è costituito da un insieme di file, e una struttura delle directory, che organizza i file.

I file

Un file è un'unità di memorizzazione logica che consiste in **dati correlati memorizzati in memoria secondaria e associati a un nome**. Un file può essere suddiviso in una sequenza di record, righe, bit, byte, con il significato dei dati definito dal creatore del file.
I file hanno vari attributi:

- **Nome**: nome del file, unico attributo leggibile dall'utente
- **Identificatore**: etichetta unica fornita dal sistema di file per identificare il file
- **Tipo**: indica il tipo di dati contenuti nel file. Non tutti i sistemi ce l'hanno, e preferiscono usare invece le estensioni per esempio
- **Locazione**: specifica il dispositivo di memoria secondaria e la posizione sul dispositivo in cui i dati del file sono memorizzati.
- **Dimensione**: indica la dimensione del file in byte, parole, record, ecc….
- **Protezione**: fornisce informazioni sul controllo degli accessi
- **Ora, data e utente**: indicano quando il file è stato creato, letto o modificato e chi ha eseguito queste azioni
- **Attributi estesi**: questi attributi possono includere il checksum, la codifica caratteri, applicazioni correlate, ecc….

Abbiamo anche varie operazioni dei processi su file:

- **Creazione**: riservato spazio nel filesystem per i dati, e viene aggiunto un elemento nella directory
- **Apertura**: effettuata prima dell'utilizzo di un file
- **Lettura**: a partire dalla posizione determinata da un puntatore di lettura
- **Scrittura**: a partire dalla posizione determinata da un puntatore di scrittura
- **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

I file possono essere di tipo **dati** o di **programmi**, e il sistema può essere più o meno consapevole del tipo di file, ma deve almeno riconoscere il tipo di file eseguibile. Per fare questo, i sistemi adottano varie strategie:

- Schema del nome (nome .estensione)
- **Attributi nei file** (in macOS viene registrato il programma che ha creato il file)
- **Magic number** all'inizio del file come primi byte: Unix usa **shebang**, DOS/Win16 usa **MZ**, e NT/Win32 usa **PE**.

Inoltre, un file può avere varie strutture:

- **Nessuna struttura**: ad esempio, nei sistemi Unix, un file è semplicemente 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**, usate soprattutto per i file eseguibili: questi sono **PE** in Win32, **a.out**/**ELF** in Unix, **Mach-O** in macOS).

Pià il sistema operativo supporta direttamente diverse strutture file, più diventa complesso; se l'OS è troppo rigido sulle possibili strutture, però, potrebbe non supportare nuovi tipi o tipi ibridi. Per questo vengono introdotte app esterne all'OS che sono in grado di riconoscere tutte le estensioni di cui abbiamo bisogno, perché se bisognasse integrare tutte le estensioni nell'OS, bisognerebbe aggiornare tutto l'OS ogni volta che si vuole aggiornare un'estensione.

Lock dei file

I file possono essere aperti da più processi contemporaneamente, il che può portare a problemi di concorrenza. Per affrontare questi problemi, alcuni OS permettono di applicare **lock** ai file o alle loro porzioni. I tipi principali sono due:

- **Lock condiviso**: detto anche lock di lettura, permette di leggere da più processi contemporaneamente. Proibisce l'acquisizione di un lock esclusivo
- **Lock esclusivo**: detto anche lock di scrittura, consentito solo per un processo alla volta per la scrittura, impendendo l'acquisizione id un lock condiviso
- **Lock obbligatori (mandatory)**: l'OS impedisce l'accesso al file ai processi che non detengono il lock. Usato da Windows
- **Lock consultivi (advisory)**: l'OS offer il lock, ma non regola l'accesso al file: sono i processi che devono evitare di accedere al file se non hanno il lock. Usato da Unix

Metodi di accesso

Accesso sequenziale

Uno dei metodi per accedere a un file è il metodo di accesso sequenziale. In questo metodo, **il file è una sequenza di record a lunghezza fissa**, e succede il seguente:

1. Si inizia la lettura dal primo record o dal punto iniziale del file
2. I record vengono letti uno dopo l'altro nell'ordine in cui sono memorizzati
3. Dopo la lettura di un record, il cursore di lettura si sposta al successivo
4. Questo processo continua fino a quando tutti i record desiderati sono stati letti o fino a quando si raggiunge la fine del file.

Ovviamente, esiste anche la scrittura sequenziale che si avvale di un procedimento simile:

1. Si inizia la scrittura dal primo record o dal punto iniziale del file
2. I nuovi record vengono scritti uno dopo l'altro nell'ordine in cui sono creati
3. Dopo la scrittura di un record, il cursore di scrittura si sposta al successivo
4. Questo processo continua fino a quando tutti i nuovi record sono stati scritti.

L'accesso sequenziale è efficiente per la lettura o la scrittura di file in modo sequenziale; però, non è adatto per cercare o modificare dati in modo casuale, perché richiede la lettura o la scrittura di tutti i record precedenti per accedere a uno specifico record.

Accesso diretto

L'accesso diretto offre operazioni come `read(n)` e `write(n)`, per accedere direttamente al n-esimo record. In alternativa, è possibile utilizzare operazioni come `read_next()`, `write_next()` e `position(n)` per navigare tra i record. L'accesso diretto è utile quando è necessario **accedere a record specifici all'interno di una struttura dati**, consentendo prestazioni efficienti per l'individuazione di record specifici.

Accesso indicizzato

In determinati sistemi operativi, come quelli progettati per i mainframe IBM, i file possono essere organizzati come **sequenze di record ordinate in base a un campo chiave** specifico all'interno di ciascun record. In questi contesti, l'accesso ai dati può avvenire utilizzando la chiave come riferimento, e il sistema operativo mantiene un indice appositamente creato per velocizzare il recupero dei dati.

Un esempio di questo tipo di accesso è il sistema **ISAM** (Indexed Sequential Access Method) nei sistemi IBM, in cui l'accesso è basato sulla chiave di ricerca. Allo stesso modo, il file system **Files-11** sviluppato da Digital per il sistema operativo OpenVMS offre tutti e tre i principali tipi di accesso: diretto, sequenziale e basato su chiave. Questo approccio ottimizza il recupero dei dati in base ai requisiti specifici del sistema e del tipo di dati memorizzati.

Le API POSIX

- Per **creare** un file chiamato "foo.txt" con diritti di lettura e scrittura per il proprietario, e di sola lettura per i membri del gruppo e altri utenti: `int fd = create("foo.txt", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);` o `int fd = open("foo.txt"), 0_WRONLY | 0_CREAT | 0_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);`
- Per **aprire** un file chiamato "foo.txt" in sola lettura: `int fd = open("foo.txt", 0_RDONLY);`
- Per **leggere** da un file: `char buf[NBYTES]; int tot_read = read(fd, but, NBYTES);`
- Per **scrivere** su un file: `char buf[] = "Hello"; int tot_written = write(fd, but, 5);`
- Per **spostarsi** all'interno di un file, specifica con un numero di quanti byte ti vuoi spostare: `off_t offset = sleek(fd, 10, SEEK_CUR);`
- Per **chiudere** un file: `int ok = close(fd);`
- Per **eliminare** un file chiamato "foo.txt": `int ok = unlink("foo.txt");`
- Per **truncare** un file chiamato "foo.txt" della lunghezza desiderata **n**: `int ok = truncate("foo.txt", n);`
- Per impostare un **lock** su un file (tipo consentivo). Per esempio, si può impostare un lock in lettura condiviso dalla posizione corrente + 10 byte fino alla fine del file:

```
struct flock fl;
fl.l_type = F_RDLCK; //lock di lettura
fl.l_whence = SEEK_CUR; //dalla posizione corrente
fl.l_start = 10; // + 10 byte
fl.l_len = 0; //fino alla fine del file
int ok = fcntl(fd, F_SETLK, &fl); //fallisce
```

Directory

Una directory, nel contesto dei sistemi operativi, funge principalmente da catalogo che elenca i file presenti all'interno del file system. Può essere quindi considerata una **tabella che associa i nomi dei file ai dati e ai metadati contenuti in ciascun file**. Tanto i file quanto le directory risiedono fisicamente sul disco, e almeno una directory deve essere presente nel file system per consentire la gestione e il ritrovamento dei file.

I processi operano sulle directory svolgendo una serie di azioni chiave:

- **Creazione** di un file all'interno di una directory
- **Cancellazione** di un file da una directory
- **Ridenominazione** di un file o il suo spostamento da una directory all'altra
- **Elenco** dei file contenuti in una directory
- **Ricerca** di un file, basata sul nome o su uno schema di possibili nomi

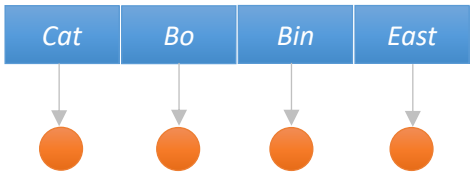
- **Traversare** la struttura del file system, ad esempio, per scopi di backup

Strutture

Le directory possono assumere diverse strutture per organizzare i file.

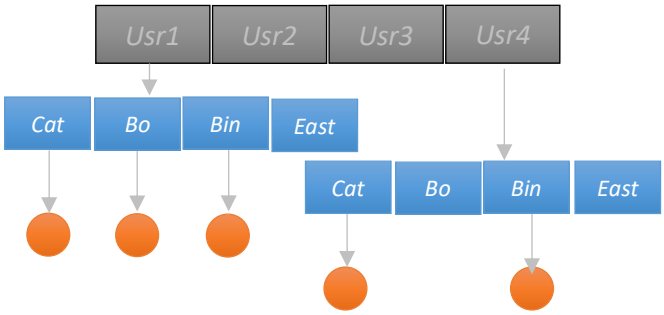
Strutture ad un livello

La struttura delle directory ad un livello prevede l'utilizzo di una sola directory per tutti i file. Questo approccio offre **semplicità nella gestione**, ma può comportare difficoltà nella denominazione dei file quando ce ne sono molti. Inoltre, raggruppare file di utenti diversi diventa problematico.



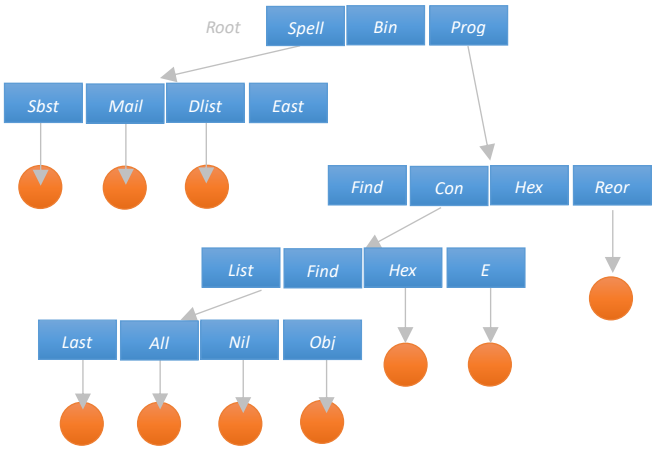
Strutture a due livelli

La struttura delle directory a due livelli implica una **directory principale** che contiene **sottodirectory**, ognuna destinata a un utente. Questo schema consente una migliore organizzazione dei file, ma richiede l'uso di **nomi di percorso** (path name) per identificare un file in modo univoco. Questi percorsi possono variare a seconda del sistema operativo, come /user2/data in sistemi Unix-like, e \user2\data in sistemi Windows-like.
Inoltre, i file di sistema sono spesso posizionati in directory speciali, e il sistema deve avere un percorso di ricerca definito per trovarli.



Strutture ad albero

La struttura delle directory ad albero offre una struttura gerarchica in cui ogni directory può contenere **file o altre directory**. Questo modello consente agli utenti di raggruppare i propri file in modo più logico.
Per semplificare l'accesso, ogni programma ha una directory corrente dalla quale possono essere specificati path relativi. Ad esempio, se la directory corrente è */programs/mail*, un path relativo potrebbe essere *prt/first*.
La cancellazione di una directory comporta la rimozione di tutto il suo contenuto, a meno che sia vuota.



Strutture a grafico aciclico

Questo tipo di struttura consente l'**aliasing**, ossia l'uso di più nomi per lo stesso file o directory. Questo può portare a domande su cosa accada quando un file o una directory con alias viene cancellato. Ciò può essere gestito attraverso **hard links**, che implicano la duplicazione delle voci di directory e l'uso di Un contatore dei riferimenti. Quando questo contatore scende a zero, il file può essere rimosso.
Al contrario, i **link simbolici** fanno riferimento a un percorso assoluto e possono diventare *abbandonati* se il percorso sottostante viene cancellato.

Strutture a grafico generico

Questo tipo di struttura è la più flessibile, consentendo hard link persino a directory in livelli superiori, comprese quelle che contengono in modo ricorsivo il link stesso. Tuttavia, la gestione dei file non più

referenziati richiede un **algoritmo di garbage collection** complesso; l'attraversamento del file system è reso più complicato in questa struttura.

Le API POSIX

- Per creare una nuova directory, con diritti di lettura, scrittura e ricerca per il proprietario, e di sola lettura per i membri del gruppo e gli altri: `mkdir("/home/pietro/newdir", S_IRWXU|S_IRGRP|S_IROTH);`
- Per eliminare una directory solo se vuota: `rmdir("/home/pietro/newdir");`
- Per visitare il contenuto di una directory:

```
#include <dirent.h>
DIR *dir;
struct dirent *dp;
…
if ((dir = opendir(".")) == NULL) {
     perror("Cannot open . ");
     exit(1);
}
while ((dp = readdir(dir)) != NULL) { … }
```

- Per cambiare il nome, o spostare, un file o una directory si può usare l'API **link**, che aggiunge una voce di directory (hard link) per un file esistente, insieme all'API **unlink**, che cancella una voce di directory. Quando il link count di un file arriva a zero, il file viene cancellato.
    - Per spostare il file **/foo/bar** nella directory **/home/pietro**, e allo stesso tempo cambiare il suo nome in bar:
        - `int ok1 = link("/foo/bar", "/home/pietro/baz");`
        - `int ok2 = unlink("/foo/bar");`
    - Per creare un link simbolico si può usare l'API **symlink**; i link simbolici non vengono aggiunti al link count: `int ok = symlink("/home/pietro/baz", "home/pietro/alias");`