Sistemi Operativi - Appunti presi a lezione

Cos'è un sistema operativo?

Software per gestire le risorse del computer, per gli utenti e le applicazioni.

Es. `printf` ha tutta una struttura a livello di sistema operativo

Utile per il programmatore per non "reinventare la ruota" e per concentrarsi sul proprio obiettivo.

Si **interfaccia con l'hardware e le applicazioni**. **Enormemente complesso**, diviso in una serie di problemi semplici: livelli, **buccia di cipolla**. **Ogni livello astrae un po' il livello precedente**, librerie di astrazione: `printf` non è una funz del sistema operativo, è una funzione di libreria scritta in C ma è anche software a corredo del linguaggio e legato al sistema operativo. La `printf` **traduce le richieste in azioni del sistema operativo**.

Un sistema operativo **agisce come**:

- **arbitro**: **alloca risorse** tra utenti e applicazioni, **isola** differenti utenti e applicazioni, **comunicazioni** tra utenti e applicazioni
- **illusionista**: **ogni applicazione crede di avere l'intera macchina per sè**, numero infinito di processori, (quasi) infinita memoria, storage affidabile, trasporto di rete affidabile, fa sembrare risorse più belle e ricche
- **colla**: fornisce **servizi comuni e standard** alle applicazioni, semplifica lo sviluppo delle applicazioni, librerie, widget della UI...

Pattern di design

Cloud computing

- Arbitro: come allocare risorse tra applicazioni competenti in cloud?
- Illusionista: come isolare app dalla continua evoluzione del cloud?
- Colla: come fornire tutti i supporti per usare tutto, i sistemi per osservare ecc...?

Web Services

- tanti utenti contemporanei da gestire molte richieste di dati e calcoli, come motori di ricerca, una richiesta può richiedere tempi di calcolo molto lunghi
- il server pulisce la cache per velocizzare, che è condivisa tra gli utenti, serve sincronia
- i siti vanno aggiornati, come mantenere consistenza?
- Client e server a velocità differenti, vanno gestite
- L'hw potrebbe diventare datato, come evitare di riscrivere il codice?

Altri aspetti dei SO

Affidabilità, sicurezza, portabilità...
prestazioni, throughput, overhead, fairness, prevedibilità

Architettura tipica

CPU 1 o più, memoria principale, set di dispositivi, interrupt e accesso diretto alla memoria DMA.
Interrupt interrompe i processi e risolve la causa dell'interrupt come dare l'accesso al processore.

CPU

Registri generali usati per calcoli generici e operazioni all'interno del programma, registri di stato (PC program counter, SP stack pointer. PSR program status register parola di stato vettore di bit info relative allo stato di esecuzione del processore, bit che segnalano overflow, risultato ok, divisione per 0... più bit riguardanti stato interno del processore, stato del programma e stato del processore quando esegue)

PSR: Condition code (status dell'ultima operazione: overflow, divisione per 0), CPU mode (kernel, user), Interrupt enable bit

Ciclo Fetch-Execution

Se interrupt e interrupt abilitati: gestisci
Altrimenti: carica istruzione con indirizzo PC, esegui, PC=PC+4 (se istruzione occupa 4 byte)

Multitasking vs Singletasking
Nei decenni passati, il **multitasking** si è rivelato soddisfacente. Ora, però, la sempre maggiore necessità di **interazione utente** durante l'esecuzione dei programmi ha fatto nascere una seconda necessità: se prima era necessario **ottimizzare il tempo processore** ora va anche **ottimizzato il tempo utente**. Dato che nei sistemi multitasking si **attende l'interruzione spontanea** del programma, si poneva il rischio che un programma monopolizzasse per troppo tempo il processore impedendo ad altri di elaborare input utente. Si è così reso necessario un approccio diverso: il **time-sharing**.

Time Sharing Operating Systems
In questo modello il processore è assegnato ad un altro programma anche se il programma in esecuzione non aveva finito di elaborare, cioè **viene assegnato un quanto di tempo uguale per tutti**. Quando un programma è stato elaborato dal processore per quel **quanto di tempo**, viene "scaricato" ed il processore viene automaticamente assegnato ad un altro programma al quale si applica lo stesso criterio. Quando tutti hanno avuto il processore per il quanto di tempo si **riparte** dal primo.
Il time sharing è completamente **trasparente** ai programmi: per ogni programma il processore viene visto come se fosse un processore **proprio del programma** ma più lento dell'originale. Con questo metodo, quindi, il sistema complessivo diventa **più lento** perché deve gestire tutte le interruzioni, che aggiungono lavoro al sistema operativo. Questa perdita di tempo processore è contrapposta ad un grosso **guadagno in interattività per l'utente**, inoltre è una spesa sensata perché i processori sono molto più economici e gli utente perdono **molto meno tempo**.

Oggi i computer sono molto economici e troviamo processori **ovunque**: smartphone, sistemi embedded, server, laptop, tablet...
Mentre prima si **progettavano sistemi operativi per macchine tutte simili fra loro**, oggi durante la progettazione va preso in considerazione la **possibilità che il sistema operativo finisca in un telefono, in un braccialetto, un portatile, un server, nel cloud** e così via, ed ovviamente **ogni dispositivo ha requisiti differenti**.
La tendenza non è di progettare sistemi operativi **ad hoc**, per ogni dispositivo, ma di **uniformare i sistemi operativi** dei vari device.
Dato che ogni device ha requisiti differenti: come assegnare le risorse? e la memoria? Tutte domande alle quali va trovata una risposta.

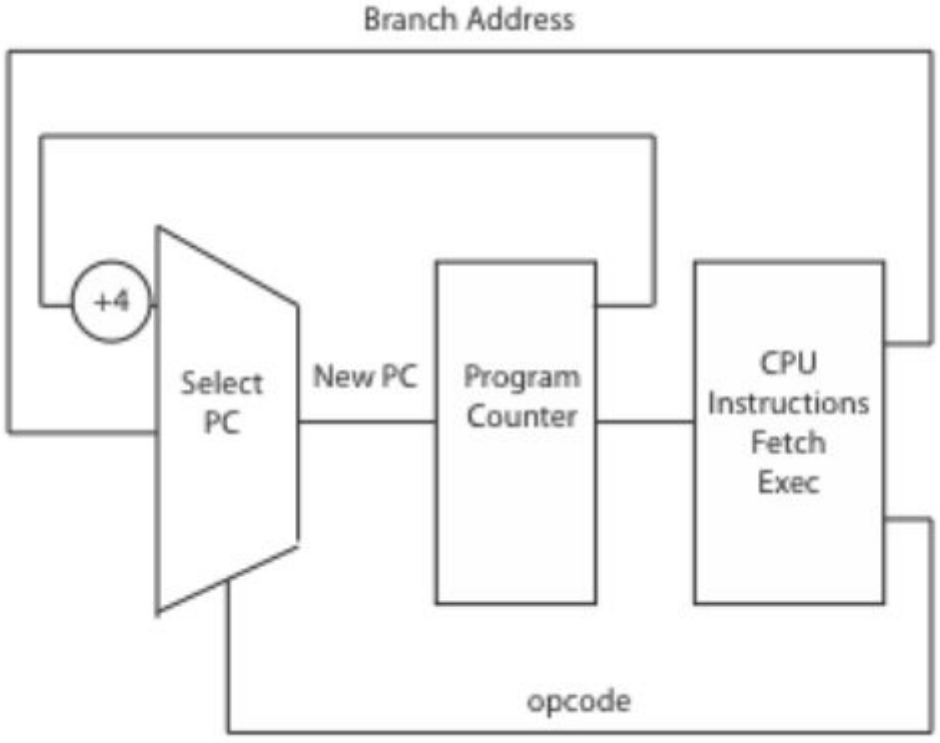
Architettura del computer
In un semplice modello, guardando l'hardware dal punto di vista di **cosa ci può offrire**, un computer è composto da:

- uno o più **processori**
- **memoria principale**, contenente i programmi, i risultati e i dati
- **dispositivi** vari, con alto grado di indipendenza e usati per inviare e ricevere informazioni
- **interruzioni**
- **accesso diretto** alla **memoria**

CPU

- Registri **generali**
- Registri **di stato**
- Registri **di controllo**:
    - **PC**, Program Counter
    - **SP**, Stack Pointer
    - **PS**, Program Status Register
      Maschera di bit con i **codici di condizione** (Overflow, risultato = 0 ecc...), **CPU Mode** e **Interrupt Enable Bit**

Il funzionamento del processore avviene attraverso un **ciclo di estrazione ed esecuzione** (**fetch-exec cycle**): per prima cosa controlla se ci sono **interruzioni pendenti** e se le **interruzioni sono abilitate**, in caso affermativo **serve l'interruzione**, altrimenti prende il PC e lo spedisce alla memoria per farsi ritornare il contenuto delle celle, cioè **carica l'istruzione** dalla memoria, poi la **esegue** ed incrementa opportunamente il PC.
L'esecuzione è **cieca**, cioè il processore **non controlla se l'operazione è valida**, sicura o che. Questo crea problemi di **sicurezza** in caso si vogliano **eseguire programmi con privilegi**: si presenta la necessità di imporre dei **vincoli**, non necessari nei sistemi single-task perché nel caso peggio il programma **si danneggiava da sè** e comunque si poteva sempre riavviare il computer. Come implementare l'esecuzione con **privilegi limitati**?
Posso eseguire l'istruzione in un **simulatore**, se l'istruzione è permessa la eseguo o altrimenti fermo il processo. Se l'hardware non è stato progettato in quest'ottica non è possibile eseguire l'istruzione direttamente sul processore avendo **garanzia di protezione**.



Questa esigenza è nata con i primi sistemi multitasking, con meccanismi **inglobati direttamente nel processore. Il** sistema viene protetto con politiche implementate sia hardware che software.

Processo
Il processo è un'**astrazione software** propria del sistema operativo, cioè una collezione di elementi astratti realizzati dal sistema operativo per permettere lo sviluppo di politiche di protezione ed usare in maniera corretta ed efficiente i meccanismi hardware.
Il processo è una **sequenza di attività** attivate da un programma ed eseguite sul processore con privilegi limitati. Tali attività sono caricate in memoria, frutto della compilazione di un programma. Comprende diversi elementi:

- **Process Control Block**, una struttura dati del sistema operativo usata per tenere traccia del processo. Ogni processo ne ha una, e sono tutte contenute nella **process table** del kernel.
  Contiene tutte le informazioni associate al processo, come il **nome** (o **PID**, che può funzionare anche da indice), i puntatori ai thread, la memoria assegnata ed altre risorse. Usando queste informazioni **attivo i meccanismi di protezione** che limitano lo spazio d'azione.

Questa astrazione è il primo elemento che serve per sviluppare le politiche di protezione, insieme alla **dual mode operation** e al **meccanismo di passaggio** tra le modalità.

- **Thread**, che è un processo più leggero, condivide alcuni elementi col processo padre ed esegue una sequenza di istruzioni di un processo. Possono esserci più thread per processo.
- **Address space**, l'insieme dei diritti, cioè la memoria che il processo può utilizzare ed altri permessi (come quali chiamate di procedura può fare, quali file accedere ecc...)

Un **programma** è statico, poiché è una **sequenza di istruzioni** ferme senza risultato, perché **rappresenta una procedura**.
Un **processo** invece è la **sequenza di attività nell'esecuzione di un programma** eseguite su una CPU e su determinati dati. Possono nascere **più processi** dallo stesso programma, eseguendo lo stesso codice ma su dati e/o in tempi diversi.

Dual-Mode Operation
User mode: modalità di esecuzione dell'istruzione con **privilegi limitati**, cioè solo quelli garantiti dal kernel.
Kernel mode: modalità di esecuzione dell'istruzione con **privilegi pieni**, quindi accesso libero a dispositivi, lettura e scrittura in qualsiasi area di memoria, spedizione e ricezione di qualsiasi pacchetto ecc....
La modalità corrente è memorizzata in un bit del **Program Status Register**.
Per garantire questa modalità il progetto dello schema del processore viene complicato. Vengono anche introdotte le **istruzioni privilegiate**, cioè istruzioni eseguibili solo in kernel mode, come ad esempio la disabilitazione delle istruzioni (perché bloccherei il timer, rendendo impossibile interrompere un programma per il quanto di tempo) o cambiare il bit di modalità user/kernel mode.
Se in user mode si esegue una istruzione privilegiata, il processore solleva un'interruzione (chiama **eccezione** in questo caso), il sistema operativo quindi analizza la situazione e tipicamente interrompe l'esecuzione con un **segmentation fault**.

Safe Control Transfer
Il meccanismo di commutazione della modalità da user a kernel e viceversa, critico perché deve essere eseguito correttamente solo in certe condizioni (non può, ad esempio, essere richiesto da un programma).

Hardware Timer
Il timer è un dispositivo hardware che periodicamente **interrompe il processore**:

- passa il controllo del processore al **gestore di interruzione del timer** del kernel (**kernel timer interrupt handler**)
- la **frequenza di interruzione** è impostata dal **kernel**, **non dal codice utente**
- le interruzioni possono essere **rinviate di poco**, comunque non dall'utente, questo è un meccanismo **cruciale** per implementare le **esclusioni mutuali**

Mode Switch
Da usermode a kernelmode

- **Interrupts**, **esterni** al codice
    - Timer
    - Dispositivi I/O (Gestite dal so)
- **Eccezioni**, funzionano come le interruzioni ma sono **interne** al codice
    - sono attivate da **comportamenti inaspettati** dei programmi o da comportamenti **dannosi** e **malevoli**
- **Chiamate di Sistema** (aka **protected procedure call**), per permettere ai processi utente di **usare risorse hardware**, il cui uso è comunque mediato dal sistema operativo in maniera controllata
    - **richiesta al kernel** di fare qualche **operazione per conto del programma** (come usare dei dispositivi per conto del programma utente)
    - sono in **numero limitato** ma **permettono uso di tutte le risorse hardware**: questo perché non si può accedere a caso al codice del sistema operativo se si vuole usare una risorsa.

Quindi l'invocazione è controllata **partendo sempre dall'inizio di ogni funzione** a disposizione dei processi utente (cioè le chiamate di sistema).
Istruzione particolare che genera interruzione per passare a kernel mode in modo protetto

Da kernelmode a usermode

- **avvio** di un **nuovo processo/thread**
    - salto alla prima istruzione nel programma/thread
- **ritorno da un interrupt**, **un'eccezione** o **una chiamata di sistema**
    - **ripresa dell'esecuzione** precedentemente sospesa
- **commutazione di contesto** tra processi/thread
    - riprendo qualche altro processo
- **user-level upcall**
    - notifica asincrona a programma utente

Comunque tutto si riduce a due meccanismi: l'**interruzione** (per passare da user a kernel) e il **return from interrupt** che è un'istruzione privilegiata (per passare da kernel a user)

Come eseguire in sicurezza gli interrupt?

Bisogna gestire un interrupt che arriva e alla fine la terminazione dell'interrupt. Per fare questo si adoperano una serie di componenti all'interno dell'architettura del sistema operativo.

Vettore di Interrupt

Si tratta di una **tabella** con i vari **indirizzi delle funzioni handler** in modo da sapere quale handler usare per ogni interruzione. La tabella è impostata dal kernel con i puntatori al codice da eseguire nei vari eventi:

- *handlerTimerInterrupt*
- *handlerDivideByZero*
- *handlerSystemCall*
- ...

Contiene anche la **parola di stato** da assegnare al Program Status Register per indicare i diritti che ogni handler possiede.
La sua inizializzazione è un'operazione delicata, perché c'è il rischio di **ricevere un'interruzione durante l'inizializzazione**: in tal caso, se l'indirizzo non è ancora stato definito, il processore **salta ad un indirizzo indefinito ed esegue ciò che trova**, avendo così un **comportamento indefinito** e un probabile crash del sistema. Per risolvere questo problema **si disabilitano interruzioni durante l'inizializzazione** del vettore di interrupt.

Kernel Interrupt Stack

Contiene il **record di attivazione dell'handler**.
Ogni processo ha un doppio stack: lo stack usato normalmente in stato utente e lo stack che viene usato in stato kernel quando arriva l'interruzione. Perché non si può usare lo stack del processo utente interrotto? Perché in presenza di più thread su più processori, un processore che esegue un handler può essere danneggiato da thread dello stesso processo eseguiti su un altro processore.

Interrupt Masking
Se durante l'esecuzione di un handler si verifica un'interruzione il sistema rischia di andare in crash, per questo, prima di avviare l'handler, si passa a kernelmode e si disabilitano gli interrupt. Le interruzioni arrivano ma il processore le ignora
Poiché l'handler è eseguito con interruzioni disabilitate, si dice che è **non-blocking**.
Anche il kernel può disattivare interruzioni, ad esempio quando **determina il prossimo processo/thread da eseguire**, ma se postpone le interruzioni per troppo tempo può ignorare eventi I/O. Su x86 si usano i seguenti comandi:

- CLI: disattiva interruzioni
- STI: arriva interruzioni

L'attivazione/disattivazione in ogni caso **vale solo per processore corrente**. Le interruzioni vengono passate ad altri thread, per tenere le interruzioni disabilitate il meno possibile.

Trasferimento di Controllo Atomico
Abbiamo una **singola istruzione atomica**, quindi **indivisibile**, per cambiare contesto, ciò eseguire **contemporaneamente**. **IRET**:

- Aggiornamento PC, quindi saltare al codice handler
- Aggiornamento SP, per puntare allo stack del kernel (in caso non ci siano due stack pointer, uno user e uno kernel)
- Disattivare protezione memoria
- Commutare kernel/user mode
- Disabilitare interruzioni
- Salvare lo stato (SP, PC, process status word) del processo interrotto per poterlo far ripartire come se niente fosse e salva anche tutti i registri che potrebbe alterare

Quindi commuta a kernel mode, a kernel stack, mette PC, SP e PSW del processo sullo stack kernel
Una volta finito l'handler, per ripristinare lo stato del processo:

- si ripristinano i registri salvati
- atomicamente si ritorna al processo/thread interrotto (istruzione **IRET**):
    - ripristina PC
    - ripristina PS
    - ripristina PSW
    - commuta a user mode
    - attiva interrupt

Esecuzione Riavviabile Trasparente
Il programma utente **non deve sapere che c'è stato un interrupt**, quindi l'handler **non deve modificare lo stato del processo utente**.

Interrupt Handlers
Un handler non si blocca fino a che non è stato completato (**non-blocking**):

- esegue il minimo necessario per permettere al dispositivo di prendere il prossimo interrupt
- ogni attesa deve essere di durata limitata, altrimenti si perde troppo tempo
- attiva altri thread per eseguire il vero lavoro, così l'handler dura poco perché una volta che ha avviato thread (eseguiti ad interruzioni abilitate) termina.

In questo modo un handler dura molto poco e può essere eseguito a interruzioni disabilitate.

System Calls

Poiché un processo in stato utente può **generare indirizzi solo nel suo spazio di memoria**, **non può direttamente invocare funzioni di sistema** perché dovrebbe eseguire un salto nel kernel e ciò violerebbe le politiche di protezione della memoria. Se così non fosse, un qualsiasi processo **potrebbe saltare arbitrariamente nel codice di altri programmi o del sistema operativo**, generando problematiche di sicurezza. Questo meccanismo sfrutta le interruzioni.

Il **meccanismo delle System Call** comprende 6 passaggi fondamentali:

1. All'interno del mio codice utente eseguo una **funzione che usa un componente del sistema operativo** (ad esempio una **printf** che utilizza l'hardware). Ciò significa eseguire una **chiamata di sistema** ed entrare nello **stub**. Lo **stub** è una funzione che predispone la corretta invocazione del codice del sistema operativo. Non faccio una vera e propria chiamata di sistema ma chiamo una funzione della libreria del linguaggio (es. printf nella libreria di I/O del C) che va a richiamare la vera e propria chiamata di sistema: questa è la **stub**.
2. Dallo **user stub**, tramite una funzione chiamata **trap** (**hardware trap** in questo caso), si passa il controllo all'**handler** del sistema operativo, all'interno del **kernel stub**. La **trap** è una istruzione non privilegiata che setta un bit di interruzione (unico per tutte le chiamate di sistema)
3. L'**handler** copia i parametri dalla memoria utente a quella del kernel, li convalida ed esegue la **chiamata di sistema vera e propria**
4. Il kernel esegue la chiamata di sistema e a fine esecuzione ripassa il controllo all'handler
5. L'handler si occupa di copiare il valore di ritorno nella memoria utente e di ritornare il controllo allo user stub (**trap return**)
6. Lo user stub ripassa il controllo al programma chiamante, concludendo il processo.

Poiché il bit di interruzione è unico per tutte le chiamate di sistema, come capire quale chiamata di sistema eseguire? Dato che gli argomenti si prendono o dallo **stack utente** (l'handler può farlo perché è del kernel) o vengono letti dai **registri generali** del processore, a seconda di dove si trovino, tramite essi si capisce quale chiamata di sistema è stata richiesta.

Le interruzioni vengono disabilitate quando viene chiamata la **trap**, riabilitate appena sono prelevati i parametri, quindi la system call vera e propria, nel kernel, viene eseguita a interruzioni abilitate. Questo per evitare di monopolizzare il processore perché le chiamate di sistema possono richiedere parecchio tempo. Quando il risultato viene copiato in memoria utente le interruzioni sono disabilitate, per essere riabilitate subito dopo.

Upcall

Dette anche **user-level interrupt**, sotto Unix sono chiamate **signal**.

L'upcall è un meccanismo analogo alle interruzioni che notificano i processi utente di eventi che devono essere gestiti subito: ad esempio allo scadere del quanto di tempo per il manager dei thread livello utente, o per implementare i meccanismi di interrupt per le VM.

Essendo l'upcall una diretta analogia degli interrupt kernel, hanno diverse caratteristiche simili:

- possiedono **handler di segnali**, con entry point fissi
- hanno una **signal stack** separata
- salva e ripristina i registri automaticamente, come per gli interrupt la ripresa del processo avviene in maniera trasparente
- **signal masking**: i signal sono disabilitati quando viene eseguito un signal handler

Quindi un processo viene eseguito e, quando arriva un segnale, esegue del codice in un'altra parte della memoria per poi riprendere come se non fosse successo niente.

Booting

Il **booting** consiste in una serie di passaggi che puntano a caricare il sistema operativo nella memoria centrale:

1. **BIOS**:
    a. guarda quale disco è attivo
    b. cerca la partizione di quel disco che contiene il sistema operativo e ne carica il **bootloader**. Il **bootloader** è specifico per sistema operativo ma si trova sempre nello stesso punto del disco (non è il BIOS a caricare direttamente il sistema operativo, altrimenti ogni BIOS potrebbe caricare un solo sistema operativo)
    c. copia il codice del bootloader e i suoi dati in memoria
2. **Bootloader**
    a. copia il codice del kernel e i suoi dati in memoria
3. **Kernel**
    a. copia il codice dell'applicazione di login e i suoi dati in memoria

Alcune fasi sono eseguite ad interruzioni disabilitate.

Macchine Virtuali

Una **macchina virtuale** consiste in un **processore fasullo** con **disco virtuale implementato in processo in stato utente** dove viene installato un sistema operativo nuovo detto **ospite**. Il sistema operativo ospite deve essere convinto di essere modalità supervisore e **mai** deve sospettare di essere in stato utente. Questo meccanismo è detto di **virtualizzazione** e per essere gestito si fa largo uso di meccanismi di interruzione di upcall.
La macchina virtuale viene eseguita a livello utente ed installa driver kernel trasparenti all'host: **richiede i diritti di amministratore** e **modifica la tabella di interrupt** per puntare a codice della macchina virtuale, così se l'interrupt è per la macchina virtuale la tabella punta a codice della macchina virtuale e viene eseguita una upcall, altrimenti ripristina la tabella degli interrupt e riprende il kernel.

Struttura dei sistemi operativi

Linux

Users
User mode System tools (shell, editors, compilers, ...) **funzioni della libreria standard** ↓
User mode Libreria standard (printf, fork, ...) **system call** ↓
Kernel mode Kernel: processi, memory management, file system, I/O...
Hardware

Windows

User mode [Processo A] [B] [C] →[Sottosistemi di ambiente ↕$_{executive}$] **system call** ↓
Kernel mode [Device drivers ↕], [Executive]→[Kernel]→[HAL ↕]
[Hardware]

HAL: hardware abstraction layer

Programming Interface

Shell

La shell è un sistema di controllo dei job (**job control system**) che consente al programmatore di creare e gestire un insieme di programmi (Windows, OS X e Linux possiedono programmi di shell).
Non funziona in kernel mode, non manipola il nucleo. A tutti gli effetti è un processo utente. Per far partire un processo quindi dovrebbe agire a livello del nucleo (passare da file eseguibile a processo in esecuzione):
deve creare nuovo processo e far in modo di copiare informazioni da file eseguibile a processo (iniz. dati, stack, codice ...). Questo è fattibile solo a livello kernel.
Se la shell è **user-level**, quali sono le chiamate di sistema per eseguire i programmi?

Windows: CreateProcess

CreateProcess è una chiamata di sistema che crea un nuovo processo per eseguire un programma

- crea e inizializza il PCB nel kernel
- crea e inizializza il nuovo spazio indirizzi
- copia il codice del programma nel nuovo spazio
- copia gli argomenti in memoria
- ...

Ha una quantità di parametri enorme: quale programma, quali parametri, quanta memoria, dove allocarla...

Unix Process Management

- **fork**: crea una copia del processo corrente e lo lancia, non richiede argomenti
  ritorna un intero: per il figlio è 0, per il padre è: valore positivo che indica PID oppure valore negativo che indica un errore
  inizia a eseguire dal solito punto, non riparte da capo
  Ogni processo ha un padre e può avere dei figli. Il PID del processo padre è scritta nel PCB. Il figlio quando creato eredita una copia di **tutti** i dati del padre. Una volta fatta la fork non si può sapere in anticipo chi viene eseguito per prima: questo perché ci sono due processi **separati** che agiscono separatamente e ciò dà origine a tutta una serie di problemi di sincronizzazione.
  Implementazione:
  - **crea e inizializza PCB** nel kernel
  - **crea nuovo spazio indirizzi**
  - **inizializza lo spazio indirizzi** con copia dell'intero contenuto dello spazio indirizzi del padre
  - **copia gli argomenti** nella memoria dello spazio indirizzi
  - **eredita il contesto di esecuzione** (ad esempio i file aperti)
  - **informa lo scheduler** che c'è un nuovo processo pronto per essere eseguito
- **exec**: cambia il programma eseguito dal processo corrente
  Riutilizza il processo esistente, **non crea nuovo processo**, ricicla anche il PCB
  Mantiene lo stesso stack e le stesse risorse (es file).
  Se **avviene con successo** non ritorna niente (il **valore di ritorno sarebbe inutilizzabile** e pertanto non significativo) ed **esegue un nuovo programma nel processo esistente**.
  Se **c'è un errore** il **valore di ritorno è significativo**, il **codice non viene cancellato** e si **prosegue nel codice "originale"**.
  Dopo l'esecuzione, si **mantiene il PID**, **il PCB** (**cambiano i riferimenti alla memoria** del codice e dei dati), vengono **azzerati i segnali** pendenti, si **mantiene il kernel stack** e **le risorse assegnate**.
- **wait**: aspetta la terminazione di un processo dato il PID
- **signal**: manda notifiche tra processi

Terminazione dei processi

Un processo **termina** quando: **commette errori**, quindi viola meccanismi protezione o compie un'operazione illecita, o **invoca la system call *exit***.

Il processo terminato **restituisce il codice della terminazione** al processo padre:

- il padre **riceve il valore tramite la system call *wait***, se figlio non ha fatto exit il padre resta sospeso.
- se il padre non ha già chiamato la wait il processo terminato **commuta allo stato *zombie***, la **exit resta sospesa** e il **codice di terminazione viene conservato**. Non è utile tenere allocata la memoria del codice e dei dati perché il processo è effettivamente terminato, quindi **si dealloca tutto tranne le strutture dati del kernel che descrivono il processo** (questo è lo stato *zombie*).
- se il padre è già terminato, il processo init aspetta la terminazione del figlio.

Linux **deve garantire che il codice di terminazione raggiunga il padre**.

`void exit(int status)`

- **status è il codice di terminazione** e la **exit non ritorna mai niente**
- **libera la memoria** e **rilascia le risorse**, se deve commutare a zombie **mantiene il PCB fino a che il padre non invoca la *wait***.

`int wait(int *status)`

- **status è il PID del processo terminato** o un codice d'errore

Implementazione di una shell di base

```
char *prog, **args;
int child_pid;
// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
      child_pid = fork(); // create a child process
      if (child_pid == 0) {
            exec(prog, args); // I'm the child process. Run program
            // NOT REACHED
      } else {
            wait(child_pid); // I'm the parent, wait for child
      return 0;
      }
}
```

Concorrenza

A cosa serve studiare e usare la **concorrenza fra processi**? Le varie soluzioni informatiche sono quasi tutte soluzioni **MTAO** (**M**ultiple **T**hings **A**t **O**nce), cioè devono gestire più cose contemporaneamente.
Sistemi Operativi devono: **gestire le interruzioni**, **gestire le interazioni**, **eseguire diversi comandi**, occuparsi del **mantenimento del sistema in background**...
Server devono gestire **più connessioni contemporaneamente**. Più programmi vengono eseguiti in parallelo per ottenere **miglior performance**. Programmi con GUI devono gestire la **computazione** e contemporaneamente essere **responsivi agli input dell'interfaccia utente**.
Quindi non solo i sistemi operativi sono MTAO ma anche i processi utente possono aver bisogno di gestire MTAO.

Esigenze per MTAO

Abbiamo visto che un processo è un **flusso di esecuzione** ed è un'astrazione necessaria per **realizzare il dominio di protezione**. Anche se un programma può lanciare più processi, essi non possono comunque condividere memoria per, ad esempio, lavorare contemporaneamente sugli stessi dati. Per questo viene introdotto il concetto di **thread**.

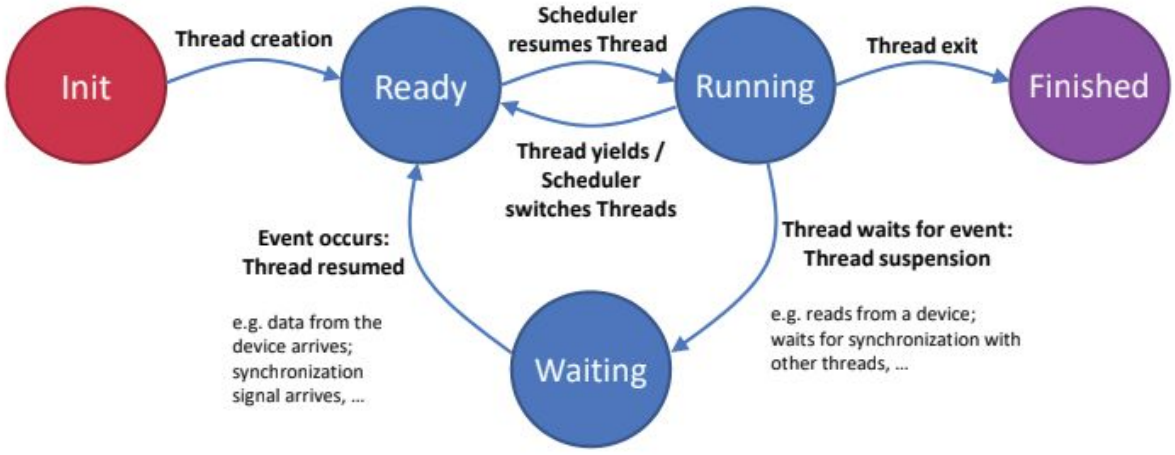
I thread

Un **thread** è una **singola sequenza di esecuzione** (**sequenziale**) che **rappresenta un'attività separata e schedulabile**.
La protezione è un concetto ortogonale a quello dei thread:

- posso avere **uno o più thread per dominio di protezione**
- un **programma utente a singolo thread ha un thread e un dominio di protezione**
- un **programma utente multi-thread ha thread multipli**, che condividono le stesse strutture dati, isolate dagli altri processi utente
- **kernel multi-thread**: thread multipli, condividono le strutture dati del kernel e possono usare istruzioni privilegiate.

Il concetto di thread è anche visto come un **numero infinito di processori dove mandare un infinito numero di sequenze**, sarà poi lo scheduler a mandare fisicamente il thread sul processore fisico disponibile, ma dal punto di vista del programma e del programmatore i thread sono eseguiti contemporaneamente e in parallelo. I thread vengono **eseguiti a velocità variabile** quindi i programmi vanno scritti per lavorare con qualsiasi schedulazione.
Astrazione: tutti (n) in contemporanea su n processori virtuali
Realtà: x in esecuzione su x processori fisici e n-x in attesa
Ma non è importante l'aspetto fisico, il programmatore se che tale thread esegue sequenzialmente quelle istruzioni. Vengono **creati ordinati**, ma **eseguiti e terminati in ordine arbitrario**.
Il **numero massimo di thread eseguibili contemporaneamente è pari al numero di processori fisici**, mentre il **numero minimo è** logicamente **0**.

Ciclo di vita di un Thread



Implementare un thread
Thread Control Block TCB: struttura dati con informazioni sul thread analoga al PCB. Contiene informazioni quali:

- il **contesto**
- i **parametri di scheduling**
- il **thread ID**
- il **process ID**
- **...**

Abbiamo poi un certo numero di operazioni sui thread:

- `sthread_fork(func, args)`
  Crea un nuovo thread che eseguirà *func(args)*.
  - **alloca TCB**
  - **alloca stack**
  - **costruisce lo stack frame** come base dello stack (stub)
  - **mette *func, args* nello stack**
  - **mette thread su lista ready**
  - **eseguirà fra un po' (forse subito)**

  Buona norma invocarlo tramite `stub(func, args)` che esegue:
  - `call (*func)(args)`, **invoca la funzione** e
  - `call thread_exit()`, **chiama la exit**, per sicurezza perché deve esserci
- `sthread_yield()`
  Rilascia volontariamente il processore lasciando andare il prossimo nello scheduler

- `sthread_join(thread)`
  Eseguito nel parent, aspetta che il thread che è partito dalla fork faccia *exit*, quindi ritorna.
- `sthread_exit`
  Esce dal thread e fa pulizia, sveglia il joiner se presente.

Quando un **thread viene creato** va nello **stato ready**: il **descrittore del thread viene inserito in una struttura dati del processore** che contiene tutti i thread pronti. In ready **il thread sta fermo e congelato**. Quando lo scheduler lo seleziona lo **toglie da ready** e **lo porta in esecuzione**, cioè **esegue la commutazione di contesto** spostandolo da ready al processore e così il thread è in esecuzione.
Il thread esegue le istruzioni programma:

- se **esegue una *exit*** allora **passa allo stato finished** e **il thread smette di esistere**
- se il thread **non termina prima**:
  - **scade il timer** (nei sistemi operativi time sharing) e **lo scheduler lo riporta in ready** eseguendo un'altra commutazione di contesto
  - il thread **vuole un dispositivo** ed **esegue la chiamata di sistema** associata, quindi deve **aspettare il risultato** dal dispositivo e non può quindi andare in ready, quindi va in **waiting**. Questo passaggio è **causato volontariamente dal thread stesso**.

In sintesi:

Thread creato (**INIT**)	TCB creato	Registri in TCB
Thread pronto (**ready**)	TCB in READY	Registri in TCB
Thread in esecuzione (**running**)	TCB in running list	Registri nel processore
Thread in attesa (**waiting**)	TCB nella lista wait della variabile	Registri in TCB
Thread finito (**finished**)	TCB in finished list, poi eliminato	Registri in TCB

Roadmap
Ci sono e ci sono stati modi diversi di implementare i thread in un sistema operativo

- **early java**: **thread multipli a livello utente dentro processo unix** che ripartisce internamente il tempo assegnato
  - Il context switch è eseguito dalla **libreria user-level** che implementa i thread all'interno del processo
  - il tempo processore assegnato al processo viene ripartito **tra i vari thread** internamente
- **early unix**: **multipli processi a thread singolo**
- **linux, macos, windows**: **mix di processi a thread singolo e multithread e kernel thread**
  - per il kernel, un processo utente singolo o multithread è simile
    - Molte transizioni tra user mode e kernel mode, poiché *fork*, *wait*, *join* eccetera sono tutte chiamate di sistema
    - Il context switch è eseguito dal kernel
- **scheduler activations** (Windows)
  - il kernel alloca i processori alla libreria user-level
  - la libreria thread implementa il context switch
  - una system call i/o che blocca lancia una upcall
  - il timer virtuale viene implementato con una upcall

Rappresentare thread e processi
PCB: struttura dati associata a ciascun **processo**

- PID
- Memoria assegnata
- Risorse: dispositivi, file…
- Handler per i thread del processo
- …

Process Table: situata nel kernel ed **unica per tutto il sistema**, **contiene tutti i PCB**

TCB: struttura dati associata a ciascun **thread**

- Thread ID
- Stato
- Contesto del thread
- Parametri di scheduling
- Riferimenti allo/agli stack
- ...

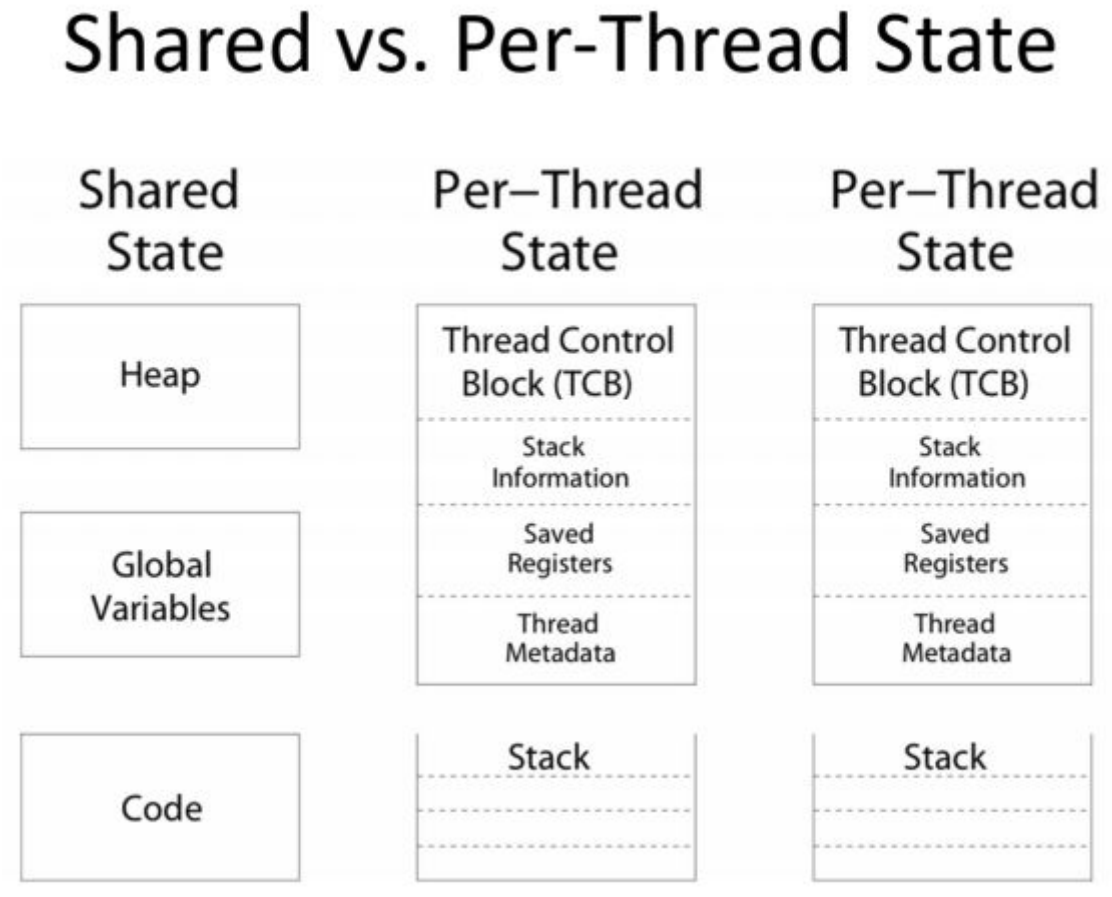
Thread Table: per i thread a livello utente è **una per processo**, per i thread a livello kernel è **unica in tutto il sistema**.
Dal PCB ai TCB sono passate le informazioni sulla schedulazione.

Il **contesto attuale è nel processore**
il **contesto del thread quando è ready è nel TCB**
il TCB è in memoria del processo in stato utente/kernel, più precisamente nella lista dei thread pronti

Thread user-level

- Implementati da chiunque anche su sistemi operativi senza supporto al multithreading
- Il sistema operativo non ha la visione dei thread user-level quindi non li schedula, vengono eseguiti in un thread unico a livello del thread processo: non sfruttano architetture multithread
- Thread table per processo: i thread sono relativi al singolo processo, quindi non gestiti dal S.O.
- Scheduling thread a livello utente svolto dalla libreria che li implementano e non dal sistema operativo
- **Una system call blocca tutti i thread** poichè blocca il processo e non il thread (s.o. non vede il thread quindi non può sospendere solo quello)
- Creazione, terminazione e commutazione di contesto molto efficienti poiché riferiscono alla libreria dei thread e non a system call mentre per la commutazione di contesto lo spazio di indirizzamento è il solito

Shared vs Per-thread state



Cosa succede se thread mette troppa roba nello stack?
Cose dovrebbe succedere: quando finisce memoria del processo non c'è più spazio per lo stack thread cumulativo di tutti i thread.
Un thread può occupare tutta la memoria del processo? **No perché prima o poi sovrascrive stack di un altro thread**. La memoria è una "striscia" quindi a porzioni dedicate ai vari thread con spazio tra esse, se un thread scrive indiscriminatamente finisce per scrivere in zona non sua: problemone.
Il sistema operativo in genere non lo impedisce perché non ha meccanismi protezione per thread dello stesso processo. Probabilmente nemmeno il supporto a tempo di esecuzione ha controlli in merito.

Implementare thread user-level
thread_fork(func, args)
- Alloca il TCB e lo stack
- Costruisce il frame alla base dello stack (stub)
- Mette func e args nello stack
- Mette thread nella lista ready
- Prima o poi verrà eseguito (forse subito!)

stub(func, args)
- chiama (*func)(args)
- chiama thread_exit()

Thread kernel-level
- Sono i thread implementati nel kernel
- Thread Table **nel kernel**
- Creazione, terminazione e commutazione di contesto **attivate da system call**
- **Thread separati dello stesso processo possono essere eseguiti in parallelo su processori separati**
- **Solo il thread che invoca** chiamate di sistema bloccanti **si blocca**

Thread switch
Una commutazione di contesto dei thread, o **thread switch**, può avvenire per due motivi:
- essere volontaria cioè che il **thread decida spontaneamente di rilasciare il processore** con una *yield*
- essere dovuta ad **interruzioni** o **eccezioni**

Per eseguire un thread switch si attua quasi lo stesso procedimento in tutti i differenti casi (kernel o user thread, processi multithread/singlethread)

Implementazione del thread switch **volontario** attuato da un user-level thread in un processo single-thread
- **salva registri** sul vecchio TCB
- **commuta al nuovo stack** e al **nuovo thread**
- **ripristina registri dal TCB** del nuovo thread
- **ritorna** (o *iret*)

Nei kernel thread viene eseguito **esattamente lo stesso procedimento**.

Implementazione del thread switch eseguito a seguito di un'**interruzione**. Questo può avvenire per l'interruzione timer o per I/O.
Versione **semplice**:
- alla fine dell'interrupt handler si chiama *switch_threads()*
- quando riprende, ritorna dall'handler e riprende il kernel thread o l'user thread

versione **veloce**:
- interrupt handler ritorna allo stato salvato nel TCB
- può riprendere un kernel thread o un user process/thread

Implementazione del thread switch

- **salva registri** (con stato) del **vecchio thread** nel TCB
- **sposta** il TCB del **vecchio thread in ready list o in waiting list**
- **seleziona il nuovo thread** dai ready
- **ripristina i registri del nuovo thread** dal TCB al processore
- **mette** il TCB del **nuovo thread nella lista running**
- **ritorna il controllo al nuovo thread** (**IRET**)

L'**overhead** è dato da:

- **salvataggio e ripristino** continuo di registri
- **gestione** della coda di **TCB**
- **memory cache invalidation**, è un costo nascosto perché la cache attuale non va bene (era del vecchio thread) e bisogna andare in **memoria**
- induce **operazioni sul manager della memoria**
    - **eccezioni** di indirizzi
    - **page fault**
    - **MMU invalidation**

04-concurrency.pdf, slide 39: ***esempio di context switch***

Modelli di cooperazione

Modello globale: i processi **condividono dati**, quindi **possiedono aree memoria condivise**. Possono avere anche dati privati ma hanno una **zona di memoria condivisa**.
Tipicamente questo avviene con i thread di uno stesso processo.

Modello locale: i processi **non condividono dati** e **non hanno memoria condivisa**. Ogni singola entità può usare solo la propria memoria quindi la **cooperazione deve avvenire tramite uno scambio di messaggi**.
In questo modello, comunque, se processi caricano la stessa libreria o eseguono una fork, cioè si pongono in condizione di avere **codice in comune**, essendo il codice **passivo e immutabile** è posto in aree di memoria condivisa.
La cooperazione avviene quindi con meccanismi espliciti, ad esempio con **canali di comunicazione**.
Questo modello è usato nei **processi unix** e nei **processi distribuiti** (ad es. internet).

Poiché in un ambiente globale la modifica di un thread alla memoria vista da tutti gli altri thread, si pongono problemi di **concorrenza** e **sincronizzazione** dei thread.

Liveness: svolgere il compito
Safety: svolgere senza sbagliare ad alterare i dati condivisi

Sincronizzazione

L'**esecuzione contemporanea** di più thread che manipolano la stessa area di memoria, in lettura **e** scrittura, pone il problema di **garantire la consistenza dei dati** acceduti. Nasce quindi la **necessità di mutua esclusione**, cioè garantire il fatto che se qualcosa la sta facendo qualcuno allora tassativamente non la fa qualcun'altro.

Race condition: **il risultato** di un'azione concorrente tra più entità **dipende dall'ordine** con cui le entità vengono eseguite.

Mutual exclusion: **se un'attività svolge una certa azione allora nessun'altra può svolgere quell'azione** in quel momento.

Critical section: porzioni di codice che possono essere eseguite solo da un thread alla volta.

Lock: meccanismo che impedisce a qualcuno di fare qualcosa (es. accedere ad una risorsa critica quando qualcun altro la sta usando)

- **lock prima di entrare** nella sezione critica, **prima di accedere** a dati condivisi
- **unlock quando esci** dalla sezione critica, **dopo aver accesso** a dati condivisi
- **wait se** la risorsa alla quale vuoi accedere è sotto **lock** (tutte le sincronizzazione coinvolgono waiting!)
- Una volta che thread ha ottenuto lock nessun altro può ottenerlo e gli altri dovranno attendere.

Liveness: qualsiasi soluzione deve garantire che **se c'è bisogno di fare qualcosa almeno uno la faccia**

Safety: quel qualcosa la può fare **al massimo uno per volta**

esempio del latte: 1 tentativo: lasciare un biglietto
if not note
    if not milk
        leave note
        buy milk
        remove note
    endif
endif
comunque problema di sincronizzazione
-> tentativo 2, lascio nota e controllo se non ci sia la nota dell'altro thread
ma nessuno compra il latte

Il problema permane in ogni caso se si realizza **codice simmetrico**, va quindi **rotta la simmetria tra thread concorrenti** e fare in modo che un thread attenta la conclusione del controllo da parte dell'altro thread. Quindi **un thread deve garantire la safety** mentre **un altro thread deve garantire la liveness**.

Il modo più semplice di realizzarlo è fare attendere ciclicamente un thread che la risorsa sia libera e poi tentare di usarla, mentre l'altro thread la usa subito se può. Questo **complica molto il codice** e **non è molto efficiente**, perché un thread aspetta per un po' **senza fare niente**.
Quindi

- la **soluzione** è **complicata**: il codice ovvio spesso presenta bug
- i **compilatori e processori** moderni **riordinano le istruzioni**, quindi le soluzioni simmetriche software possono fallire per riorganizzazione del codice
- le **generalizzazioni** a molti thread/processori **aggiungono complicatezza**

Serve quindi una soluzione senza attesa attiva, con aiuto da parte dell'hardware: la **lock**.

Questo codice funziona?

```
[...]
if (p == NULL) {
  lock_acquire(lock);
  if (p == NULL) {
    p = newP();
  }
  release_lock(lock);
}
use p->field1
```

```
newP() {
   p = malloc(sizeof(p));
   p->field1 = ...
   p->field2 = ...
   return p;
}
```

In generale no. Può capitare **field1 non inizializzato perché testiamo (p == NULL) fuori dalla lock**
io locko e inizializzo il puntatore **ma non il campo** (mi deschedula prima), entra altro thread e testa P che risulta allocato (ma non inizializzato correttamente) e va direttamente a usare field1 non inizializzato.
Anche il puntatore P è condiviso, quindi anche il suo accesso deve essere eseguito all'interno della lock

Locks
lock acquire: si **attende** che il lock sia **disponibile** e **lo si acquisisce**
lock release: si **rilascia** il lock **svegliando chiunque lo stia attendendo**
Un processo può ottenere massimo un lock alla volta (**safety**)
Se nessuno sta tenendo lock, il primo che esegue l'*acquire* lo ottiene (**progress**)
Se tutti i lock holder concludono e non c'è *waiter* con priorità più alta, un *waiter* prima o poi ottiene il lock (**liveness**)
I lock sono implementati con supporto hardware ed istruzioni per leggere/modificare/scrivere
Il meccanismo dei ock consente di scrivere codice concorrente in maniera molto più semplice.

Regole per usare lock

- inizialmente libera
- **prima di accedere** alle strutture dati condivise bisogna **acquisire la lock**
  es. all'inizio della procedure/funzione
- **rilasciare sempre dopo aver usato dati condivisi**
  es. alla fine della procedure
  Non lasciare rilasciare la lock a qualcun'altro, fallo subito
- **Non accedere a dati condivisi senza lock**!

Condition variables

- mantieni sempre lock quando usi variabili condizione (wait, signal, broadcast).
- **mai usarle fuori dalla lock**
- variabili di condizione **sono senza memoria**
    - se signal senza nessuno che aspetta **essa si perde, evapora**
- **Wait rilascia atomicamente** il lock
  No prima wait e poi release perché potrei essere deschedulato tra le due. In caso, potrebbe entrare nel lock un altro thread che fa una signal ma viene persa
- Signal sveglia 1 waiter (il primo della lista wait), broadcast sveglia tutti
- quando thread riattivato dopo wait **potrebbe non essere subito messo in esecuzione** ma viene messo in stato di pronto.
  Questo è importante perché il thread riattivato non detiene il lock e le condizioni potrebbero cambiare. Ecco perché wait deve stare in un loop.
- La wait deve **sempre** stare in un loop
    - while (needtowait()) condition.Wait(lock);

Sincronizzazione strutturata

- identificare strutture od oggetti che possono essere **accessi da più thread concorrentemente**
- aggiungere lock agli oggetti
    - ottienilo all'inizio della procedura
    - rilascialo alla fine
- Se c'è bisogno di wait
    - while(needtowait...
    - non dare per scontato che, quando finisci wait, sia appena stato eseguito il signaler. Quindi sempre riverificare le condizioni
- Se fai qualcosa che potrebbe svegliare qualcuno
    - signal o broadcast
- Lascia sempre dati condivisi in uno stato consistente
    - Quando rilasci lock o quando wait

Mesa vs Hoare

- Mesa
    - signal mette waiter in lista ready
    - signaler mantiene lock e processore
    - Cioè **signal != waiter in esecuzione**
- Hoare
    - signal dà processo e lock a waiter
    - quando waiter finisce, processore e lock ridate a signaler
    - possibili signal annidati
    - Cioè **signal = waiter in esecuzione**

Implementare sincronizzazzione

- usare memoria load/store
    - come esempio del latte
      usa attesa attiva, da evitare come la peste
- secondo metodo
    - lock.acquire disabilita interruzioni **!DANGER**
    - lock.release riabilita interruzioni **!DANGER**
    - Non funziona con i thread perché un thread che disabilita interruzioni è **pericoloso**, blocca lo scheduler e potrei perdere input/output dai dispositivi. Sono meccanismi che non posso dare in mano ai processi utente
-

Lock Implementation, Uniprocessor

```
LockAcquire(){
  disableInterrupts ();
  if (value == BUSY) {
    waiting.add(current TCB);
    suspend();
  } else {
    value = BUSY;
  }
  enableInterrupts ();
}
```

```
LockRelease() {
  disableInterrupts ();
  if (!waiting.Empty()){
    thread = waiting.Remove();
    readyList.Append(thread);
  } else {
    value = FREE;
  }
  enableInterrupts ();
}
```

La **coda di attesa è relativa alla singola lock**. Suspend() invoca lo scheduler, il context switch e riabilita le interruzioni.
Sezione critica della lock, comprende dati lista processori in attesa e value (il valore della lock). Essendo sezione critica devo proteggerlo, per impedire interferenza su value o sui processi. Su uniprocessore basta disabilitare le interruzioni.
Su multiprocessore ciò non basta, perché se disattivo le interruzioni si blocca un solo processore, gli altri processori (e thread) continuano a lavorare.

Multiprocessore
Read-modify-write instructions: istruzione **atomica** per leggere e scrivere una stessa cella di memoria.
L'hardware fa si che anche se l'operazione è eseguita da due proc separati essa sarà **strettamente serializzata**.
Quindi vi è un **supporto hardware più esteso**.
esempi: test and set, xchgb, lock prefix, compare and swap.

Spinlock

Lock dove il processore aspetta in un loop fino a che il lock non si libera. **attesa attiva**

- si assume che la lock rimanga occupata per poco tempo
- viene usato per proteggere la lista ready dall'implementazione

```
SpinlockAcquire() {
  while (testAndSet(&lockValue) == BUSY)
    ;
}
SpinlockRelease() {
  lockValue = FREE;
}
```

Lock Implementation, Multiprocessor

```
LockAcquire(){
  spinLock.Acquire();
  disableInterrupts ();
  if (value == BUSY){
    waiting.add(current TCB);
    suspend(&spinLock);*
  } else {
    value = BUSY;
    enableInterrupts ();
    spinLock.Release();
}
* Marks thread as waiting;
  enables interrupts; release
  spinlock; invokes scheduler;
```

```
LockRelease() {
  spinLock.Acquire();
  disableInterrupts ();
  if (!waiting.Empty()){
    thread = waiting.Remove();
    readyList.Append(thread); *
  } else {
    value = FREE;
  }
  enableInterrupts ();
  spinLock.Release();
}
* Marks thread as ready
```

Implementazione su linux

Fast path:

- se lock è FREE e nessuno wait, test&set

Slow path:

- se lock è BUSY o qualcuno wait, vedi screenshot sopra

Su user-level:

- Fast path: acquisici lock con test&set
- Slow path: system call al kernel per usare il kernel lock

Semafori

Semafori non più offerti ai programmatori, ora relegati a funzionalità del nucleo. Considerati pericolosi.

Semafori hanno **valori interi non negativi associati alla coda**:

- P() **aspetta atomicamente che il valore diventi > 0, quindi decrementa**
- V() **atomicamente incrementa il valore** (**se non sono presenti thread sospesi**, in tal caso riattiva un thread sospeso **senza incrementare**)

La struttura dati del semaforo è int + coda: operazioni sono **solo P e V**, e sono atomiche (se valore è 1, due P ritorneranno 0 e un thread attesa)

P&V Implementation, Multiprocessor

```
P(sem){
  spinLock.Acquire();
  disableInterrupts ();
  if (sem.value == 0){
    waiting.add(current TCB);
    suspend(&spinLock); *
} else {
    sem.value --;
    spinLock.Release();
    enableInterrupts ();
}
* Also enables interrupts
```

```
V(sem) {
  spinLock.Acquire();
  disableInterrupts ();
  if (!waiting.Empty()){
    thread = waiting.Remove();
    readyList.Append(thread);
  } else {
    sem.value ++;
  }
  spinLock.Release();
  enableInterrupts ();
}
```

Semaphore Bounded Buffer

```
get() {
  empty.P();
  mutex.P();
  item = buf[front]
  front= (front+1) % size;
  mutex.V();
  full.V();
  return item;
}
```

```
put(item) {
  full.P();
  mutex.P();
  buf[last] = item;
  last = (last +1) % size;
  mutex.V();
  empty.V();
}
```

Initially: front = last = 0; size is buffer capacity
empty/full are semaphores (initialized to 0 and size)
Mutex is a semaphore initialized to 1

Il semaforo deve essere inizializzato a 1 (solo uno alla volta nella sezione critica)
empty inizializzato a 0 perché inizialmente buffer vuoto e primo che arriva deve bloccarsi
full inizializzato a dimensione perché massimo n processi scrivere contemporaneamente perché n celle
in ogni istante valore empty dice quante celli in lettura, full quante le celle libere

implementing condition variables using semaphores take 1
se signal senza nessuno sospeso evapora, non va bene perché wait e signal non hanno stato e implementando così sto dando uno stato. Se inverto sem.P() e lock.release() peggioro le cose, se faccio P e semaforo è 0 mi sospendo tenendo il lock

take 2
“if semaphore are not empty” non è legale, uniche operazioni sono P e V.
deschedulo tra release e P senza fare P quindi non in attesa, faccio signal ma semaforo vuoto quindi niente V, faccio P ma non avrò mai V quindi wait perenne.
-> take 3, devo usare le liste per mantenere stato e gestire lock indipendenti

Riassunto synchronization

- Usare una struttura consistente
- Usare **sempre** lock e condition variables
- Acquisire lock all’inizio della procedura, rilasciarla alla fine
- Quando si usa una condition variable **sempre** mantenere una lock
- wait **sempre** in ciclo while
- Mai usare sleep() per sincronizzare il thread

Multi-object Synchronization
Quando bisogna sincronizzare più oggetti insorgono problemi: si rischia di fare attendere processi/thread indefinitamente, o di finire le risorse. In multithread si coopera usando oggetti condivisi: la condivisione va gestita bene altrimenti si verificano stalli e problemi.
In programmi grandi dobbiamo sincronizzare su più oggetti ognuno con il proprio lock, variabili di condizione...

Deadlock, definizioni

- **Risorsa**: qualsiasi cosa passiva necessaria a un thread per eseguire il proprio lavoro (CPU, disco, memoria, lock). Può essere:
    - **Prerilasciabile** (preemptable): il sistema operativo la può togliere a sua discrezione
    - **Non-Prerilasciabile**: il thread la deve rilasciare esplicitamente
      Una **sezione critica è non-prerilasciabile** per definizione perché **non può essere ceduta ad altri finché non è conclusa**. Se il sistema operativo potesse riassegnarla a piacimento si perderebbe tutto il senso di aver costruito una sezione critica.
      Un thread in sezione critica **può rilasciare il processore ma non la sezione critica**, sono due **risorse diverse**
- **Starvation**: thread attende indefinitamente l'acquisizione di una risorsa
- **Deadlock**: attesa circolare per ottenere le risorse (ho qualcosa che tu vuoi, hai qualcosa che voglio, non ti dò ciò che ho finché non mi dai ciò che hai e viceversa...)
  deadlock => starvation, non viceversa

Lo stallo si verifica se vanno in wait, attesa circolare per cui nessuno dei due può essere riattivato
Condizioni necessarie perché si verifichi deadlock

- **Accesso limitato** alle risorse e **mutua esclusione**: una risorsa non può essere usata da più processi contemporaneamente, se avessimo risorse infinite non esisterebbe il deadlock!
- **No prerilascio**: se le risorse sono virtuali possono rompere deadlock
- **Multiple richieste indipendenti**: wait while holding
- **Catena circolare di richieste**

Tutte e quattro le condizioni **sono necessarie**: se anche una sola condizione viene meno non si può verificare deadlock.

Come risolvere il deadlock

- **Detect and Fix**
  Essendo una situazione rara, non mi preoccupo più di tanto di stare attento al non verificarsi di deadlock. Se mi accorgo di uno stallo lo aggiusto (**risolvo a posteriori**). Si realizza tenendo un **grafo con tutte le relazioni thread-risorse** e ogni chiamata di sistema aggiorna il grafo. Si visita il grafo cercando i cicli, se se ne trova si avvisa e si risolve.
  Risoluzione **rimuovendo un thread e riassegnando le sue risorse** (avendo scritto un codice di gestione delle eccezioni molto robusto) oppure **eseguendo rollback** delle sue azioni, come in un ambiente database (quindi tutte le azioni diventano provvisorie fino al commit).
- **Prevenzione Statica**
  Non risolvo a tempo di esecuzione ma lo **prevengo a livello di progettazione di sistema operativo** (in modo che **manchi almeno una delle condizioni** necessarie allo stallo).
  Es.: creo una **virtualizzazione** della stampante e assegno una delle stampante virtuali al thread e genero un thread servente che assegna stampante virtuale alla stampante fisica quando è disponibile, quindi per il thread ho stampanti infinite.
- **Prevenzione Dinamica, Algoritmo del Banchiere**
  La richiesta della risorsa viene vagliata dall'algoritmo del banchiere e se approva ok prende risorsa.

Molteplicità di risorse
il modello del grafo non va bene se una risorsa ha più di una moltiplicità (più esemplari di una risorsa)
Quindi rappresento la risorsa a seconda del tipo e con due attributi:

- M moltiplicità quante ce ne sono
- D disponibilità quanto sono disponibili

Le richieste:

- Risorse singole
- Risorse multiple: processo richioede k risorse di un tipo, se k <= D sono tutte assegnate, se k > D nessuna è assgnata (e il processo attende) questo per evitare

Algoritmo del banchiere
dare risorse, per permettere ai processi di andare avanti, e poi riprendere tutto. il sistema deve però stare attento senno finisce in stallo (banca in fallimento)
Bisogna conoscere in anticipo qual'è il massimo numero di richieste che può fare (nell'arco dell'esistenza, quali risorse chiede nel caso peggiore).
Opera ogni volta che c'è richiesta di allocazione di risorse
Alloca le risorse dinamicamente quando la risorsa è richiesta -- aspetta se concedere la risorse porterebbe a stallo, altrimenti la concede se qualche ordinamente sequenziale dei thread è deadlock free
Il banchiere è il gestore di risorse.

All'interno degli stati insicuri i thread non sono in stallo (alcuni almeno sono attivi) ma può finire in deadlock, quindi far sì che evoluzione sistema si muova all'interno di stati sicuri
Non c'è garanzia di uscire dallo stato insicuro.
Definizioni:
Stato sicuro se per ogni possibile sequenza di richiesta di alloc di risorse è possibile trovare sequenza assegnazione (ne basta una) tale per cui tutti i processi terminano. Potrebbe essere necessario wait anche con risorse disponibili.
Stato non sicuro se questo non è possibile
Stato doomed se per ogni possibile computazione portano a deadlock.

Garantisce requeste se result è stato sicuro
La somma delle massime risorse richieste dei thread attuali può essere maggiore delle rirose totali se c'è un qualche modo per non finire in deadlock
esempio...

slide riassuntiva: banker's algorithm for resources of the same type

Filosofi a cena
...
soluzione possibile
uno filosofo prima dx e poi sx, tutti gli altri prima sx e poi dx
Problema costringe a programmare un filosofo (un thread) diverso dagli altri. Cosa da tenere conto ogni volta che si fa manutenzione ed è molto complesso.
Quindi si va verso l'avere un unico protocollo di acquisizione
...soluzione - filosofo i...

Scheduling
Riguarda la gestione del processore. Lo **scheduling** è l'attività che compie il sistema operativo quando deve **decidere a quale thread assegnare il processore**.
Ha effetto su tutta una serie di aspetti riguardo l'efficienza thread (tempo di risposta, throughput...).

Scheduling policy: cosa fare dopo quando ci sono più thread da eseguire (o più pacchetti da spedire, o più richieste da servire...)

Definizioni
Task/job: azioni più brevi svolte all'interno del singolo thread (es. reazione a seguito movimento mouse). Può corrispondere ad un thread ma anche ad una sua parte. Job **non interattivo**, task **interattivo**.
Latenza: tempo che passa dal comando all'azione completata.
Throughput: quanti task completati per il singolo quanto di tempo. Non sempre ottimizzare latenza è ottimizzare throughput anzi spesso sono contrastanti fra loro.
Overhead: quanto lavoro addizionale è svolto dal sistema per effettuare lo scheduling, quindi quello che si "paga" per avere il sistema funzionante. Servono quindi algoritmi di scheduling molto veloci (questo riguarda non soltanto il tempo processore ma anche le commutazioni di contesto, letture e scritture registri, invalidazioni cache...)
Fairness: non bisogna prediligere arbitrariamente un processo rispetto ad un altro
Predictability: le prestazioni devono essere consistenti nel tempo
Workload: il numero di thread in stato pronto ed esecuzione, non in attesa. Cioè il set di task che il sistema deve eseguire.
Preemptive scheduler: prerilascio o no, con prerilascio se in qualsiasi istante posso interrompere l'esecuzione, acquisire processore e riassegnarlo.
Work conserving: ogni qualvolta che un thread è pronto il processore è impegnato in questo thread, non si lascia un processore in idle se ci sono thread pronti
Scheduling algorithm:

- prende workload come input
- decide quale task da mandare per primo
- aspetti prestazionali (throughput, latenza) come output

Tempo medio risposta: media dei tempi di completamento dei vari task. Il tempo di completamento è calcolato dall'istante in cui task arriva al sistema all'istante in cui è completato.

FIFO
Probabilmente primo scheduler usato storicamente, senza prerilascio, task mandati in esecuzione nell'ordine in cui vengono immessi nel sistema.
Su quali carichi funziona particolarmente male? Se chi arriva ha tanto lavoro toglie tanto tempo a chi arriva dopo con poco lavoro

SJF Shortest Job First

- Senza Prerilascio **SJF**
  prendo il task con il più breve tempo e lo mando in esecuzione finché non rilascia spontaneamente o finisce
- con prerilascio **SRTF** Shortest Remaining Time First
  mando quello col tempo rimanente minore, se un task più breve si sveglia context switch e mando il task nuovo

FIFO ottimo quando si comporta come SJF cioè tutti i task hanno solita lunghezza
FIFO pessimo quando arrivano prima i task lunghi da eseguire
Lati negativi SJF: i task lunghi aspettano molto in media e se continuano ad arrivare task brevi il task lungo va in starvation (verrà eseguito ma non si sa quando), inoltre dobbiamo conoscere i tempi di esecuzioni a priori, o almeno le stime e ciò è molto difficile da stimare adesso.

Nei sistemi interattivi l'SJF è limitante. Funziona bene su insiemi di task abbastanza statici e di cui sappiamo tempi e caratteristiche

Round Robin
Prevede Prerilascio
Ogni task **ottiene una determinata risorsa** (ad es.: processore) **per un periodo fisso di tempo** (**quanto di tempo**, **time slice**, **time quantum**, ...), allo scadere del quanto di tempo scatta il timer e il task torna nella coda pronti.
Come scegliere il quanto di tempo?

- qual è troppo lungo? infinito?
- qual è troppo corto? un'istruzione?

La scelta del quanto di tempo cambia molto le prestazioni del sistema. Bisogna trovare un quanto che ottimizzi bene l'overhead ma che non diventi troppo simile al FIFO. I valori tipici odierni sono nell'ordine dei 20-120 ms
Time Slice più piccolo significa maggiore overhead e minor tempo di risposta.
Proporzionalità: turnaround (tempo speso nel sistema) proporzionale alla lunghezza del task
Non ci può essere starvation perché il tempo attesa è superiormente limitato da *#processi pronti * quanto di tempo*, per quanto grande possa diventare è un valore limitato: ciò garantisce tempi risposta misurabili (e certi a parità di condizioni). **Il tempo di risposta è superiormente limitato da** *# processi pronti * time slice*

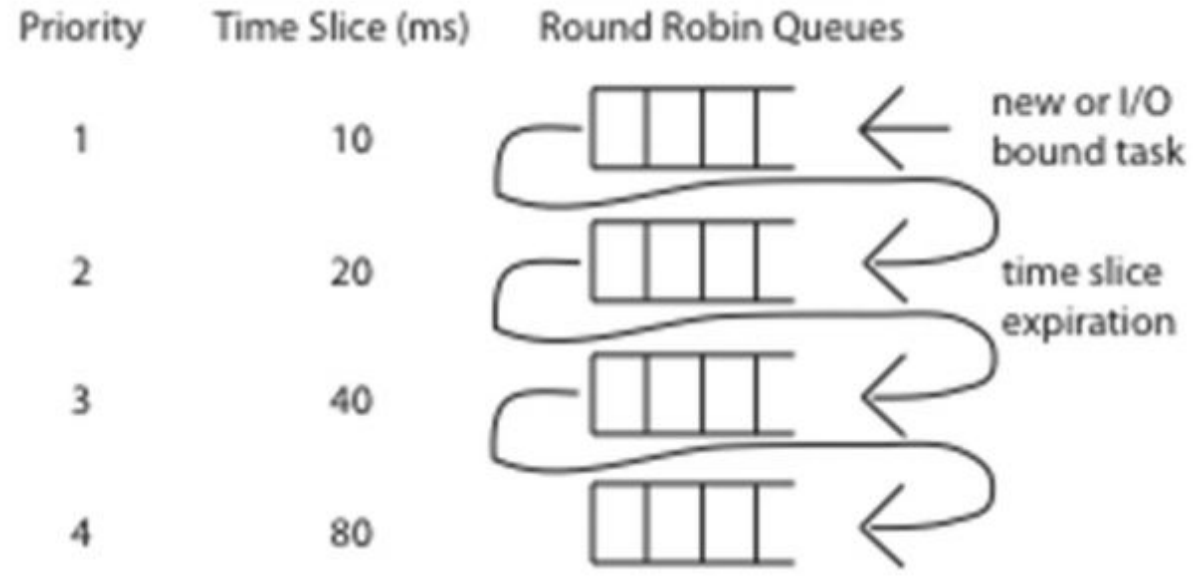
Il Round Robin è sempre equo? Equo inteso in senso di fairness
In realtà il problema sta nel fatto che ci sono processi con caratteristiche molto differenti

- **CPU bound**: lunghi utilizzi di CPUcon raro I/O, tempo completamento legato in maniera stretta al tempo passato sul processore. tipicamente usano tutto il quanto di tempo
- **I/O bound**: poca CPU e tanti I/O, legati alla velocità dei dispositivi, ad es. un processo che stampa un file

MFQ, Multi-Level Feedback Queue
Obiettivi:

- Responsività
- Basso overhead
- No Starvation
- Qualche task ha priorità alta/bassa
- Fairness su task dalla stessa priorità

Non è perfetto in nessuno di questi obiettivi! Usato su Linux, Windows, MacOS...
Strutturato come un **insieme di code Round Robin**, dove ogni coda ha priorità separata. Le code ad **alta priorità** hanno **quanti di tempo inferiori**, quelle a **bassa priorità** hanno **quanti di tempo maggiori**.
Lo scheduler pesca il **primo thread** della **coda a priorità più alta**, eseguendo un Round Robin in ogni coda.
Un task inizia dalla coda a priorità più alta, se finisce il quanto di tempo scala di un livello di priorità.



Problemi:

- **Starvation** se tutte le code a priorità superiore sono piene di processi I/O bound
  Un processo può "cambiare le sue abitudini", da I/O bound a CPU bound e viceversa
- Necessità di politiche per aumentare la priorità a processi I/O bound, e CPU bound in starvation
- MFQ viene solitamente usato in combinazione con priorità dinamiche

Es. windows: thread creato messo a priorità 8, incrementata o ridotta a seconda di come si comporta.
Priorità aumentata se:

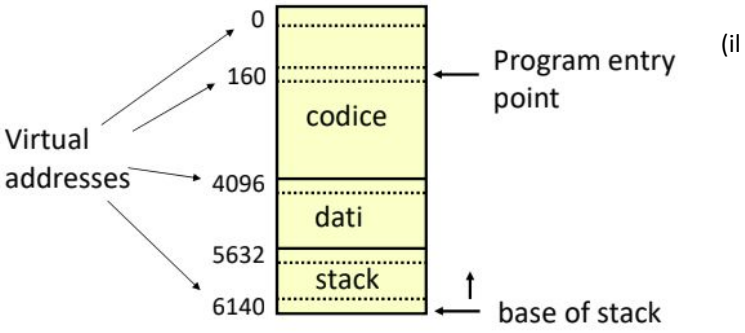
- thread **riattivato dopo operazione I/O** (disco +1, porta seriale +6, tastiera +8, scheda audio +8, ...)
- thread **riattivato dopo aver atteso su mutex/semaforo** (+1 se in background, +2 foreground)
- thread **non eseguito per un dato periodo di tempo**, la priorità va a 15 per due quanti di tempo

Priorità ridotta se il thread **usa tutto il quanto di tempo** (-1).
Quando una finestra ottiene il focus, il timeshare di quel processo è aumentato.

Riassuntino 29-30 07-scheduling.pdf

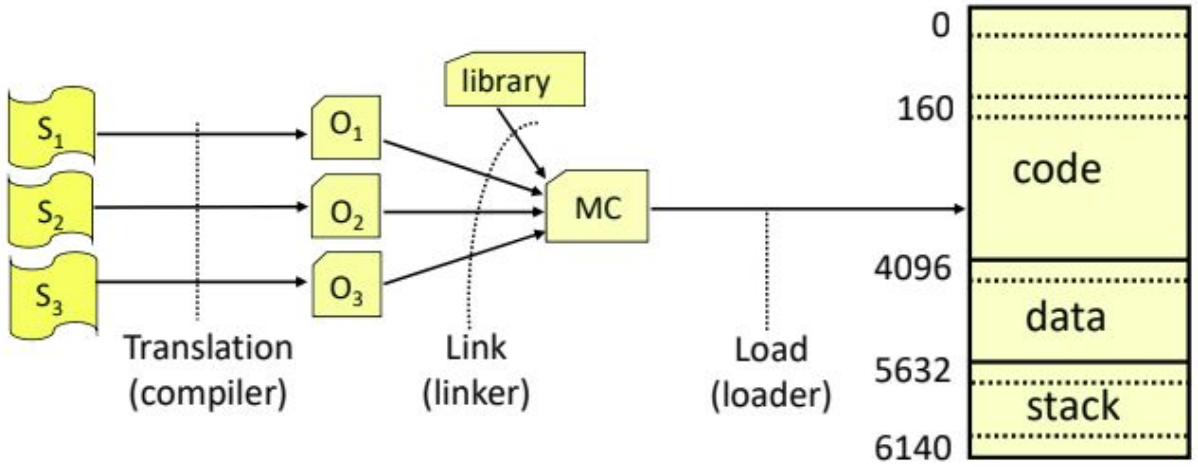
Traduzione degli Indirizzi
Come si converte un indirizzo virtuale in un indirizzo fisico? Molti metodi: Base & Bound, Segmentazione, Paginazione, Multilevel translation.

Un processo in memoria
Entry point: punto iniziale del processo, da dove parte l'esecuzione main tipo)
Dati: dati processo e dati heap
Lo stack **si allarga e restringe**, così come lo heap. Per questo lo stack è messo in fondo, l'heap sopra e tra loro c'è spazio vuoto.



(il

Dalla compilazione all'esecuzione



S_i: moduli sorgente, O_i: moduli oggetto, MC: eseguibile (file)
Ad esempio, un ciclo è tradotto con delle istruzioni di salt, in cui bisogna specificare l'indirizzo a cui saltare, ma **non è noto in compilazione**. La compilazione lascia in sospeso tutti i riferimenti esterni. I riferimenti, quindi, possono fare riferimento ad indirizzi virtuali compresi tra 0 e MAX.

Non possiamo, in generale, caricare a partire dall'indirizzo fisico 0 reale, quindi come fare?

- Anticamente **rilocazione statica**, cioè indico l'indirizzo iniziale e da lì lo carico
- **Rilocazione dinamica**: indirizzi virtuali, base bound, MMU traduce indirizzi da virtuali a fisici

Rilocazione statica degli indirizzi: **caricatore rilocante.** Una volta caricato un indirizzo quello non può più essere spostato.

Caricatore non rilocante
Praticamente copio-incollo il processo in una zona di memoria.
Se per esempio avevo un salto LOAD A, con A in indirizzo 4112, in compilazione diventa LOAD 4112. Ma che significa *LOAD 4112* se una volta in memoria A è a 14352. Quindi la rilocazione deve avvenire a lato di esecuzione.
Ci serve quindi hardware che permetta di tradurre gli indirizzi, cioè un oggetto fisico che prenda l'indirizzo virtuale, fare anche tutte verifiche di protezione e sicurezza, verificare i diritti e tradurre indirizzi.

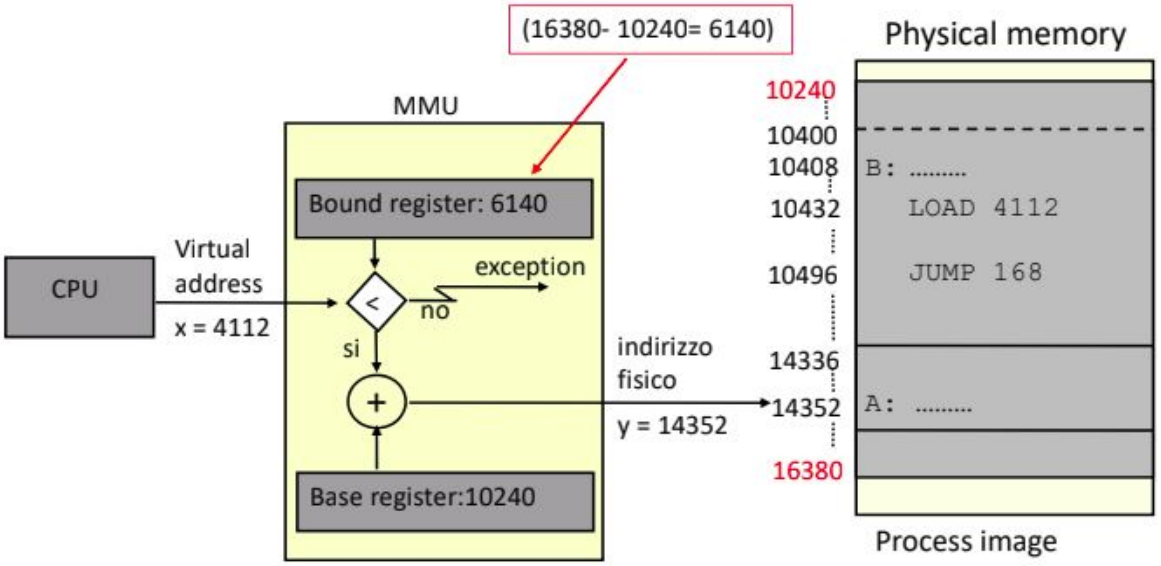
Address Translation Goals

- Protezione della memoria e condivisione
- Uso della memoria flessibile, con indirizzi anche sparsi
- Efficienza nel lookup di indirizzi e tabelle di traduzione compatte, non posso perdere tempo con traduzione lenta
- Portabilità

Address Translation Uses

- Isolamento dei processi
  Impedire che un processo intacchi la memoria di un altro processo o del kernel
- Efficienza nella comunicazione interprocesso, con zone di memoria condivise
- Segmenti di codice condivisi (ad es. librerie)
- Inizializzazione dei programmi, cioè far partire l'esecuzione prima che sia completamente in memoria
- Allocazione dinamica della memoria
  Allocare e inizializzare pagine su stack/heap su richiesta
- Gestione della cache
- Debug del programma, con i breakpoints e i watch
- Zero-Copy I/O, far comunicare dispositivo I/O e memoria utente direttamente
- File mappati in memoria, accederli con istruzioni load/store
- Memoria virtuale paginata, illusione di una memoria pressoché infinita, aiutata dal disco o dalla memoria su altre macchine
- Checkpointing e restart, cioè salvare trasparentemente una copia di un processo senza fermare il programma mentre avviene il salvataggio
- Strutture dati persistenti che sopravvivono al reboot di sistema
- Migrazione dei processi, cioè trasporto trasparente tra macchine
- Controllo del flusso di informazione, tenendo traccia dei dati condivisi esternamente
- Memoria condivisa distribuita

Virtual Base and Bounds



L'MMU prende l'indirizzo virtuale generato dalla CPU, controlla che sia inferiore al MAX (bound register, calcolato come differenza tra il registro massimo e il registro minimo della partizione di memoria dov'è contenuto il processo) e, in caso lo sia, trova l'indirizzo fisico sommando l'indirizzo minimo all'indirizzo virtuale.

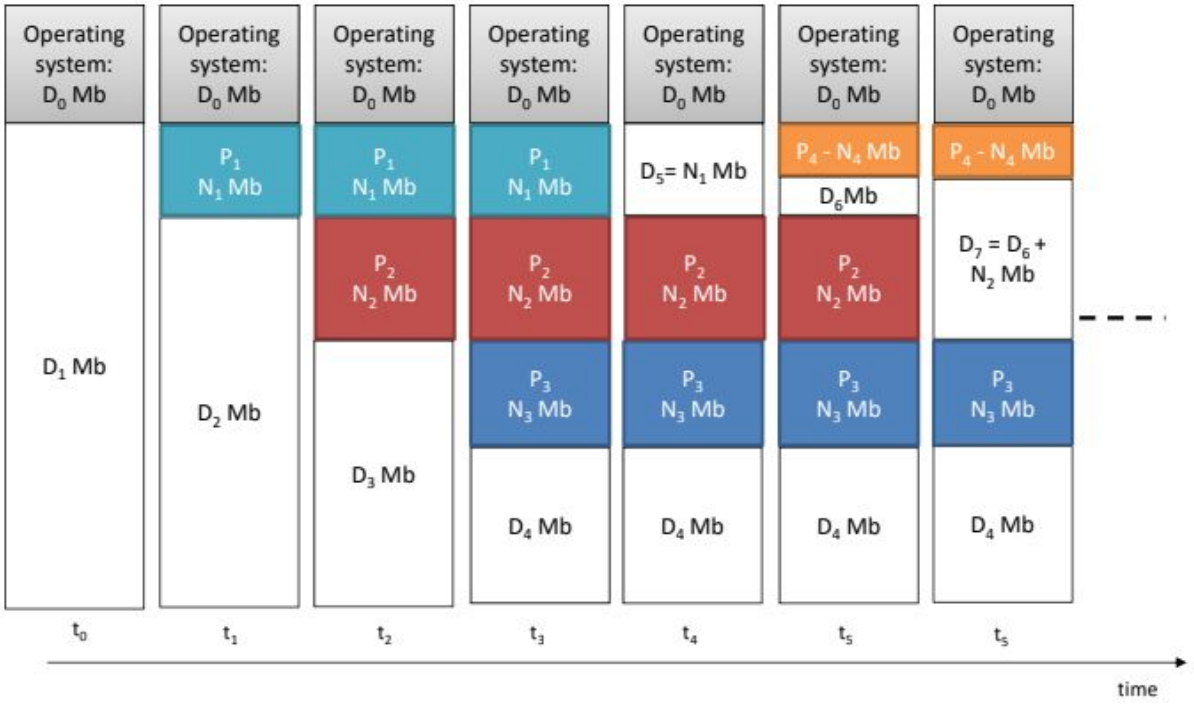
Pro

- Semplice
- Veloce (2 registri, sommatore, comparatore)
- Può rilocare nella memoria fisica senza cambiare processo

Contro

- Non può impedire ad un programma di riscrivere il proprio codice accidentalmente
- Non può far condividere codice e dati tra processi
- Non può far crescere stack/heap se necessario

Partizioni Variabili

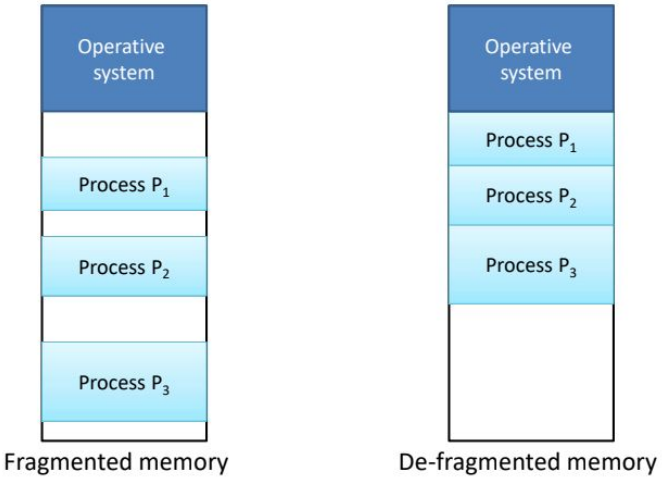


Allocazione di una nuova partizione

- **First fit**: tra tutte le partizioni libere **prendi la prima abbastanza grande** da poter allocare la nuova partizione
- **Best fit**: tra tutte le partizioni libere **prendi la più piccola abbastanza grande** da poter allocare la nuova partizione

Frammentazione

- **Frammentazione interna**: la memoria è allocata alla partizione ma non è usata/non è necessaria al processo. Succede se sono usate partizioni a dimensione fissa: es. ho partizioni da 10k, devo allocare 1k, ma posso solo allocare una partizione da 10k per tenerne 1k, i restanti 9k sono inutilizzati
- **Frammentazione esterna**: le partizioni libere sono troppo piccole per poter allocare, anche se la memoria libera totale potrebbe essere sufficiente. Succede con partizioni a dimensione variabile

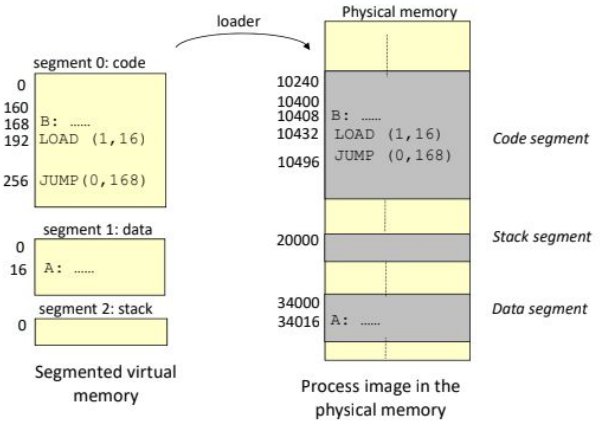


Segmentazione

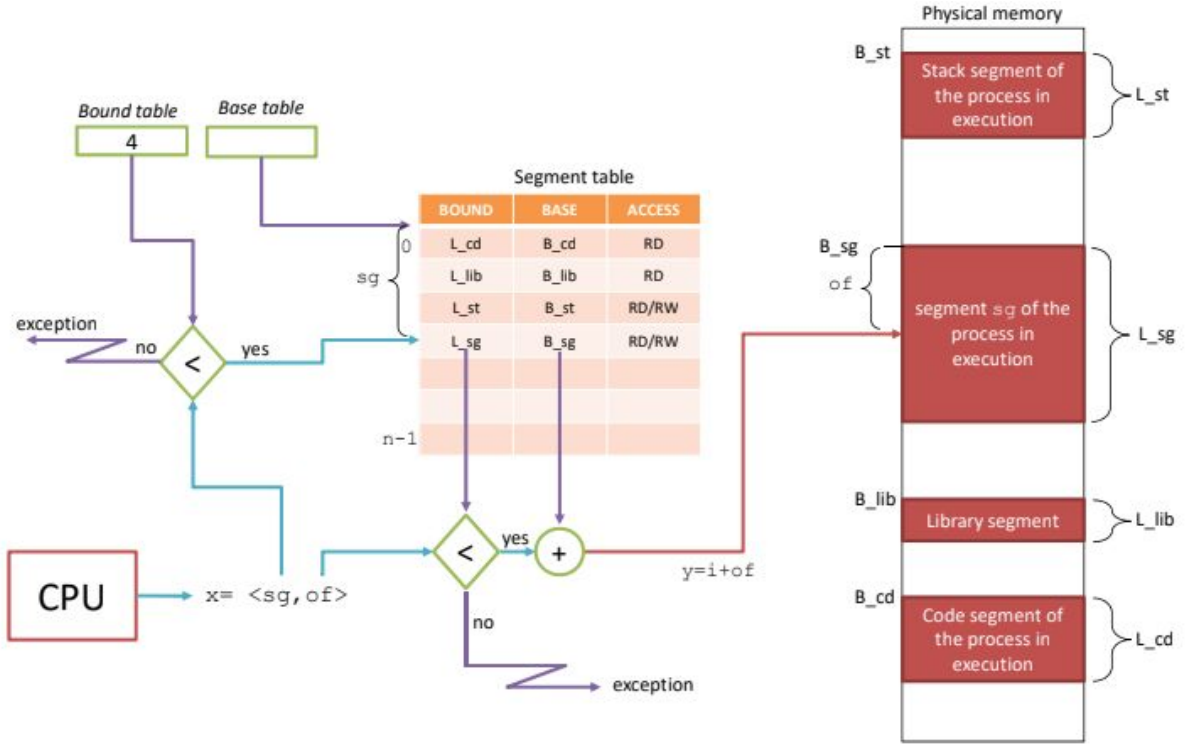
Un segmento è una **regione contigua di memoria** virtuale o fisica. Ogni processo ha una **tabella dei segmenti** nell'hardware. Ogni riga della tabella è un segmento.

I segmenti possono essere localizzati ovunque nella memoria fisica: inizio, fine, permessi d'accesso.

I processi possono condividere segmenti, con medesimo inizio e fine ma possibilità di avere diversi diritti d'accesso.



Traduzione degli indirizzi



UNIX Fork

Esegue una copia completa del processo.

I Segmenti consentono un'implementazione più efficiente:

- copia la tabella segmenti nel figlio
- setta i segmenti parent e figlio come sola lettura
- avvia processo figlio, torna al padre
- se parent o figlio scrive su segmente, si passa al kernel, che fa una copia del segmento e riprende l'esecuzione (**Copy on Write**)

Zero on reference

Allo stack e allo heap vengono allocati 0B. Quando il programma tocca uno dei due, segmentation fault al kernel. Il kernel alloca un po' di memoria, **la azzera** per evitare leak di informazioni, riavvia il processo.

Segmentazione

Pro

- Condivisione dei segmenti codice e dati tra processi
- Protezione segmenti codice da sovrascritture
- Fa crescere heap/stack trasparentemente quando bisogno
- Può rilevare quando è necessario il copy on write

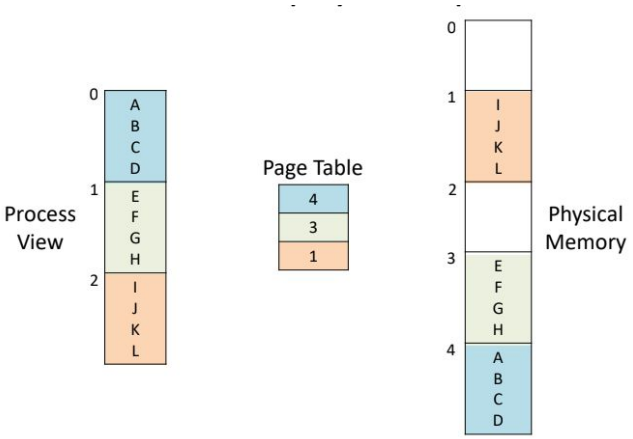
Contro

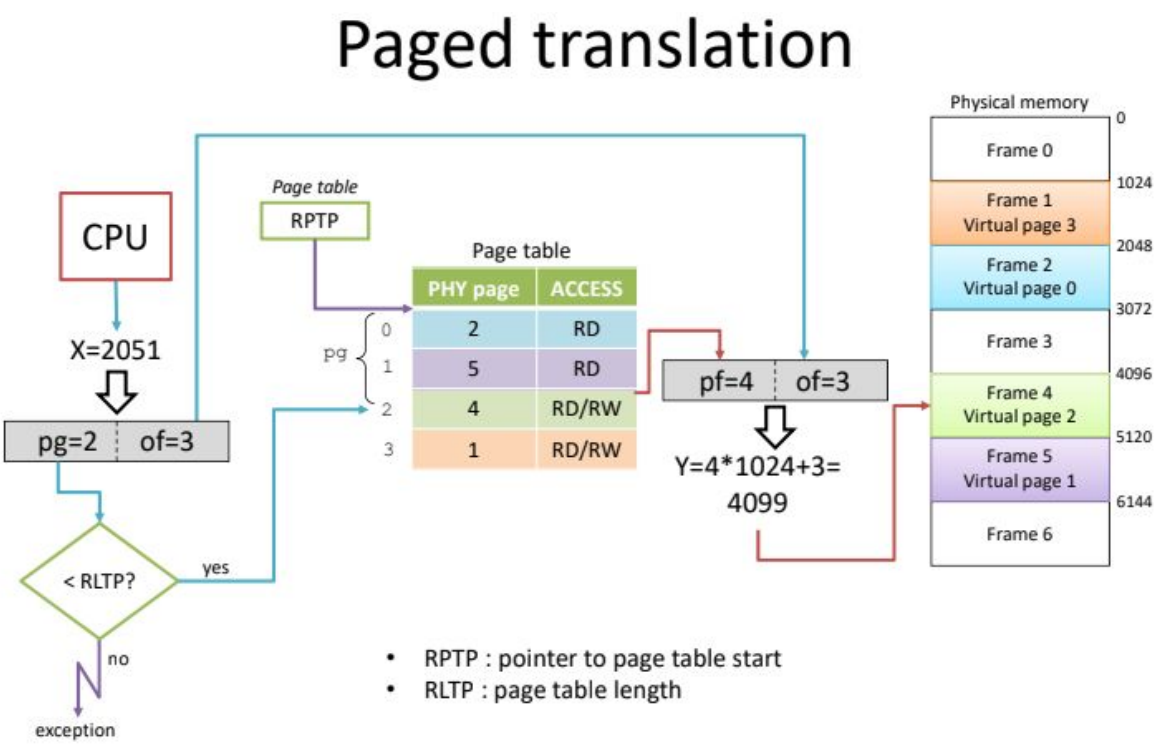
- gestione della memoria complessa, necessita di trovare spazio di una particolare dimensione
- necessità di riarrangiare la memoria ogni tanto per fare spazio ai nuovi segmenti o a segmenti che crescono (frammentazione esterna)

Traduzione paginata

Gestisce la memoria in unità dalla dimensione fissa, o **pagine**. Viene tenuta traccia delle allocazioni con una bitmap: es. 0011111100000001100, ogni bit rappresenta una pagina fisica, quindi trovare una pagina libera è facile.

Ogni processo ha la sua tabella delle pagine, memorizzata nella memoria fisica e con registri hardware: puntatore all'inizio della tabella e registro lunghezza.





Cosa deve essere salvato/ripristinato su un context switch di un processo?: Il puntatore alla tabella delle pagine, la sua lunghezza, e la tabella stessa nella memoria principale

Paging e Copy on Write
Possiamo condividere memoria tra processi? Le entrate di tutte e due le tabelle delle pagine puntano alle solite pagine. Necessità di una mappa che tiene traccia di quali processi puntano a quali pagine.
UNIX fork con copy on write

- copia la tabella delle pagine in un nuovo processo
- setta tutte le pagine come sola lettura
- su scrittura passa al kernel, copia pagine e riprende esecuzione

Paging e Fast Program Start
Posso avviare un processo prima che il suo codice sia nella memoria principale?

- Setta tutte le pagine come invalide
- Quando una pagina è riferita per la prima volta
    - passa al kernel
    - kernel porta la pagina in memoria
    - riprendi esecuzione
- Le pagine rimanenti possono essere trasferite in background mentre il programma viene eseguito

Multilevel Translation
Per poter realizzare le traduzione l'MMU ha bisogno di accedere alla tabella pagine, che sono quindi **imposte dal processore** e non dal sistema operativo, che quindi si deve adattare.
Il sistema operativo dovrebbe poter realizzare tab pagine come meglio crede, per questo motivo i SO utilizzano tabella e pagine imposte dal processore, ma **a livelli superiori implementano strutture dati proprie** per rappresentare le pagine in memoria principale. Queste strutture dati lo rendono indipendente e consentono la realizzazione di politiche specifiche. Il processore deve solo tradurre gli indirizzi virtuali. Il SO ha l'esigenza di creare un meccanismo intorno agli indirizzi che consenta al processore di trovare le informazioni nelle tabelle. Deve quindi poterle inizializzare e tutto.
La tabella pagine viene costruita per poter eseguire la traduzione e non per gestire memoria, i quali meccanismi sono introdotti dal SO stesso.
Albero delle tabelle di traduzione:

- Paginazione segmentata
- Tabelle delle pagine multilivello
- Paginazione segmentata multi-livello

Tutte e tra hanno **pagina dimensione fissa** come **unità** di livello inferiore: efficienza nell'allocazione di memoria, nel trasferimento tra dischi, più facile costruire il buffer di traduzione, lookup efficiente (fisico -> virtuale), granularità delle pagine per protezione/condivisione.

diversi approcci segmentazione paginata, tabelle a multilvello e combinazione delle due
Elemento base è sempre pagina, spazio virtuale è sempre spazio diviso in pagine di dimensione fissa.
Gruppi di pagine continue stanno assiene: allocazione efficiente, trasferimenti su disco efficienti

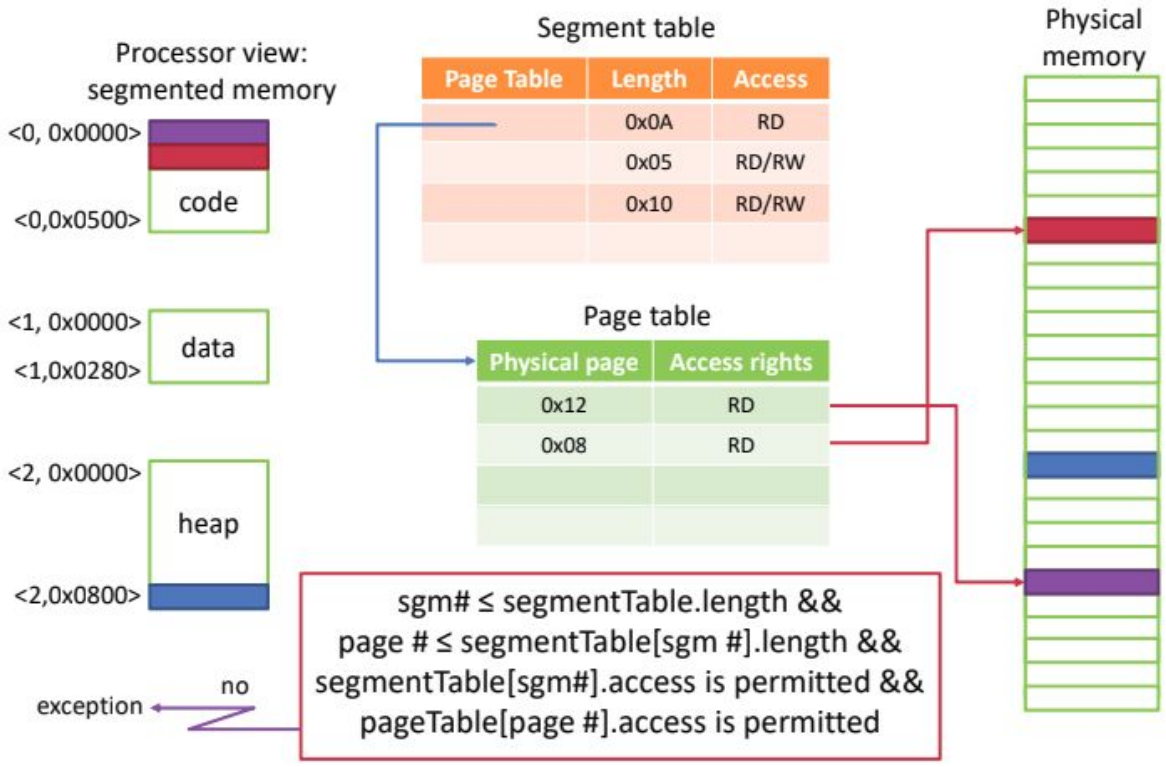
Segmentazione Paginata
La memoria dei processi è segmentata.
Ogni entrata della **tabella dei segmenti** ha:

- Puntatore alla tabella delle pagine
- Lunghezza della tabella delle pagine (cioè numero di pagine nel segmento)
- Permessi d'accesso

Ogni entrata nella **tabella delle pagine** ha:

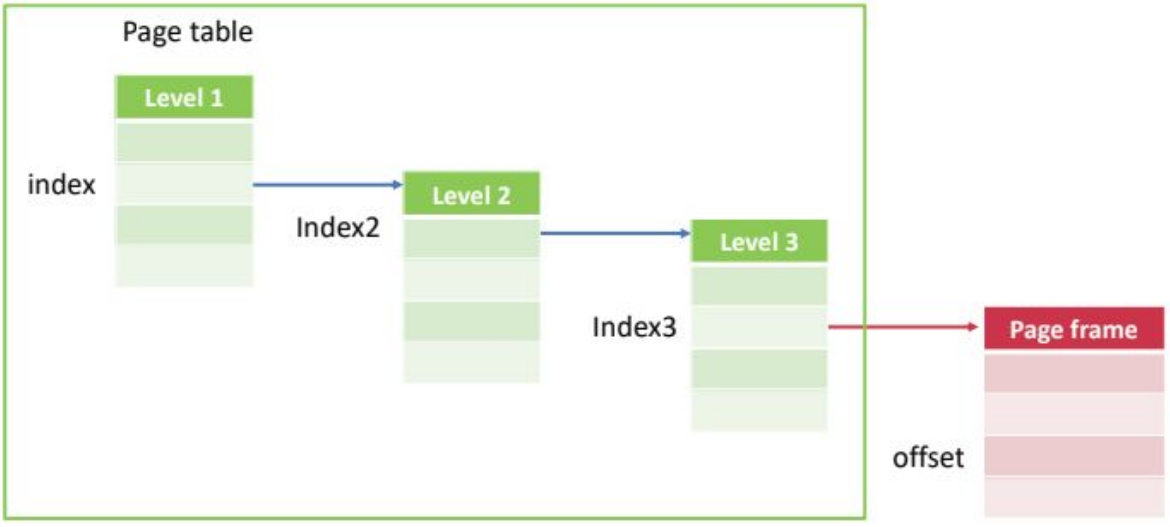
- Pagina
- Permessi d'accesso

Condivisione e protezione a livello di pagina o segmento.



Paginazione Multilivello

physical address=pageTable[index].pageTable[index2].pageTable[index3]|page offset



Traduzione Multilivello

Pro:

- Alloca/riempie solo le tabelle di pagine necessarie
- Allocazione memoria semplice
- Condivisione a livello di pagina o segmento

Contro:

- Due o più lookup per riferimento di memoria

Molti sistemi operativi tengono le proprie strutture dati per tradurre gli indirizzi

- Lista di oggetti di memoria (segmenti)
- Virtuale -> Fisico
- Fisico -> Virtuale

Tabella delle pagine inversa:

- Hash della tabella virtuale -> tabella fisica
- Spazio proporzionale al numero di pagine fisiche

La **tabella delle pagine inversa rappresenta stato di allocazione blocchi fisici, non virtuali**. Ne esiste una per l'intero sistema operativo ed è condivisa tra tutti i processi. Il sistema operativo può sapere quali sono i blocchi fisici liberi e occupati, quale processo possiede quale blocco e quale pagina in quale blocco.
Questo consente tutti meccanismi di manutenzione della memoria relative alle pagine.
Abbiamo proprio bisogno delle tabelle pagine multilivello dato che tutte info sono duplicate nel SO?
Sì, usando solamente tabelle inverse. es. IBM PowerPC, estremamente complicato a livello hardware.

Traduzione Efficiente degli Indirizzi

TLB translation lookaside buffer, cache all'interno dell'MMU:

- Cache delle pagine virtuali recenti
- Se la cache ha corrispondenza, la usa. Altrimenti passa alle tabelle di pagine multilivello

L'MMU ha una cache interna quindi contiene solo un sottoinsieme dei descrittori per poter tradurre indirizzi virtuali più utilizzati in un dato intervallo temporale.

```
Costo traduzione = costo della ricerca in TLB + (probabilità(assenza in TLB)*costo ricerca pagina in tabella)
```

Se la traduzione non è nella TLB allora passa al kernel, che calcola la traduzione e la carica nella TLB. Il kernel usa qualsiasi struttura dati voglia, anche le tabelle di pagine multilivello.

Se la cache non ha la pagina cercata (**cache miss**), l'MMU va a cercare lei stessa il descrittore di pagina e ciò costa circa 20 istruzioni macchina.
Se invece la lasciamo gestire al SO, l'MMU deve sollevare un'eccezione al processore, che fa partire l'interruzione, esegue l'handler ecc.... Una gestione del cache miss in questo caso ha costo di 100-1000 istruzione macchina, non è efficiente.

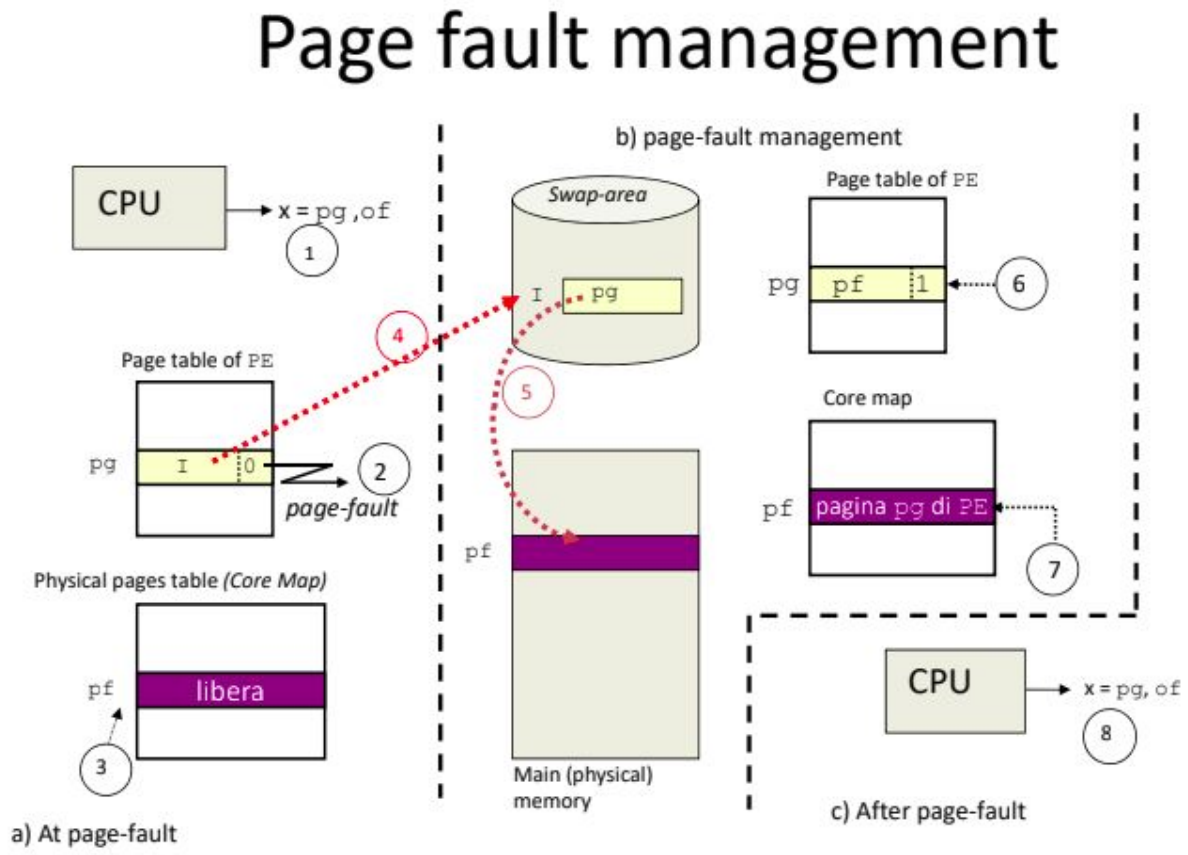
Nella commutazione di contesto si **annulla la cache della MMU**, perché se continuo ad usarla il nuovo processo potrebbe accedere alle pagine del processo precedente. Un'altra possibile soluzione è aggiungere l'identificatore del processo per le varie entry della TLB.

Superpagina

Una possibile entry nella TLB che rappresenta un gruppo contiguo di descrittori di pagine.

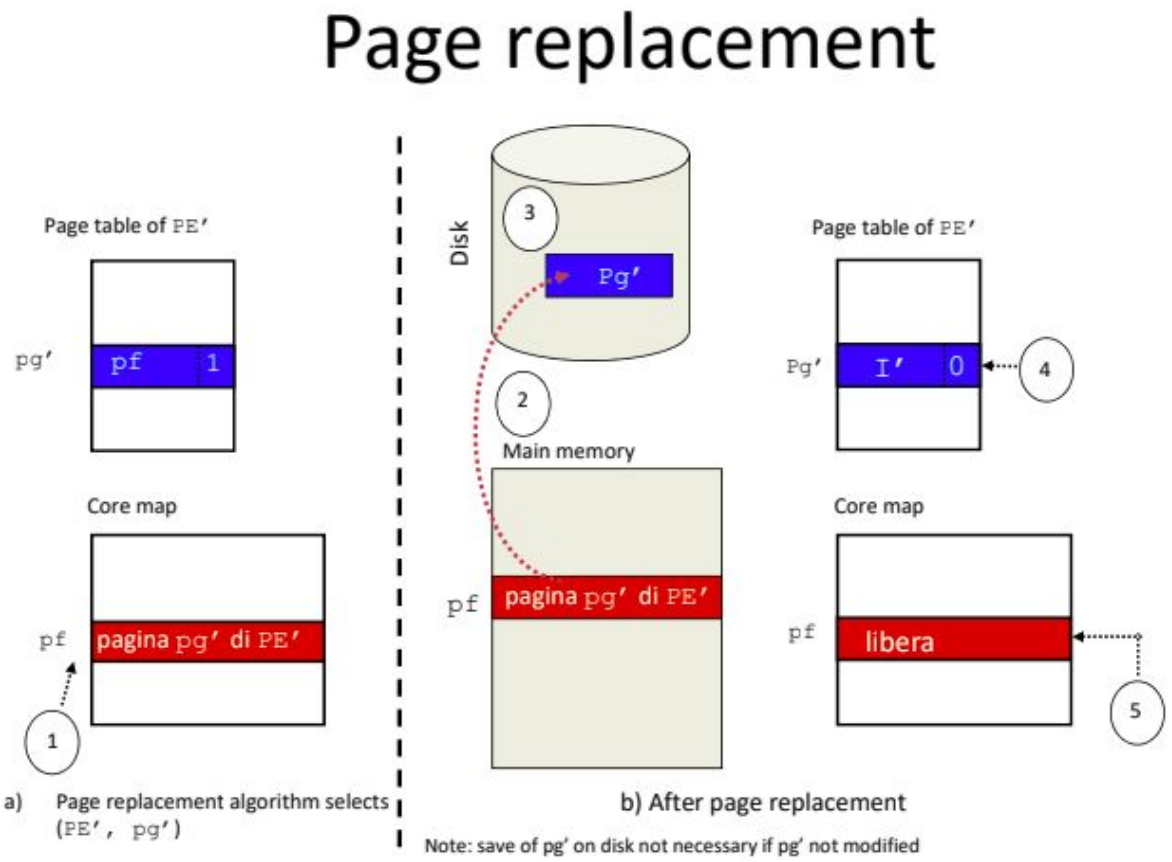
Paging On-Demand

Il campo della pagina contiene il numero di pagina fisica se P=1 (presenza), **altrimenti contiene l'indirizzo sul disco**. Questo non solo per tradurre gli indirizzi come sempre, ma anche per **estendere memoria virtuale del processo** usando la memoria fisica come cache



Con il paging on-demand, il page fault viene gestito così:

1. TLB miss
2. Vado nella tabella delle pagine
3. Page Fault (pagina non valida nella tabella delle pagine)
4. Passo al Kernel
5. Converte l'indirizzo in file+offset
6. Alloca la pagina (eliminando un'altra pagina se necessario)
7. Inizializza il disco per leggere il blocco della pagina
8. Interrupt del disco quando DMA completo (Direct Memory Access)
9. Marchio la pagina come valida
10. Riprendo il processo all'istruzione che ha causato il fault
11. TLB miss
12. Navigo la tabella delle pagine per trovare la traduzione
13. Eseguo l'istruzione



Allocare una pagina

- Seleziono la pagina da eliminare
- Trovo tutte le entries delle tabelle di pagina che riferiscono la pagina da eliminare (se la pagina è condivisa)
- Setto ogni entry trovata come non valida
- Rimuovo tutte le entry nel TLB (cope della ora non valida entry nella tabella di pagina)
- Scrivo modifiche su disco, se necessario (es. la pagina è stata modificata, o i suoi permessi)

Come so se una pagina è stata modificata?

Ogni entry della tabella delle pagine ha una specie di registro:

- Pagina modificata? Settato dall'hardware sull'istruzione STORE sulla pagina, sia nel TLB che nella entry nella tabella delle pagine
- Pagina usata? Settato dall'hardware su istruzioni LOAD o STORE sulla pagina, nella tabella delle pagine quando c'è un TLB miss

Può essere resettato dal kernel, quando i cambiamenti sono scritti su disco e per tracciare se l'utilizzo della pagina è avvenuto recentemente.

Alcune architetture dei processori non mantengono un bit di modifica o di utilizzo nella entry della tabella delle pagine (maggiore complessità). Il SO può emulare questi bit.

Emulare bit modifica

- Setta tutte le pagine pulite come sola lettura
- Sulla prima scrittura, porta il page fault al kernel
- Dalle informazioni nella core map il kernel sa cosa fare
- Kernel setta il bit di modifica marcando la pagina come lettura e scrittura

Emulare bit utilizzo

- Setta tutte le pagine inutilizzate come invalide
- Sulla prima lettura/scrittura, porta il page fault al kernel
- Dalle informazioni nella core map il kernel sa cosa fare
- Kernel setta il bit di utilizzo marcando la pagina come lettura o lettura/scrittura

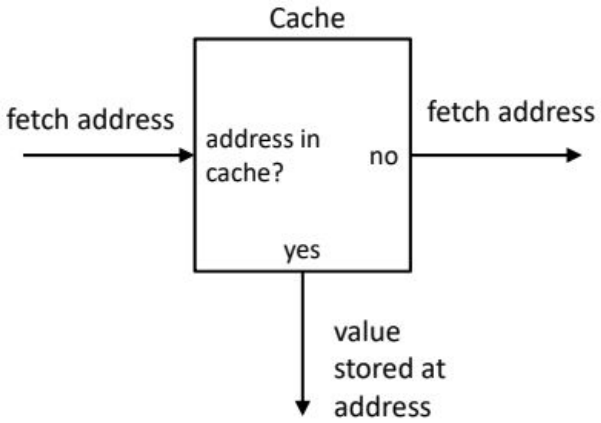
Caching e Memoria Virtuale

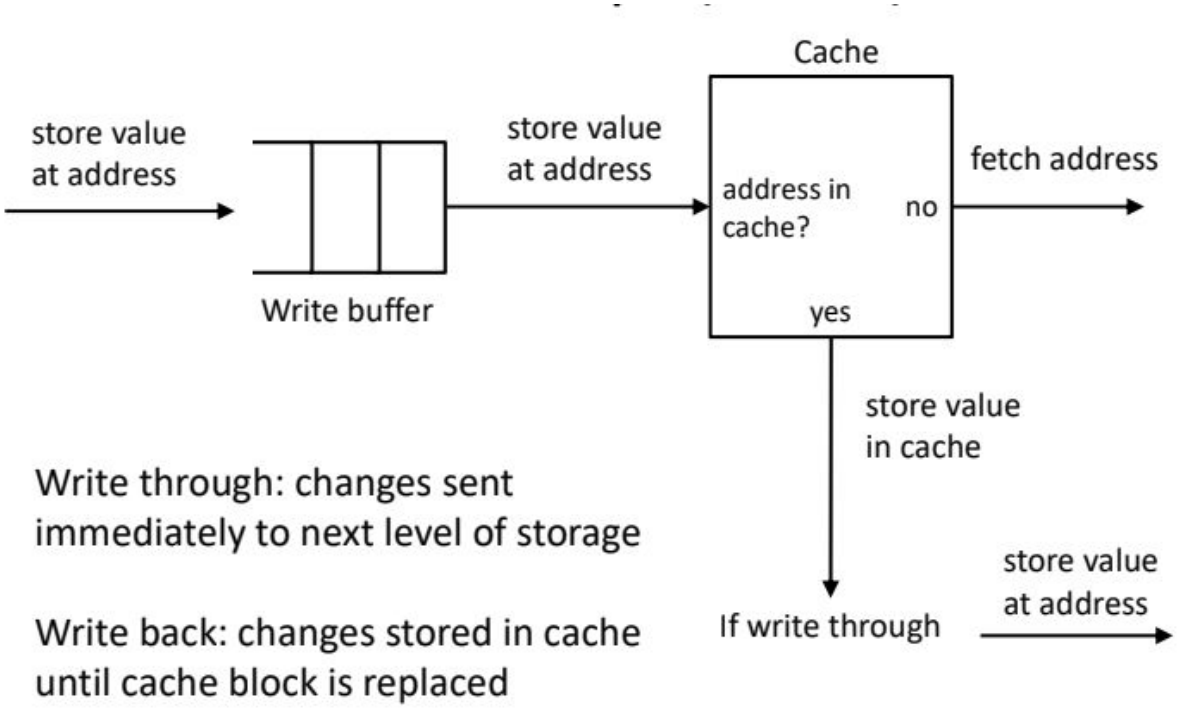
Cache: **copia dei dati più veloce da accedere che degli originali**. **Hit** se la cache ha la copia, **Miss** se non ce l'ha.

Cache block: **unità di memorizzazione** della cache (locazioni di memoria multiple)

Località temporale: **programmi tendono a riferire le stesse aree di memoria più volte in un dato periodo di tempo**. Ad es.: le istruzioni in un ciclo.

Località spaziale: **programmi tendono a riferire le aree vicine fra loro**. Ad es.: i dati in un ciclo

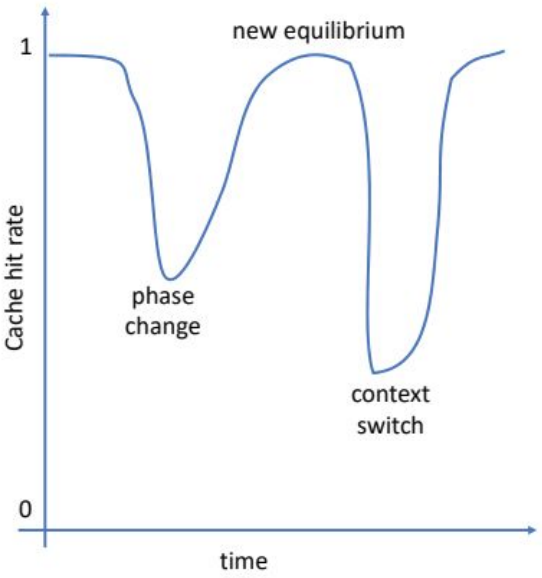


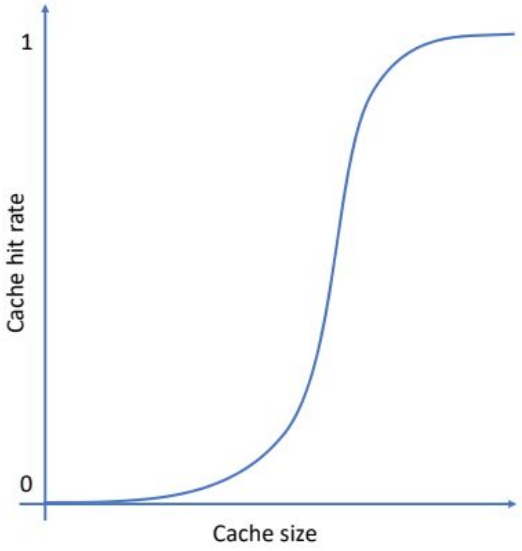


Modello Working Set

Working Set: collezione di locazioni di memoria che devono essere in cache per avere un tasso di hit ragionevole

Trashing: quando un sistema ha una cache troppo piccola (troppi miss)





Modello a Cambiamento di Fase

I programmi cambiano il proprio working set, e pure i cambi di contesto.

Politiche di Rimpiazzo della Cache

Come scegliere l'entry da rimpiazzare quando avviene un cache miss? Assumendo che la nuova entry venga usata più frequentemente nel prossimo futuro.

Obiettivo della politica: ridurre i cache miss, migliorare le performance e ridurre la possibilità di avere performance pessime.

Una politica semplice

Random? Una pagina a caso

FIFO? Rimpiazzo la pagina che è nella cache da più tempo. Cosa potrebbe andare storto?

Il caso peggiore per FIFO è un programma che scorre una memoria più grande della memoria cache principale.

Reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

MIN (ideale, ottimale)
Rimpiazzo la pagina che **non verrà usata** per il tempo più lungo (cioè la cui **distanza futura** è massima). Ottimo: se tolgo una pagina usata prima avrò un page fault (quindi un cache miss) prima.

Least Recently Used LRU
Rimpiazzo la pagina che **non è stata usata** per il tempo più lungo (cioè la cui **distanza passata** è massima).
Approssimazione di MIN

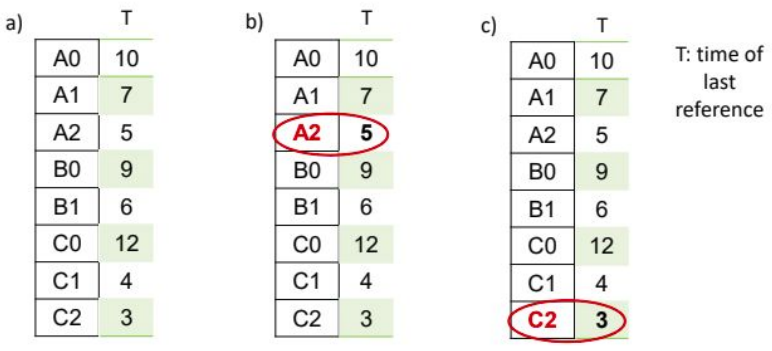
Not Recently Used NRU
Rimpiazzo una delle pagine che non sono state usate recentemente. Rilassa le condizioni dell'LRU.
Approssimazione dell'LRU, più facile da implementare. Es.: second chance, working set algorithm

Clock Algorithm
Periodicamente analizza tutte le pagine.
Se la pagina è inutilizzata, la richiama.
Se la pagina è usata, la marchia come inutilizzata

Page Replacement Second-Chance
Cicla su tutte le pagine. Se deve operare una sostituzione, rimuove una pagina e carica la pagina richiesta. Il lavoro di ricerca lo svolge l'algoritmo ad orologio.
Generalizzazione **Nth-chance**. Periodicamente cicla tra tutte le pagine.
Se una pagina non è stata usata in nessuno dei precedenti N controlli, la richiama.
Se una pagina è usata, la marchia come usata e la setta come attiva nel ciclo corrente.

Page replacement locale e globale
Su quale insieme di pagine opera l'algoritmo? Se il processore richiede di caricare una pagina e ne deve essere rimossa una, meglio **rimuovere una pagina di un processo qualunque o dello stesso processo**? **Pagina caricata sostituisce una mia pagina o una pagina qualsiasi**? Non c'è risposta univoca.
Buona parte algoritmi possono essere implementati in entrambi i modi:

- **Globali**, rimuove una pagina indipendentemente dal processo.
  Non rispettano il possessore della pagina da rimuovere.
  La "distanza dal passato" si basa sul tempo globale (**orologio assoluto**)
  Può causare trashing in processi più lenti
- **Locali**, rimuove una pagina appartenente processo che ha causato page fault.
  Limita il trashing di processi lenti
  Distanza dal passato basata sul tempo relativo del processo (**tempo che il processo ha passato in esecuzione**)



a) Initial configuration
b) Page replacement with a local polity (WS, LRU, sec. chance)
c) Page replacement with a global policy (LRU, sec. Chance)

Processi avanzano con velocità diverse
immaginiamo alg globale lru (least recently used), rimuove pag usata meno di recente. Molto facile che pag rimossa (quella usata meno di recente) appartenga a processo lento. Un effetto di untilizzo indiscriminato alg globali è che tengono a penalizzare processi lenti anche perché, rimuovendo pagine loro, è più facile che causino page fault

Con alg globali non ci preoccupiamo dimensione insieme residente, perché alg globali automaticamente caricano pagine richieste e definiscono i maniera automatica inseieme residente

Sis moderni tendono a usare alg locali
Essi hanno una conseguenza molto forte sul sistema: stabilire per ogni processo quanti blocchi fisici poter utilizzare e sono fissi, l'insieme residente è rigido. Sostituire e caricare pagine all'interno dell'insieme residente.

lru richiede clock interno e tempo per marcare pagine, sostituito da second chance

alg globale toglie pagina tra tutti i processi in memoria, locale va a rimuovere pagina solo del processo che ha causato page fault

Algoritmo del Working Set
Working Set: non è semplicemente l'insieme delle pagine che un processo sta usando, ma un **gruppo di pagine abbastanza ampio da contenere ciò che serve al processo per un tempo ragionevole**. Due definizioni possibili:

- **Insieme delle pagine riferite negli ultimi k accessi in memoria** del processo
  Difficile da implementare
- **insieme delle pagine riferito nell'ultimo periodo di tempo T**

Se un processo non cambia la porzione di codice in esecuzione la dimensione del working set tende a stabilizzarsi.
Ogni processo ha un insieme di pagine fisiche assegnate, esegue una politica locale per gestire le proprie pagine fisiche. Va a rimuovere pagine e caricarne altre sempre all'interno del pool di pagine fisiche che gli sono state assegnate.
Insieme residente: pagine effettivamente in memoria.
Vorremmo che in un determinato istante **l'insieme residente sia pari al working set**, ma non è sempre possibile: potremmo avere **pagine nel working set non ancora caricate** o **pagine residenti che sono uscite dal working set**. Obiettivo dell'algoritmo è far combaciare il più possibile l'insieme residente e working set.
Viene eseguito preventivamente in modo da avere pagine libere per gestire velocemente page fault.

L'algoritmo definisce il Working Set come la collezione di pagine riferite nell'ultimo periodo di tempo T, parametro. Per ogni pagina:

- **bit R**, di "riferimento" o "uso". Indica se la pagina è stata riferita dall'ultima volta che è stata analizzata dall'algoritmo
- **TLR**, un'approssimazione del tempo in cui è stata riferita la pagina l'ultima volta. Ogni volta che il time tick scatta, l'algoritmo resetta il bit R e aggiorna il TLR (time of last reference) di ogni pagina
- **età** di una pagina, definita come la differenza tra il tempo corrente e il tempo di ultimo riferimento

Quando viene eseguito (che sia al page fault o periodicamente):

- Per ogni pagina gestisce i valori di R, TLR ed età
  Se R = 0 allora *età = tempo_corrente - TLR*
  Se R = 1 allora *TLR = tempo corrente*, cioè *età = 0*, e resetta R
- Le pagine con età < T (riferite nell'ultimo periodo T) sono nel working set e (se possibile) non vengono rimosse

```
For each page: {
  if (R==0)
     age = current_time – TLR;
  else if (R==1) {
     TLR= current_time; R=0; age=0;
  }
  if ( (age>T)
     removes the page
}
-----------------------------------

if (age<=T for each page)
  removes the page with higher age
```

WSClock
Considera solo le pagine nella memoria principale (più veloce che scansionare tutta la tabella delle pagine).
Le pagine vengono disposte in una lista circolare. Al page fault si cerca una pagina fuori dal working set, meglio se non è "sporca". Se la pagina selezionata è sporca, la pagina è salvata prima della rimozione.
Slide 38-40 di 09-caching.pdf per il funzionamento.

Nella pratica, working set e tutti gli algoritmi per il rimpiazzo delle pagine sono eseguiti preventivamente, garantendo così la presenza di pagine fisiche libere in caso di page fault (per velocizzare il sistema).
Dettagli nei casi di studio (Unix e Windows)

Algoritmi dinamici: **Page Fault Frequency**
Osserva il comportamento dei processi dal punto di vista dei page fault generati.
Se il numero supera una certa soglia A, cioè genera troppi page fault, allora è indice che l'insieme residente è piccolo: alloco più memoria.
Se è minore di un'altra soglia B allora significa che l'insieme residente è più che sufficiente, ha in memoria più pagine del working set, posso togliere un po' di memoria.

Cosa succede se aumento i processi? Cioè la dimensione di tutti i working set è più grande della memoria fisica? Avrò per forza un insieme residente più piccolo del working set e tanti page fault.
Situazione di **trashing**, il sistema collassa: aumentando i processi fino ad un certo punto aumento il troughput, ma dopo una certa soglia il sistema collassa perché passa più tempo a gestire page fault che a far avanzare i processi.

Dove sono memorizzate le pagine
Ogni processo ha un segmento gestito tramite un file su disco:
- Segmento codice -> Porzione del codice dell'eseguibile
- Segmenti dati, heap e stack -> file temporanei
- Librerie condivise -> file codice e file dati temporanei
- File mappati in memoria -> File mappati in memoria
- Quando il processo termina elimino i file temporanei.

Ciò fornisce l'illusione ai programmi di avere memoria infinita.

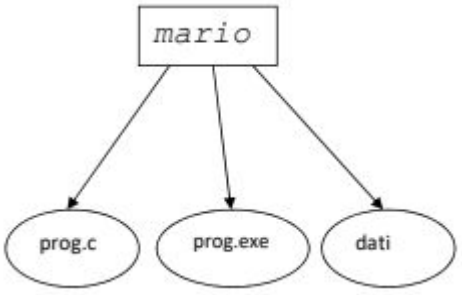
Slide 50-70 di 09-caching.pdf per i casi di studio.

File Systems
Il File System è l'astrazione che sta al di sopra dei dispositivi ed in alcuni casi, come in unix, ha un forte ruolo di astrazione per tutto ciò che è presente nel computer.
il più importante dispositivo che astrae è quello di memorizzazione, cioè i dischi (magnetici, flash…)
I dispositivi di memorizzazione forniscono:
- spazio di **memorizzazione persistente** che di solito sopravvive tra le accensioni
- **tecnologie** di memorizzazione **diverse dalle RAM** (memoria principale), cioè impongono restrizioni. Una delle caratteristiche principali è che l'**informazione** non è **accedibile** a byte ma **a blocchi**. Questo è vero sia per dischi magnetici che flash
- **grandi capacità a basso costo**
- **performance relativamente basse**
  Una lettura necessita di 10-20 milioni di istruzioni del processore (ecco perché è importante l'attesa dei processi che leggono)

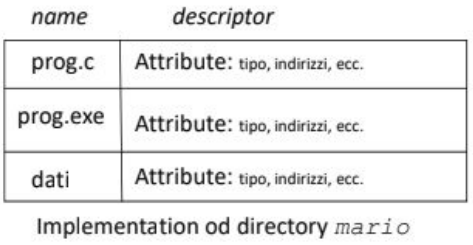
Il file system opera come illusionista per nascondere le limitazioni del disco fisico
Il disco è persistente ma nell'utilizzo **possono sorgere problemi che danneggiano la persistenza** (ad es. crash durante la scrittura, che potenzialmente distrugge o rende inconsistente il contenuto del disco e delle strutture dati che indicano dove sono memorizzati dati).
Il file system FAT non ha questa caratteristica, **un crash poteva devastare l'integrità disco**.
Oltre ai crash, **col tempo alcuni blocchi possono deteriorarsi**, smagnetizzarsi. Il file system deve mappare queste zone inaccessibili ed evitarle.
Naming: viene assegnato un nome ai dati invece di usare i blocchi numerati, questo per rendere raggiungibili i singoli byte invece del blocco
Performance: viene sfruttata la cache e viene ottimizzato il posizionamento dei dati e l'organizzazione delle strutture dati.

Il file system è organizzato secondo una **struttura gerarchica** (directory, sottodirectory, file).
Un file è una **collezione di dati con nome**, sequenza (o insieme di sequenze) di byte.

- Directory: gruppo di file nominati o di sottodirectory
  Mappa dal nome del file alla locazione dei metadati del file
- Path: stringa che identifica univocamente file o directory
- Links
  - Hard link: collegamento da nome alla posizione dei metadati
  - Soft link: collegamento da nome a nome alternativo
- Mount: mappatura dal nome in un file system alla root di un altro



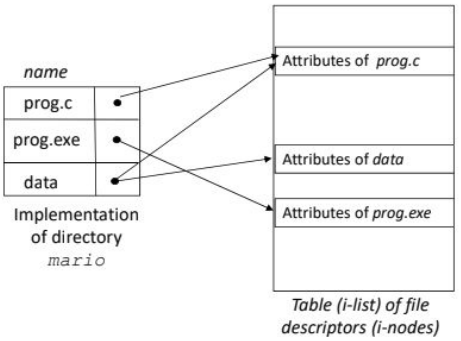
Directories
Ogni file e directory **appartiene ad una directory**. Ogni directory è una struttura dati che collega il nome del file ai suoi attributi, ad es.: grandezza del file, indirizzo su disco, diritti di accesso, ...



Implementation od directory `mario`

Una possibile implementazione (FAT): la directory è una tabella e associa ogni nome di file al suo descrittore di file (che contiene tutte le informazioni sul file)

Implementazione delle directory in unix
La directory è una tabella che include i riferimenti ai descrittori di file (**i-node**), che sono memorizzate in una struttura dati separata su disco.
L'i-node **contiene tutti i metadati** del file.



Implementation of directory `mario`
Table (i-list) of file descriptors (i-nodes)

Accesso ai file

Operazioni di accesso ai file: scrivere/leggere un record logico su/da file

Per eseguire queste operazioni su un file è necessario leggere gli attributi del file, come l'indirizzo su disco dei record logici, i permessi...

Essendo **molto costoso leggere questi dati da disco ad ogni accesso**:

- vengono letti una sola volta **prima di accedere il file**, con la chiamata di sistema **open**
- la chiamata di sistema **close** è necessaria per deallocare quelle informazioni dalla memoria

Metodi di accesso

Il metodo di accesso è indipendente dal dispositivo fisico e dal metodo di allocazione dei file sul dispositivo

L'accesso può essere:

- **Sequenziale**

  Il file è una sequenza di record logici [R_1, ..., R_N]: per accedere ogni record R_i è necessario prima accedere i precedenti i-1 record:

  Operazioni di accesso:
  - readnext: legge il prossimo record logico
  - writenext: scrive il prossimo record logico

    Ogni operazione (read/write) posiziona il puntatore d'accesso sul prossimo record logico
- **Diretto**

  Il file è una collezione di record logici {R_1, ..., R_N}: l'utente può accedere direttamente ciascun record logico

  Operazioni di accesso:
  - read(f, i, &V): legge il record logico i del file f e memorizza nel buffer V
  - write(f, i, &V): scrive il contenuto del buffer V nel record logico i del file f

UNIX File System API

create, link, unlink, mkdir, rmdir: crea file, link to file, remove link, crea e rimuove directory

open, close, read, write, seek: apre e chiude file, leggere/scrive, resetta la posizione

fsync: forza la scrittura delle modifiche in cache su disco

La **open** è uno strumento potente: apre il file e ritorna il descrittore, ma ha una miriade di opzioni tra cui

- se il file non esiste, errore
- se il file non esiste, lo crea e lo apre
- se il file esiste, errore
- se il file esiste, lo apre
- se esiste ma non è vuoto, lo svuota e lo apre
- se esiste ma non è vuoto, errore
- ...

Accesso ai Dispositivi

Questo livello gestisce l'accesso ai dispositivi e l'interfaccia comune, così che con le system call gli utenti possano avere un modo più o meno universale per accedere ai dispositivi: l'equivalente di accedere ai file, in termini di aprire/chiudere e leggere/scrivere.
In Unix tutti i dispositivi vengono astratti come file.

Due sottolivelli di accesso

- Parte inferiore **dipendente dall'hardware**: a livello più basso ogni dispositivo ha le sue specificità (mouse, scheda di rete, stampante...). Scritta dal produttore del dispositivo (driver)
- Parte superiore **indipendente dall'hardware**: bufferizzazione, caching, interfaccia, valgono un po' per tutti i dispositivi. Integrata nel S.O.

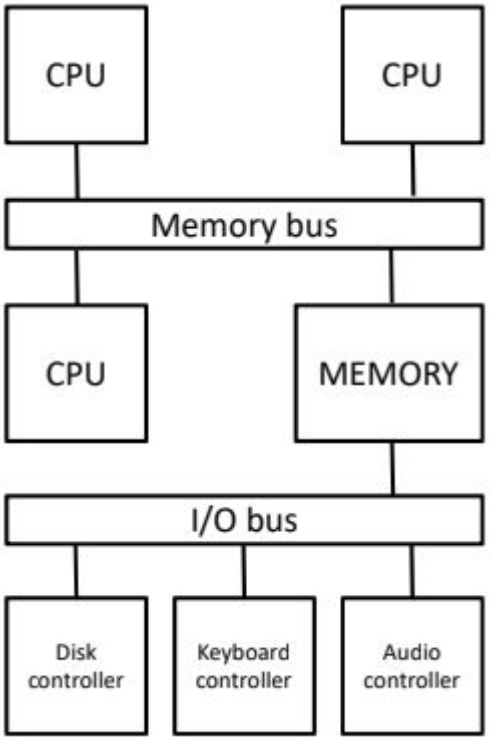
Per l'accesso, il file system:

- definisce uno spazio di nomi per i dispositivi
- implementa le politiche di caching
- è indipendente dall'hardware

mentre il driver del dispositivo:

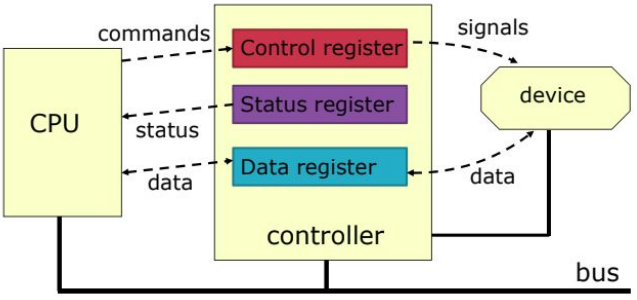
- si interfaccia con l'hardware (HW-dependent)
- gestisce la sincronizzazione con il dispositivo
- gestisce il trasferimento di dati da/a dispositivo/processo
- gestisce gli errori del dispositivo

Memory Mapped I/O



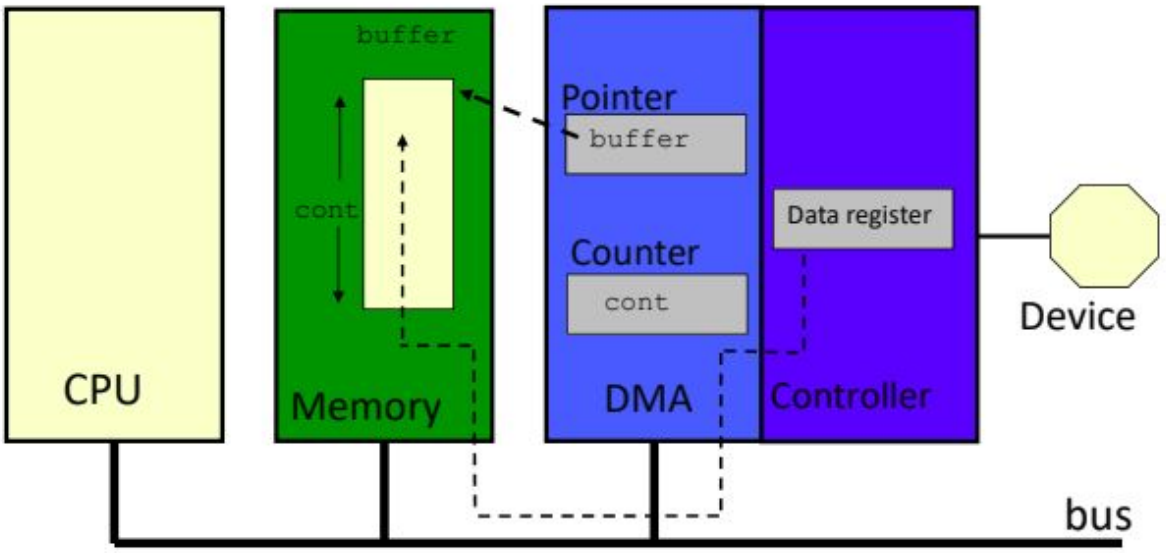
I registri interni del dispositivo sono mappati in locazioni di memoria: le operazioni di lettura e scrittura sono redirette al controller del dispositivo.

Un semplice controller



Direct Memory Access (DMA)
Molti dispositivi trasferiscono dati in massa, come dischi, stampanti.... Questi trasferimenti sono inefficienti se gestiti dalla CPU a livello di byte. DMA può gestire trasferimenti senza la CPU:

- il driver programma la DMA
- una volta che il trasferimento è completato, il DMA genera un'interruzione
- l'interrupt fa riprendere il driver che conclude l'I/O



Dispositivi di archiviazione
Dischi magnetici:

- dati raramente corrotti
- grandi capacità a basso costo
- accesso casuale a livello di blocco
- basse performance su accesso casuale
- alte performance ad accesso sequenziale

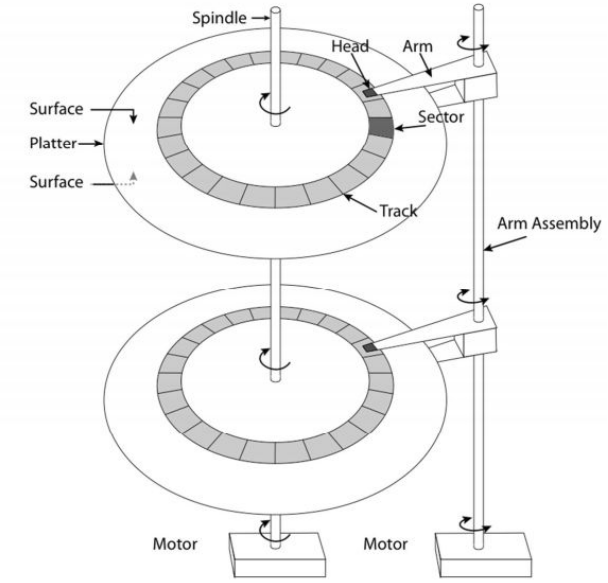
Memorie flash:

- dati raramente corrotti
- capacità a costo intermedio (2x dischi magnetici)
- accesso casuale a livello di blocco
- Buone performance per le letture, peggiori per scritture casuali

Dischi magnetici

Tracce

- **Larghe 1 micron** (l'occhio umano ha risoluzione 50 micron)
  Ce ne sono circa 100k su un disco da 2.5"
- Sono separate da **zone di guardia inutilizzate**, che riducono (ma non eliminano) la possibilità di corruzione durante la scrittura
- La lunghezza varia all'interno del disco:
  - all'esterno: più settori per traccia, larghezza di banda maggiore
  - il disco è organizzato per regioni di tracce con lo stesso numero di settori per traccia
  - solo la metà esterna del raggio è usata



Settori

I settori contengono sofisticati codici di correzione degli errori: il magnete sulla testina del disco ha un'area più larga della traccia, e nasconde le corruzioni grazie alla scrittura delle tracce adiacenti.

- **Sector sparing**: rimappa i settori bad trasparentemente in settori di scorta sulla stessa superficie
- **Slip sparing**: rimappa tutti i settori (quando c'è un settore bad) per preservare il comportamento sequenziale
- **Track skewing**: il numero dei settori è legato da una traccia alla successiva, per consentire il movimento della testina per operazioni sequenziali

Settori e blocchi

A **basso livello il controller accede i settori individualmente**: tipicamente la dimensione di un settore è 256/512 byte ed è **identificato da una tripla <#cilindro, #faccia, #settore>**.
Il driver del disco **raggruppa una collezione di settori contigui in un blocco**. Tipicamente un blocco è di 2/4/8Kb ed è **identificato da un puntatore su uno spazio di indirizzamento contiguo**.

Dato un settore `b` e una tripla `<c, f, s>`: **`b = c(#facce * #settori) + f(#settori) + s`**
`#facce` è il numero di facce su un disco
`#settori` è il numero di settori per traccia
Di conseguenza

- `c = b / (#facce * #settori)`
- `f = (b % (#facce * #settori)) / #settori`
- `s = (b % (#facce * #settori)) % #settori`

Performance del disco

`Latenza = tempo seek + tempo rotazione + tempo trasferimento`
`Tempo seek`: tempo per muovere il braccio del disco sopra la traccia (1-20ms)
`Tempo rotazione`: tempo da attendere perché il disco ruoti sotto la testina (4-15ms)
`Tempo trasferimento`: tempo per trasferire dati da/su disco (50-100Mb/s, circa 5-10 us/settore), dipendente dal tipo di collegamento (USB, SATA...)

Per completare 500 letture randomiche in ordine FIFO:

- seek: media 10.5 ms
- rotazione: media 4.15 ms
- trasferimento: 5-10 us

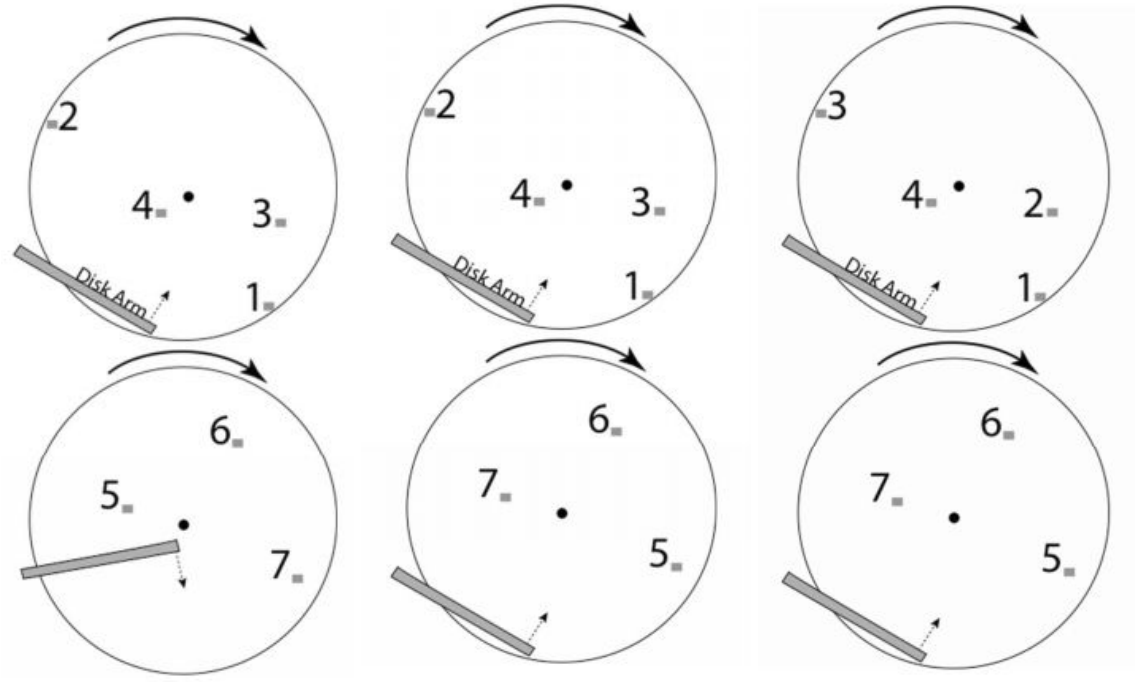
Quindi 500 * (10.5 + 4.15 + 0.01)/1000 = 7.3 secondi
Per 500 letture sequenziali, invece:

- seek: 10.5 ms (per raggiungere il primo settore)
- rotazione: 4.15 ms (per raggiungere il primo settore)
- trasferimento (su traccia esterna): 500 settori * 512 byte / 128MB/s = 2 ms

Totale 10.5 + 4.15 + 2 = 16.7 ms
Potrebbe essere necessario uno switch di traccia o di testina in più (+1ms)
Buffer della traccia potrebbe consentire di leggere qualche settore non in ordine (-2ms)

Quanto deve essere grande un trasferimento perché sfrutti l'80% del max transfer rate?
Ponendo y rotazioni, risolvo per y:
0.8 (10.5 ms + 8.4ms * y + 1 ms (y-1)) = 8.4ms * y
seek iniz. y rotaz. y-1 extra seeks
80% del tempo richiesto per leggere y tracce con overhead = tempo per leggere y tracce senza overhead
Totale circa 9 rotazioni, circa 10 Mb

Scheduling del disco
FIFO: schedula le operazioni nell'ordine in cui arrivano
Shortest seek time first: non ottimale nei tempi di risposta (es.: due cluster di richieste ai lati opposti del disco)
SCAN: muovi il braccio in una direzione finché non soddisfi tutte le richieste, poi stessa cosa ma nell'altra direzione.
CSCAN: muovi il braccio in una direzione finché non soddisfi tutte le richieste, poi riparti dalla richiesta più lontana
R-CSCAN: CSCAN ma considera anche che short track switch < rotazione



SCAN C-SCAN R-CSCAN

Tempo per 500 letture random in qualsiasi ordine?

- seek: 1ms
- rotazione: 4.15ms
- trasferimento: 5.10us

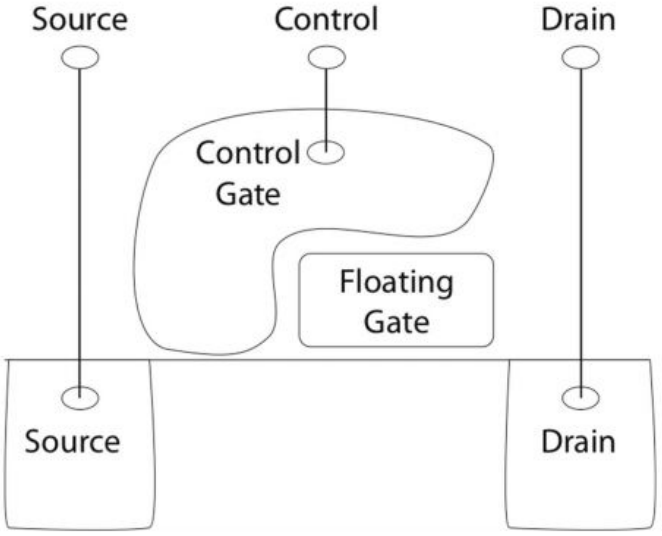
Totale = 500 * (1 + 4.15 + 0.01) = 2.2 s
Con R-CSCAN sarebbe un po' più veloce, al contrario di FIFO che sta sui 7.3 secondi
Per leggere tutti i byte di un disco?

- capacità 320Gb
- banda: 54-128Mb/s

Tempo di trasferimento: capacità/banda media = 3500s (1h)

Memoria Flash

Basate su transistor.
Per la **lettura** si guarda la tensione tra un capo e l'altro (**velocissimo**). La lettura avviene su 2-4Kb e richiede 50-100us.
Per la **scrittura** va svuotato un'insieme di transistor e poi scritti e non è banale, dovendo essere **fatta su blocchi di transistor e non sul singolo**. Il blocco di cancellazione è nell'ordine di 128-512Kb e **richiede tanti millisecondi**.
Per questo le prestazioni degradano col tempo.



Flash Translation Layer

Per evitare di dover sempre cancellare per ogni scrittura, le **cancellazioni sono fatte in anticipo**: sono **sempre disponibili pagine pulite** per le nuove scritture. Questo significa che una scrittura non può essere redirezionata in una pagina arbitraria della memoria, ma solo in una già cancellata. Cosa succede se riscrivi il blocco di un file?

- La pagina che contiene il blocco non può essere riscritta immediatamente, ma deve essere prima cancellata insieme alle pagine adiacenti
- Una pagina pulita viene usata per applicare la scrittura, ma questa pagina è da qualche parte nel disco...
  Le vecchie pagine vanno a finire "nel cestino" per essere "riciclate"

Come faccio a sapere dove sono le pagine del mio file? Il **firmware mappa il numero di pagina logica ad una locazione fisica**

- sposta le pagine vive se necessario per l'eliminazione (**garbage collection**)
- attenzione all'usura: ogni pagina fisica può essere riscritta solo un numero limitato di volte

Quindi è **in grado di scoprire precisamente dove si trova il blocco appena scritto** e in maniera del tutto trasparente all'utente.

File system su memorie flash

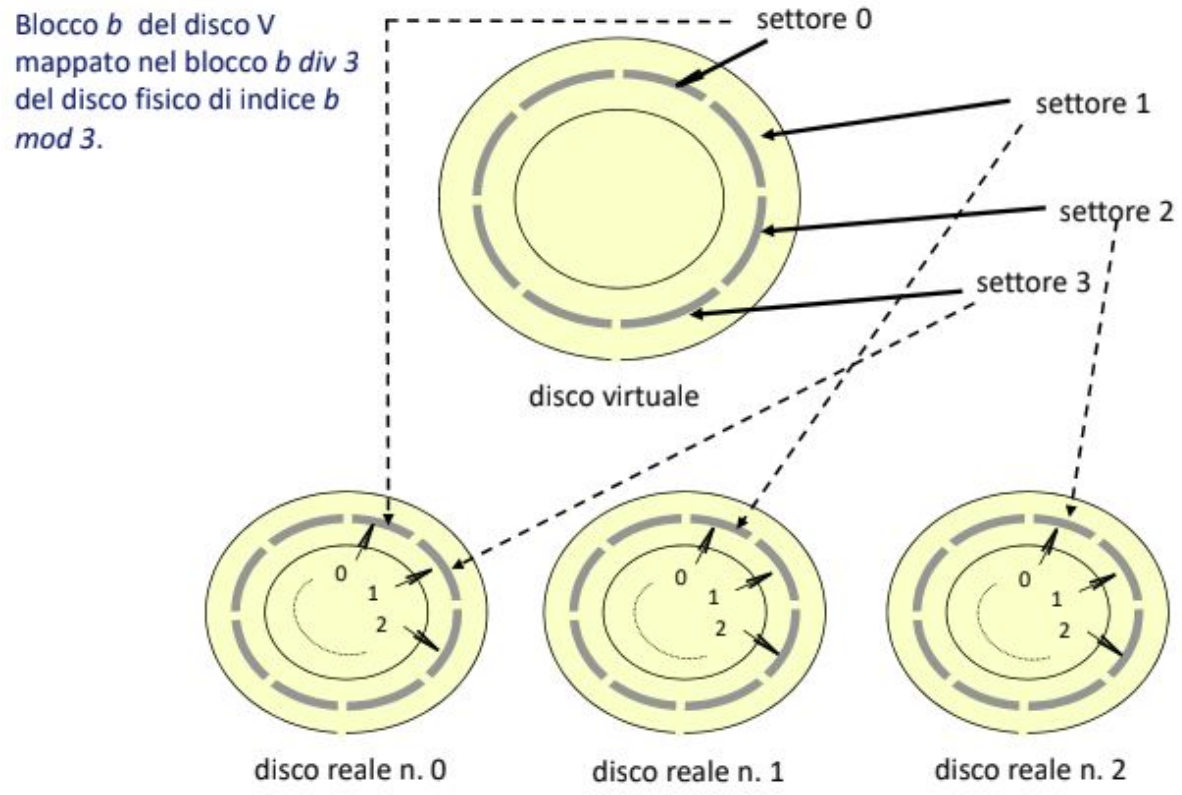
Un disco magnetico non ha bisogno di dire al file system quali blocchi sono in uso: quando un blocco è libero viene marcato tale in una bitmap e il file system lo riusa quando vuole. Quando questi file system venivano usati sulle memorie flash le performance peggioravano col tempo perchè il flash translation layer era

impegnato col garbage collection: **blocchi vivi rimappati da un'altra parte, per poter "compattare" le pagine libere e procedere con la cancellazione**.
Questo succedeva anche con tanto spazio libero. Per esempio, se il file system muoveva un file grande da un insieme di blocchi ad un altro, la memoria flash non sapeva che i vecchi blocchi potevano andare nel cestino a meno che il file system non glielo dicesse.

Comando **TRIM**: file system comunica al dispositivo quando le pagine non sono più in uso. Aiuta la Flash Translation Layer a ottimizzare il garbage collection ed è stata introdotta nel 2009/2011 nella maggior parte dei SO.

Dischi RAID Redundant Array of Independent Disks
L'architettura RAID

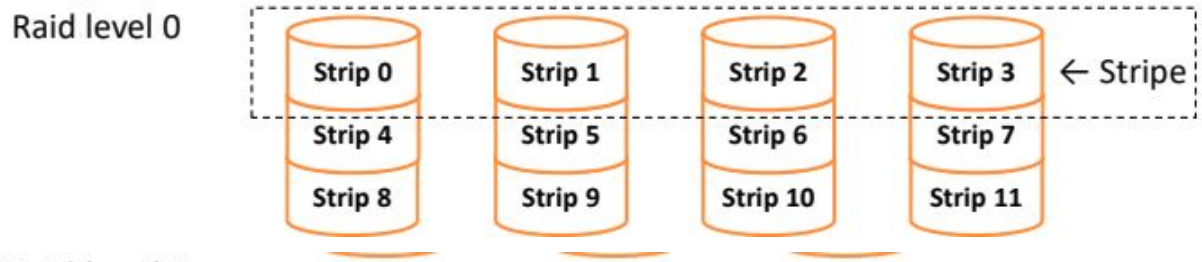
- realizza disco **virtuale** di capacità superiore a quella dei singoli dischi
  L'interfacciamento è realizzato come se fosse un unico disco
- **sfrutta il parallelismo** per ottenere un **accesso più veloce** (non linearmente), poiché i blocchi consecutivi di un file sono distribuiti sui dischi in modo da poter fare **operazioni contemporanee**
- **sfrutta la ridondanza per aumentare l'affidabilità** permettendo di correggere certi tipi di errore
- In compenso ha più probabilità che il disco (virtuale) si scassi, avendo più dischi fisici

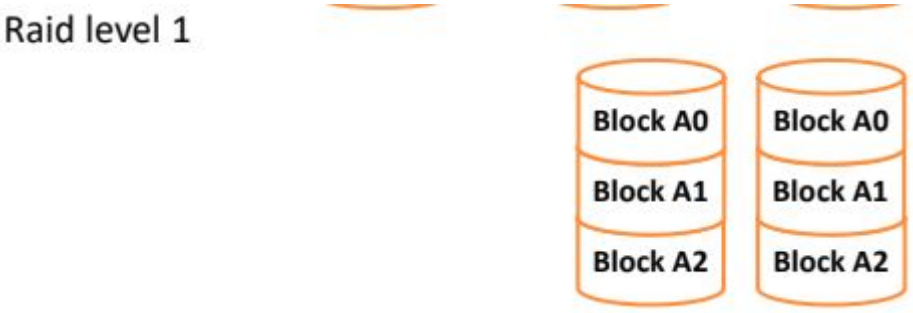


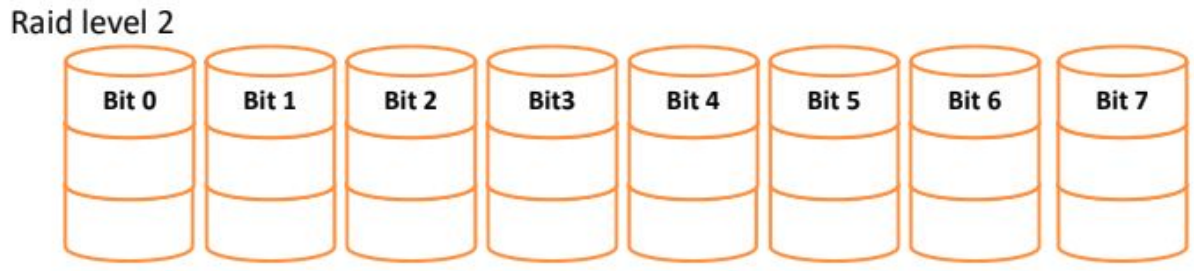
Livelli di RAID

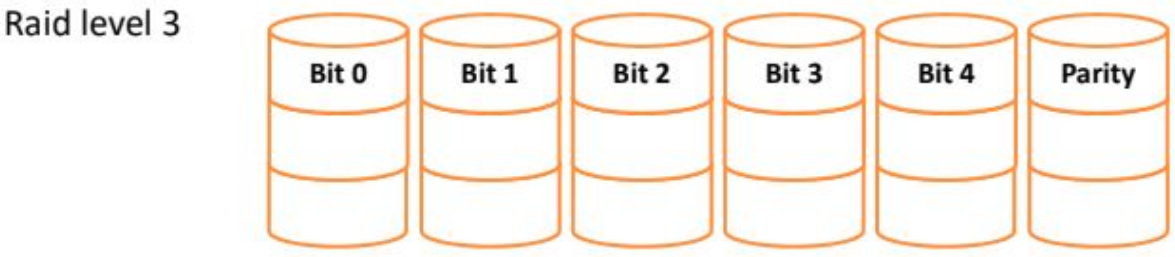
- Livello 0: dischi asincroni, nessuna ridondanza
  Si possono effettuare **contemporaneamente operazioni indipendenti**.
  Anche detto **JBOD** (Just a Bunch Of Disks)
- Livello 1: dischi asincroni, disco con copie ridondanti (mirror)
  Si possono effettuare **contemporaneamente operazioni indipendenti** e **correggere errori**
- Livello 2: dischi sincroni, dischi ridondanti contengono codici per la correzione degli errori
  Non si possono effettuare contemporaneamente operazioni indipendenti e correggere errori

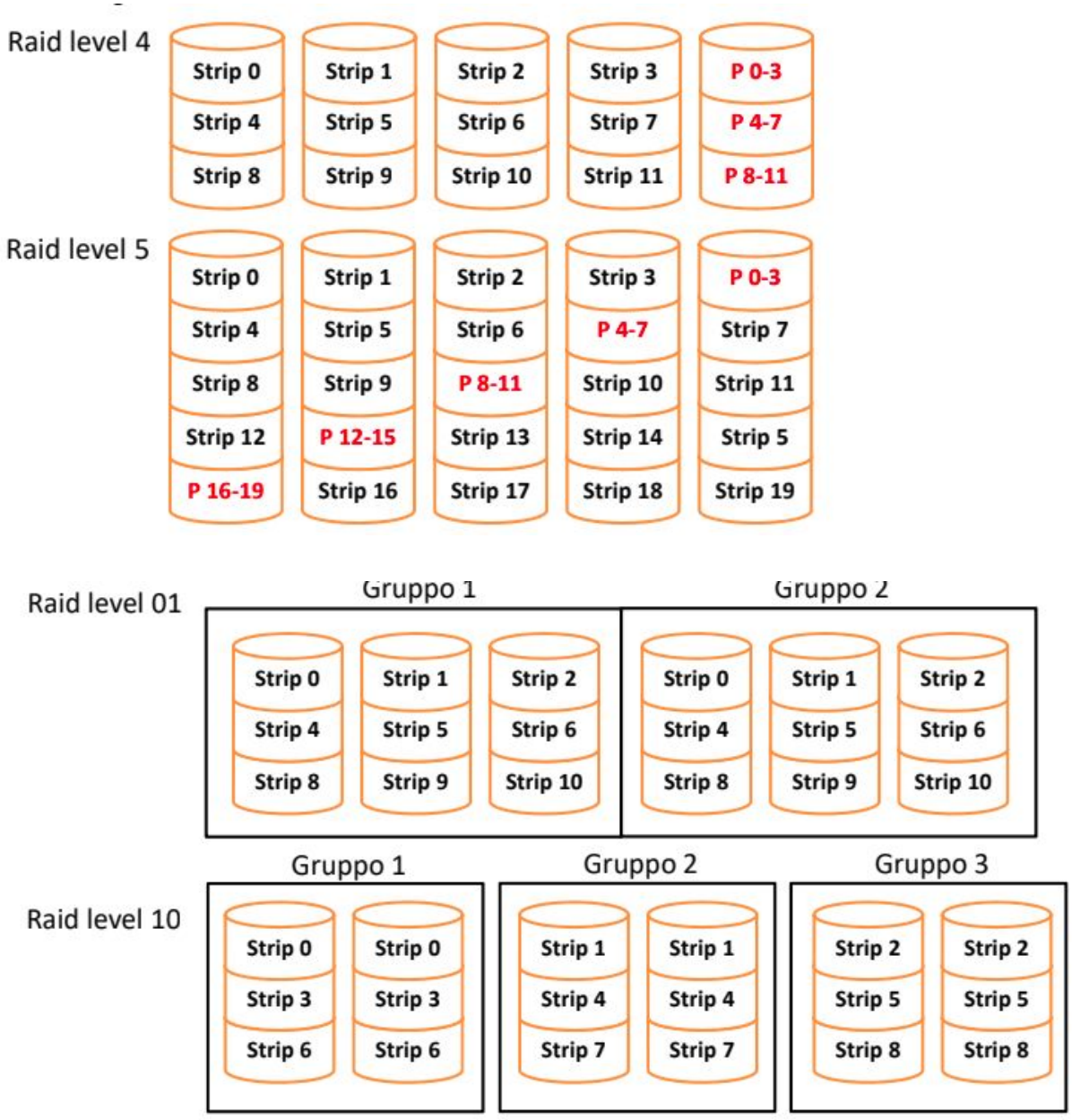
- Livello 3: dischi sincroni, un solo disco ridondante
  Il **disco ridondante contiene la parità** del contenuto degli altri dischi
  Non si possono effettuare contemporaneamente operazioni indipendenti e correggere errori
- Livello 4: dischi asincroni, un disco ridondante
  Il **disco ridondante contiene la parità** del contenuto degli altri dischi
  Si possono effettuare **contemporaneamente operazioni indipendenti** e **correggere errori**
  Il **disco ridondante è sovraccarico nei piccoli aggiornamenti**
- Livello 5: come il 4, ma la parità è distribuita fra tutti i dischi
  Permette un bilanciamento del carico migliore tra i dischi
- Livello 1+0: dischi asincroni, mirror di stripes
  Livello 0+1: dischi asincroni, stripe di mirror











Slide 67-68 di 11-12-storage.pdf per esempio di RAID4

I File Systems

In un file system **la stragrande maggioranza dei file sono piccoli**, ma **il maggior spazio è occupato dai file grandi**.

La **maggior parte degli accessi sono a file piccoli**, la **maggior parte di byte spostati sono di file grandi**.

Ci interessa sapere se un file è piccolo o grande perché gli **algoritmi non è detto funzionino in modo uguale su entrambi i tipi**. **I file grandi determinano il throughput**, ma la **maggior parte degli accessi sono su file piccoli**, quindi **se mi concentro sui grandi rischio di penalizzare i piccoli**.

Le problematiche nei file system sono differenti e le **prestazioni sono frutto del compromesso**.

La **maggior parte dei file sono scritti/letti sequenzialmente**, **alcuni in modo randomico** (es. database, swap).

Alcuni file hanno una dimensione predefinita alla creazione, altri partono piccoli e crescono col tempo (es. stdout, log di sistema...)

Design dei File System

Per i file piccoli:

- **blocchi piccoli** per massimizzare prestazioni storage
- **operazioni concorrenti più efficienti** che delle sequenziali
- **file usati insieme memorizzati insieme**

Se so che devo memorizzare un file piccolo conviene usare blocchi piccoli, per ridurre la frammentazione interna.

Per i file grossi:

- **blocchi grossi**, quindi meno accessi al disco
- **allocazione contigua** per un accesso sequenziale
- **lookup efficiente** per accesso randomico

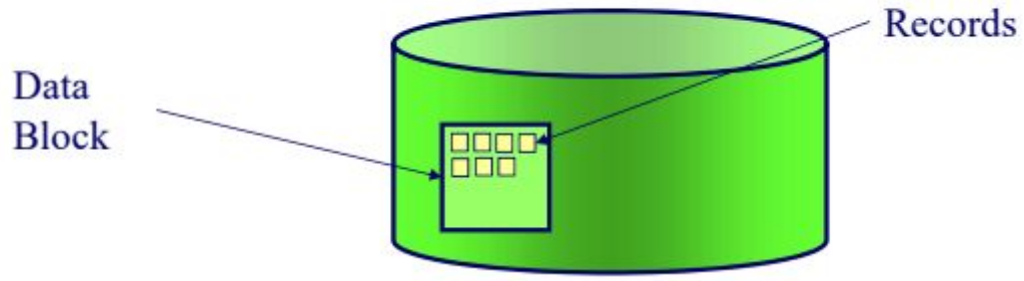
Alla creazione del file, posso non sapere:

- se il file sarà grande o piccolo
- se il file sarà persistente o temporaneo
- se il file sarà usato sequenzialmente o randomicamente

Tre strutture dati principali:

- **Directory**: nome file -> metadati
- **File metadata**: per scoprire i blocchi sui quali il file è memorizzato ed altre proprietà
- **Freemap**: lista dei blocchi liberi del disco

Data Blocks e Record



I **dati nei file sono accessibili in record**. Ad es. su Unix il singolo record è il byte.

I **dati sono fisicamente memorizzati** (e accessi) **a blocchi**, su Win i blocchi sono solitamente chiamati clusters.

Solitamente, **block size >> record size**, un blocco spesso è composto da più settori.

Si pongono una serie di domande riguardanti le scelte di design:

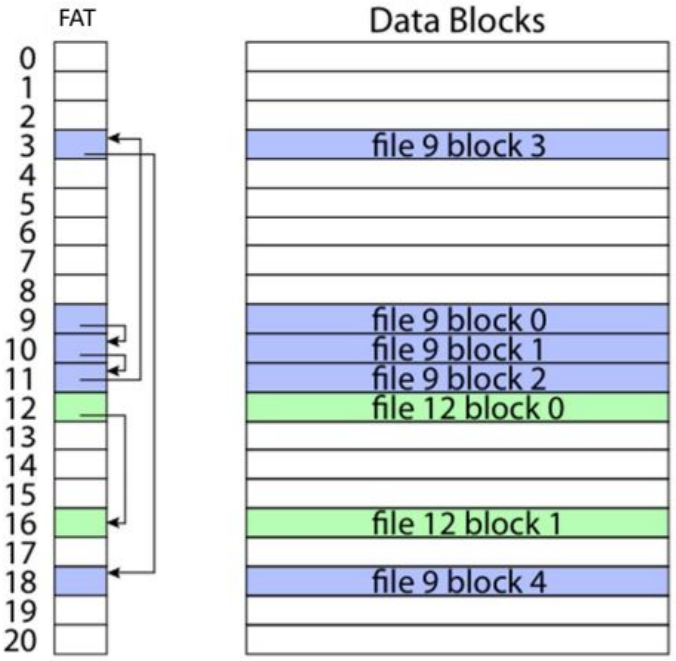
- **Struttura degli indici**: come vengono localizzati i blocchi di un file?
- **Granularità degli indici**: che dimensione del blocco usare?
- **Spazio libero**: come si trovano i blocchi inutilizzati del disco?
- **Località**: come preservare la località spaziale?
- **Affidabilità**: che succede se il computer crasha mentre sta avvenendo un'operazione sul file system?

	FAT	FFS	NTFS
Index structure	Linked list	Tree (fixed, asym.)	Tree (dynamic)
granularity	block	block	extent
free space allocation	FAT array	Bitmap (fixed location)	Bitmap (file)
Locality	defragmentation	Block groups + reserve space	Extents Best fit defrag

file name offset --directory→ *file number offset* --index structure→ *storage block*

Microsoft File Allocation Table (FAT)

La struttura principale del file system FAT è una **linked list**: semplice, facile da implementare e ancora largamente usata. La **file table** è una mappa lineare di tutti i blocchi sul disco, ed **ogni file è una linked list di blocchi**.



Pro:

- **Facile trovare un blocco libero**
- **Facile appendere** ad un file
- **Facile eliminare** un file

Contro:

- **Dimensione FAT**: dovrebbe essere caricata in memoria principale e **limita la dimensione del file system**
- **Metadati limitati** e **nessuna protezione**
- **Frammentazione**
  - **Blocchi di file** con un dato nome potrebbero essere **sparsi**
  - **File nella solita directory** potrebbero essere **sparsi**
  - Il problema peggiora tanto più che il disco è pieno

Limitazioni del FAT

Dati **L lunghezza in bit degli elementi della FAT** e **B dimensione in byte dei blocchi del disco**, allora il numero massimo di blocchi indirizzabili è 2^L (**capacità del disco** o della partizione).

Di conseguenza, la **massima estensione del file system** è di 2^L blocchi, cioè di **B * 2^L byte**.

Se ogni elemento occupa N byte (solitamente L è multipla del byte), la FAT occupa complessivamente N * 2^L byte.

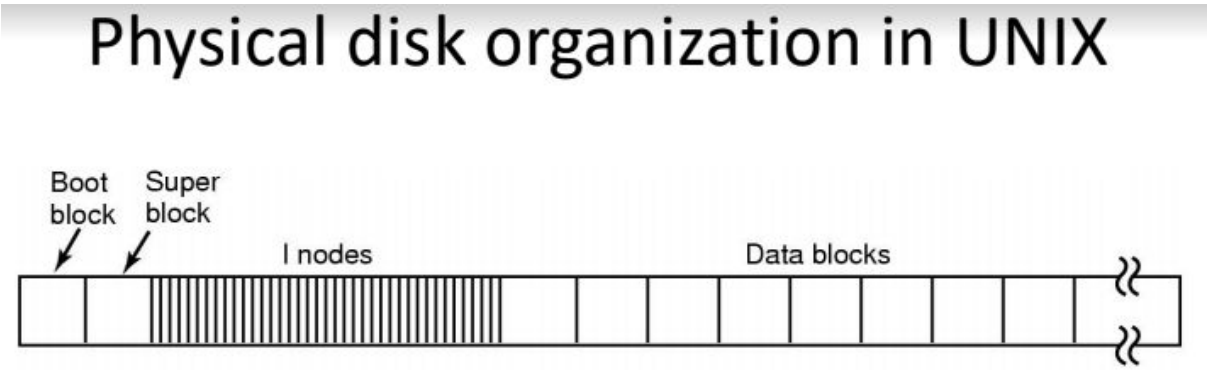
Es.: N = 2 (FAT16), B = 2^{10} (blocchi da 1Kb). La massima estensione del file system è di 2^{16} blocchi, quindi 2^{26} byte cioè 64Mb, e la fat occupa 2 * 2^{16} byte = 128Kb. Con una memoria paginata e pagine di 1Kb, la FAT occupa 128 pagine.

Dato che gli elementi che descrivono un file possono essere distribuiti su molte pagine diverse, possono verificarsi frequenti errori di pagina quando si percorre un file. quindi per realizzare file system più estesi si usano blocchi di dimensioni maggiori.

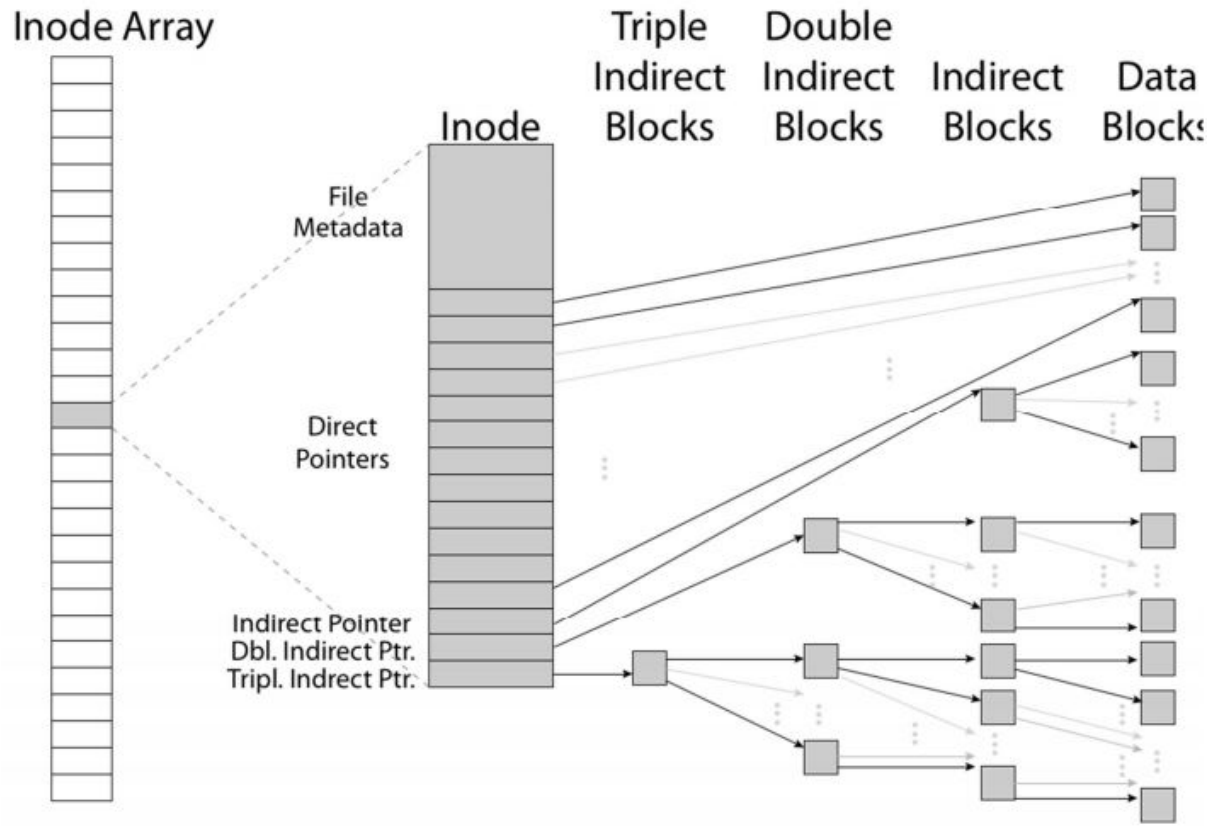
Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

Massima dimensione del File System per diverse ampiezze dei blocchi

Berkeley UNIX FFS (Fast File System)
La struttura dati principale è la **inode table**, "analoga" alla FAT table. Un **inode** contiene i **metadati** (proprietario del file, permessi di accesso, tempi di accesso...) e una **serie di puntatori ai blocchi** dei dati.



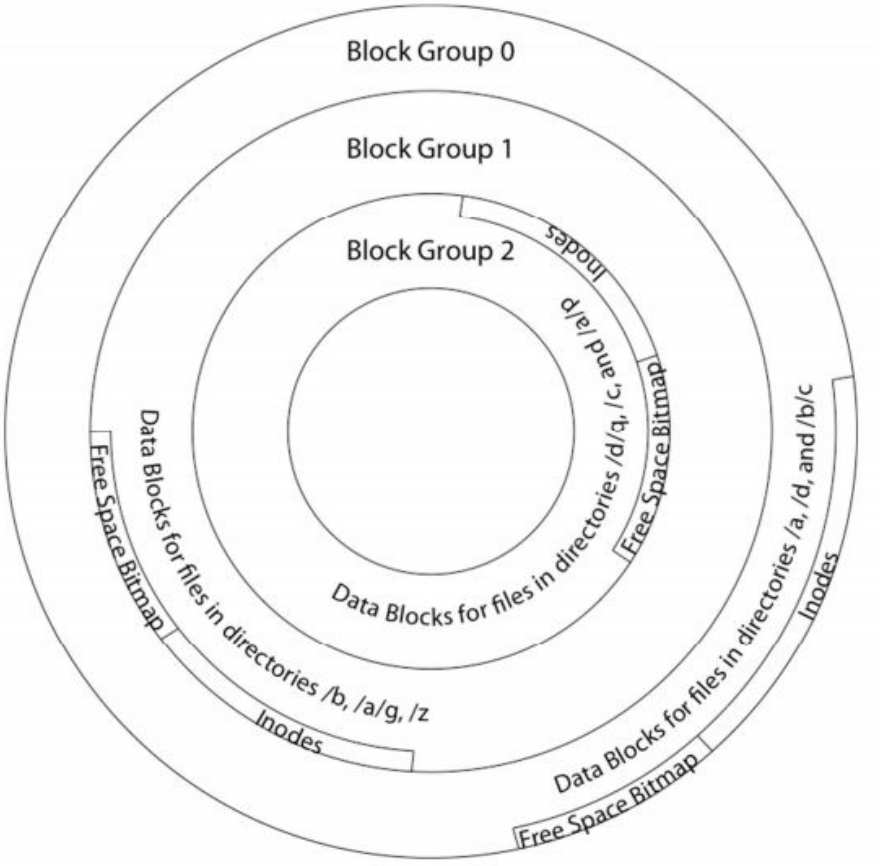
inode FFS
Metadati: proprietario del file, permessi di accesso, tempi di accesso...
Set di 12 puntatori dati: con blocchi da 4Kb si ha una dimensione massima di 48Kb
Puntatore indiretto al blocco: puntatore ad un blocco del disco di puntatori dati. Con blocchi di 4Kb e 1k puntatori a blocchi dati si ha 4Mb di spazio
Puntatore indiretto doppio al blocco: 1k blocchi indiretti, quindi 4Gb (+ 4Mb + 48Kb)
Puntatore indiretto triplo al blocco: 1k blocchi indiretti doppi, quindi 4Tb (+ 4Gb + 4Mb + 48Kb)



Slide 24-27 di 13-filesys.pdf esempio di inode

Albero Asimmetrico FFS
File piccoli -> albero shallow, storage efficiente per file piccoli
File grandi -> albero deep, lookup efficiente per accessi casuali a file grandi

Località dell'FFS
Allocazione a gruppi di blocchi: un **block group** (gruppo di blocchi) è una collezione di cilindri vicini. I file nella solita directory sono localizzati nello stesso gruppo, le sottodirectory sono in differenti block group.
L'inode table è **distribuita su tutto il disco** e i **file sono allocati** secondo politica **first fit**: i file piccoli sono frammentati, quelli grandi sono contigui.



FFS
Pro:

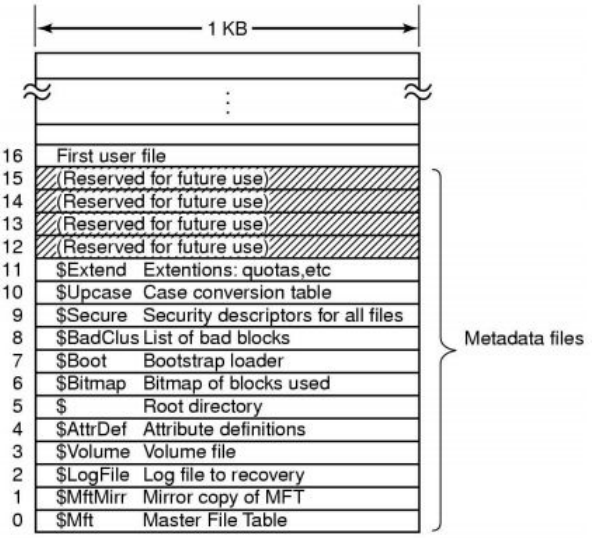
- **Memorizzazione efficiente** sia per file piccoli che per file grandi
- **Località** sia per file piccoli che grandi
- **Località per metadati e dati**

Contro:

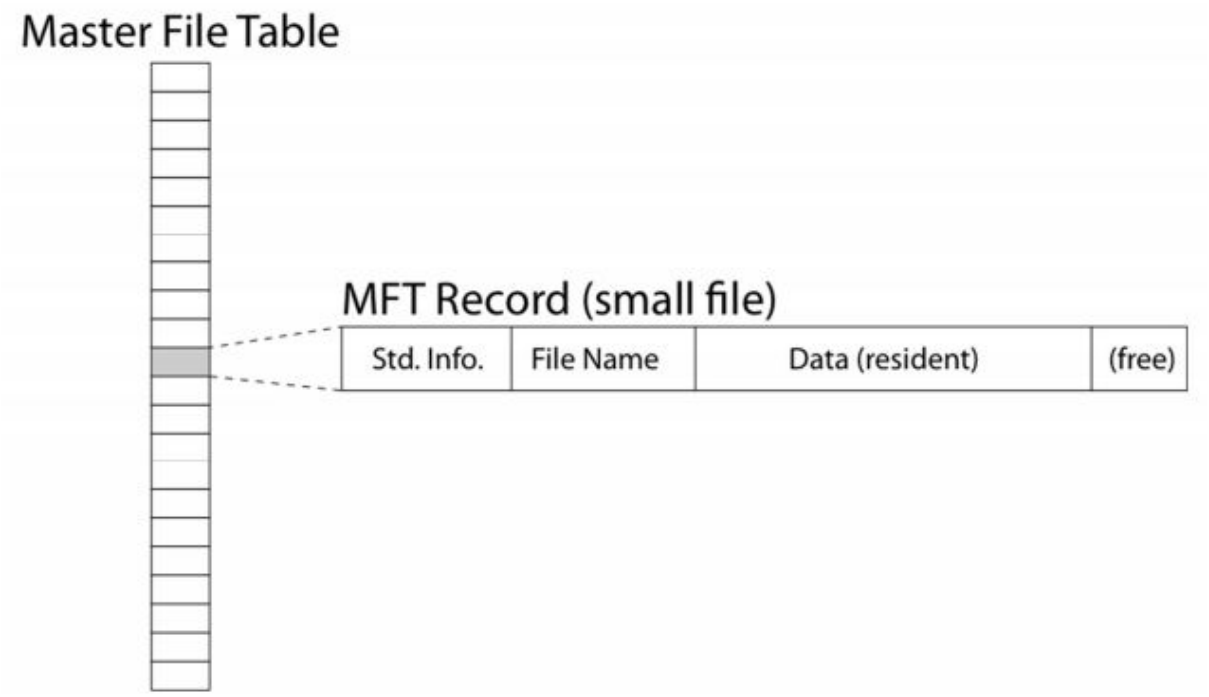
- **Inefficiente per file minuscoli** (es. un file da 1 byte richiede sia un inode che un blocco dati)
- **Codifica inefficiente quando un file è prevalentemente contiguo su disco** (non c'è l'equivalente delle superpagine)
- **Necessita di almeno il 10-20% di spazio libero** per evitare la frammentazione

NTFS

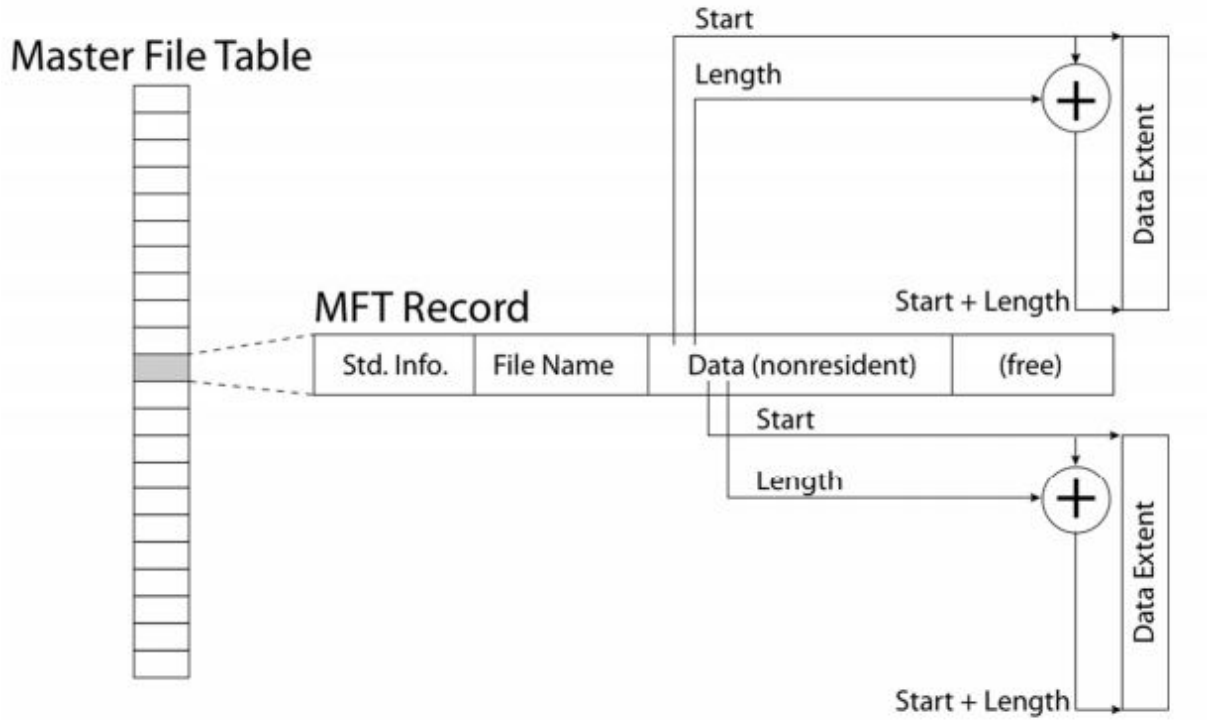
La **Master File Table** è una tabella di record (uno per file), con 1Kb di storage flessibile sia per metadati che per dati. **La MFT stessa è un file**.
Puntatori a blocchi per gestire i blocchi, ext4 adotta un approccio simile. Inoltre alla creazione di un file si può suggerire la dimensione che esso avrà.
Journaling per l'affidabilità.

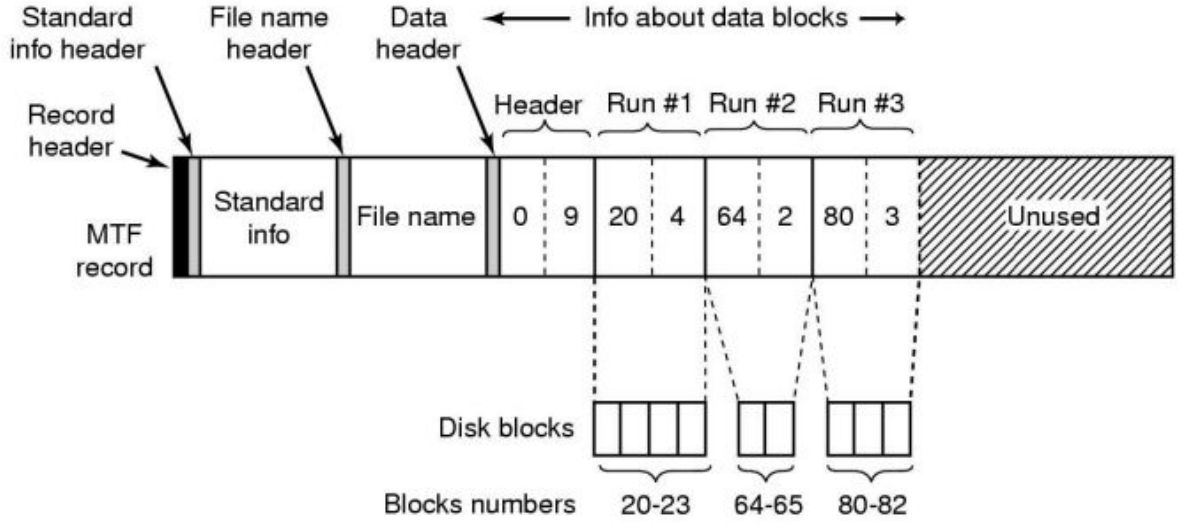


NTFS File Piccoli



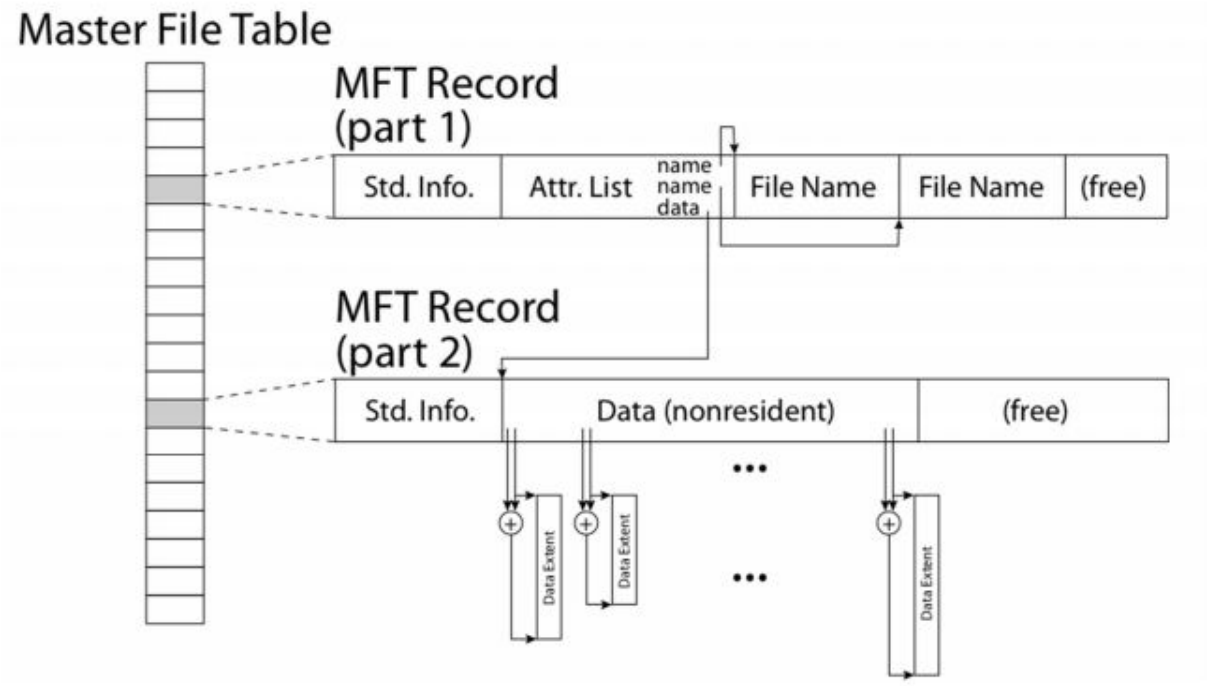
NTFS File Medi



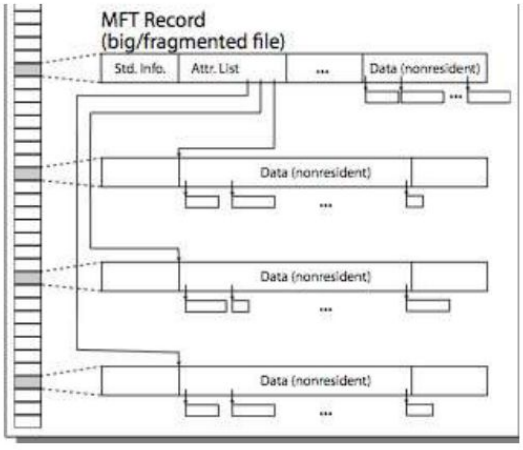


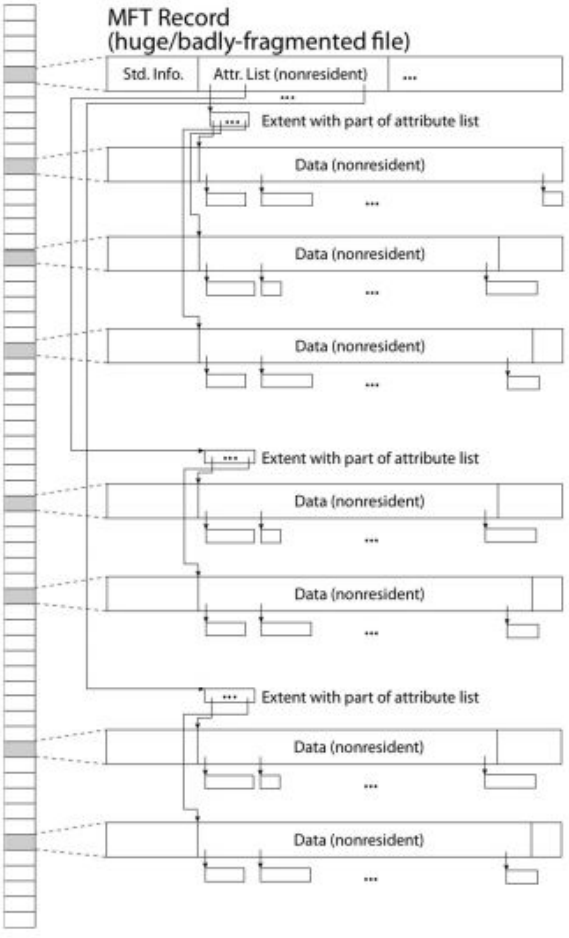
NTFS blocco indiretto

Singolo

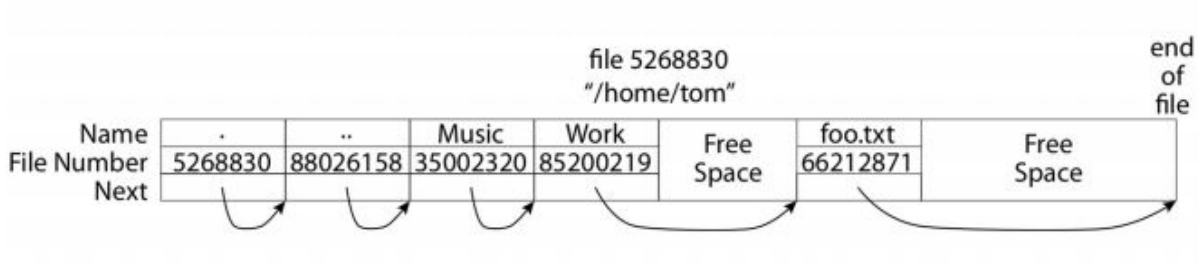


Multiplo





Directory
Le directory sono file anche in NTFS, **mappano il nome di un file al numero del file** (MFT #, inode #...)
file name offset --directory→ *file number offset* --index structure→ *storage block*



la fat non ha un descrittore di fat e neanche un nome, uguale per la ilist. Ecco perché non è un file, non ha bisogno di essere rappresentate in una directory: è posizionata dopo il master boot record in un punto fisso del disco, la ilist uguale e quanto è lunga è scritto nel mbr.

master file table della ntfs è file perché può essere aumentata e ristretta a seconda del numero di file presenti sul disco

descrittore 0 mft è descrittore della mft. primo blocco mft memorizzato nel superblocco (primo blocco dopo master boot record)

Journaling usando log, scrivo azione da fare e scrivo quando è fatta.

Segmentazione paginata
Spazio virtuale processo è segmentato, e il programmatore (a livello linguaggio macchina almeno) vede segmentato
Si mantiene tabella segmenti nel sistema
Ogni segmento è paginato a sua volta -> tabella delle pagine
quindi tabella segmenti rappresenta segmento, all'interno non trovo base segmento ma puntatore tabella pagine del segmento. Tabella pagina è del segmento, tabella segmenti è del processo
In ogni segmento: puntatore pagine, lunghezza tabella pagine (# pagine), permessi
In ogni pagina: frame pagina, diritti accesso per pagina
La condivisione può essere per segmento o per pagina

...slide come funziona…

frammentazione:
esterna dovendo allocare porzioni contigue di dim variabili capita che la memroia libera sia divisa in frazioni troppo piccole per allocare il segmento necessario, la segmentazione richiede di allocare blocchi contigui
interna perdita memoria all'interno dello spazio del processo, alloco pagine intere, se qualcuno dichiara poco spazio l'allocazione minima è comunque una pagina (es. pagina 1k per array da 5 byte -> frammentazione interna)
con segmentazione paginata ho entrambe segmentazioni?
esterna non ci può essere perché le pagine sono dimensione fissa, quindi possono stare ovunque ma i blocchi sono di dimensione fissa.

interna invece ci può essere perché segmento di un byte ha comunque almeno una pagina da tot (1kb es.) quindi spazio rimanente inutilizzato.
Non è grave perché in genere lo spazio sprecato è mezza pagina per segmento (circa 10 segm per processo)

paginazione multilivello
non ci sono più segmenti: spazio virtuale è unica zona contigua. Indirizzo virtuale è indirizzo che fa riferimento a qualsiasi byte dello spazio, però viene diviso in quattro campi: indice, indice2, indice3, pageoffset
Associato al processo tabella pagine primo livello, si estrae un descrittore che riferisce tabella pagine secondo livello, indicizzata tramite secondo campo, individua descrittore all'interno che punta tabella terzo livello, indicizzata dal terzo campo che estrae descrittore della pagina.
...es a 2 livelli...
Ha una sola tabella di primo livello perché index va nella tabella di 1 livello, altrimenti non saprei quale prendere, ha un numero variabile di tabelle di 2 livello a seconda di quante servono. Dal punto di vista del processo è un unico spazio di memoria contiguo paginato, può essere usato in maniera sparsa e frammentato, ma non è un problema perché alloco tabelle di 2 livello solo se mi servono (stesso discorso per livelli più profondi)