1 Lezione del 04-12-25

Continuiamo a trattare le tecniche di mappatura dei record fisici in record logici.

1.0.1 Allocazione a lista concatenata

In questo caso vogliamo organizzare i blocchi sui quali viene mappato ogni file secondo una struttura a **lista concatenata**.

I vantaggi di questo approccio sono:

- Non esiste frammentazione esterna, in quanto ogni record fisico libero può essere usato ed inserito come successivo a qualsiasi altro record logico;
- Il costo di allocazione è quindi minore;
- L'accesso sequenziale è a basso costo (le liste concatenate sono ottime proprio per gli accessi sequenziali).

Gli svantaggi sono invece:

- La facilità di introdurre errori nel caso di danneggiamenti di link fra nodi della lista;
- Lo spazio aggiuntivo occupato dai puntatori ai prossimi elementi di ogni elemento della lista;
- L'accesso diretto risulta oneroso: per accedere all'i-esimo byte abbiamo bisogno di:

$$t_B = \frac{i}{N_{\text{byte}}}$$

  iterazioni, cioè dobbiamo scansionare i primi t_B blocchi della list;

- Dobbiamo inoltre notare il costo della ricerca di un blocco puntato, che non è più contiguo ai blocchi precedenti, ma potrebbe trovarsi in regioni arbitrarie della memoria.

Il sistema operativo Microsoft Windows implementa l'allocazione a lista concatenata attraverso le cosiddette **FAT** (*File Allocation Table*). Queste non sono altro che tabelle che associano ad ogni blocco fisico, il blocco fisico successivo.

1.0.2 Allocazione a lista doppiamente concatenata

Possiamo facilmente estendere l'approccio precedente rendendo la lista **doppiamente concatenata**.

I vantaggi sono immediati:

- Possiamo permettere la scansione in due direzioni di ogni file;
- Il sistema è reso più robusto dalla presenza di più link (si possono recuperare errori di perdita di un link).

Gli svantaggi sono invece dati principalmente dall'aumento dello spazio necessario per blocco, in quanto raddoppierà il numero di puntatori da mantenere (da 1 a 2).

1.0.3 Allocazione a indice

Nell'**allocazione a indice**, a ogni file viene associato un blocco (detto *indice*) in cui sono contenuti tutti gli indirizzi dei blocchi su cui è allocato il file.

I vantaggi sono gli stessi dell'allocazione a lista, con l'aggiunta di:

- Possibilità di fare accesso diretto senza scansioni (sfruttando l'indice);
- Maggiore velocità di accesso rispetto alle liste.

Lo svantaggio principale di questo approccio è la sua difficile **scalabilità**, data dalla crescita delle dimensioni dell'indice.

Assunta N_{byte} come la dimensione del blocco in byte, S come la dimenzione del disco, abbiamo che ogni entrata dell'indice dovrà essere almeno:

$$S_r = \frac{S}{N_{\text{byte}}}$$

per cui il numero di blocchi che potremo indicizzare sarà:

$$N_{\text{blocchi}} = \frac{N_{\text{byte}}}{S_r}$$

e la dimensione massima del file indicizzabile:

$$S_{\text{max}} = N_{\text{blocchi}} \times N_{\text{byte}}$$

Una soluzione può essere di concatenare più indici, dedicando ad ogni indice un nuovo blocco.

Vediamo che in UNIX si usa un approccio simile all'allocazione a indice.

1.1 Filesystem UNIX

Abbiamo introdotto il fatto che in UNIX un file è rappresentato da un descrittore detto **i-node**.

Questo è un descrittore che contiene i seguenti attributi:

- Il **tipo** del file, scelto fra *file ordinario*, *directory* o *file speciale*;
- Il **proprietario** del file (utente e gruppo, user-id e group-id);
- I 12 bit di **protezione** (i 9 bit dei permessi, SUID, SGID e STIcky bit);
- Le **date** di creazione e modifica del file;
- La **dimensione** del file;
- Il numero di **link** al file;
- Il cosiddetto **vettore di indirizzamento** (costituito da 13 a 15 indirizzi di blocchi). Questo consente l'indirizzamento dei blocchi di dati sui quali è allocato il file secondo una struttura ad indice.

Approfondiamo il vettore di indirizzamento. Assunto che la dimensione di un blocco è 512 byte, e gli indirizzi si trovano su 32 byte, si ha che ogni blocco può contenere 128 indirizzi di blocco.

Adottiamo quindi un'approccio a più livelli di indirezione sulla base della dimensione del file:

- I primi 10 blocchi di dati sono accessibili direttamente (10×512 byte $= 5$ KiB);
- I prossimi 128 blocchi sono accessibili con *indirezione singola*, accedendo al puntatore 11 (128×512 byte $= 64$ KiB);
- I prossimi 128×128 blocchi sono accessibili con *indirezione doppia*, accedendo al puntatore 12 ($128 \times 128 \times 512$ byte $= 8$ MiB);
- I prossimi $128 \times 128 \times 128$ blocchi sono accessibili con *indirezione tripla*, accedendo al puntatore 13 ($128 \times 128 \times 128 \times 512$ byte $= 1$ GiB);

Questo approccio si ripete cos' fino al 15-esimo indice. In questo modo si riesce a raggiungere dimensioni massime del file dell'ordine del GB, mantenendo però le strutture a indice piccole e con poca indirezinoe per file di piccole dimensioni.

1.1.1 Organizzazione logica del file system UNIX

La filosofia del file system UNIX è che *tutto è un file*, per cui un file può rappresentare un file effettivo in memoria, una directory, o un *file speciale*, che può rappresentare un dispositivo accessibile in lettura/scrittura, come un socket, come un costrutto di sistema (come ad esempio le pipe).

Abbiamo visto come il descrittore di file è l'i-node. Possiamo anticipare che gli i-node sono allocati in una tabella centralizzata detta i-list, dove ogni i-node è identificato da un identificatore univoco detto i-number.

Una volta stabilita l'identificazione attraverso i-number, si possono rappresentare le directory semplicemente come liste di i-number associate a nomi per ogni file.

1.1.2 Organizzazione fisica del filesystem UNIX

Abbiamo quindi visto che in UNIX si adotta un tipo di allocazione ibrida (allocazione ad indici a più livelli). I blocchi fisici hanno dimensione tra i 512-4096 byte. La superficie del disco virtuale è quindi divisa in quattro partizioni:

- Il **boot block**: contiene informazioni fondamentali per il bootstrap (appendice sul disco delle informazioni o programmi che devono essere eseguiti al momento del bootstrap, inizializzazione del sistema);
- Il **super block**: contiene nformazioni sull'organizzazione del file system (in particolare i limiti delle quattro regioni che stiamo descrivendo, il puntatore alla lista dei blocchi liberi e il puntatore a una lista degli i-node liberi)
- L'**i-list**: tabella contenente tutti gli i-node di file, directory e dispositivi. Ogni file ha associato uno o più nomi simbolici, uno e un solo descrittore (già visto) detto i-node (raggiungibile a partire da un intero detto i-number, che è l'indice dell'elemento posto nell'array i-list);
- La partizione **data blocks**: contenente i blocchi utilizzati per allocare file.

1.1.3 Accesso ai file in UNIX

L'**accesso ai file** si fa con le solite primitive `read()` e `write()`, notando che si possono avere diversi tipi di accesso anche per la solita primitiva (`write()` in truncate o append, ecc...).

L'accesso è *sequenziale*, e non si ha *strutturazione* (i file sono solamente sequenze di record, cioè byte). La posizione corrente dell'accesso viene mantenuta da un puntatore detto *I/O pointer*.

Vediamo quindi come il programmatore accede al filesystem. A ogni processo è associata una tabella dei **file aperti**. Ogni elemento della tabella rappresenta un file (che ricordiamo può anche essere un dispositivo, cioè un file speciale). Il *file descriptor* (o **filde**) di tale file non è altro che l'indice del file nella tabela dei file aperti di un processo. Esistono 3 filde di default:

- **stdin** (indice 0): lo stream di ingresso di default;
- **stdout** (indice 1): lo stream di uscita di default;
- **stderr** (indice 2): lo stream di errore di default.

Essendo un'informazione utile quando il processo è caricato, la tabella dei file aperti del processo è allocata nella sua *user structure*.

Lato kernel, manteniamo 2 tabelle per l'accesso ai file:

- La tabella dei **file attivi**, che per ogni file aperto contiene una copia dell'i-node e il numero di riferimenti a tale file (per permettere la rimozione automatica quando si disattiva l'ultimo riferimento);
- La tabella dei **file aperti di sistema**, che ha un elemento per ogni operazione di apertura non ancora chiusa, contenente:
  - L'I/O pointer, cioè la posizione corrente all'interno del file;
  - Il puntatore all'i-node del file nella tabella dei *file attivi*.

La divisione in due tabelle è resa necessaria dal fatto che più processi possono aprire lo stesso file: l'informazione condivisa fra i processi (cioè l'inode) è contenuta nella prima tabella, mentre l'informazione specifica ad ogni processo (cioè l'I/O pointer) è contenuta nella seconda tabella.

1.1.4 Primitive di accesso ai file

Vediamo quindi le primitive di accesso ai file nel dettaglio:

- `int open(char nomefile[], int flag, int? mode);`
  Questa primitiva si occupa di aprire un file, cioè crearne l'entrata nella tabelle dei file aperti e restituirne il filde.
  - `nomefile` è il nome del file, relativo alla working directory corrente, o assoluto;
  - `flag` esprime il metodo di accesso. Ad esempio, si può specificare `O_RDONLY` per l'accesso in sola lettura, o `O_WRONLY` per l'accesso in sola scrittura;
  - `mode` è un parametro richiesto soltanto se l'operazione di apertura richiede la creazione di un file (con `O_CREAT`). In tal caso, specifica i bit di protezione.

  Il valore restituito è il fild del file, oppure -1 in caso di errore;

- `int close(int fd);`
  Questa primitiva è la duale alla `open()`, e si occupa di chiudere il file aperto indicizzato da un certo filde.

  – `fd` è, appunto, il fide del file da chiudere.

  Restituisce 0 se l'operazione va a buon fine, e -1 in caso di errori;

- `int read(int fd, char* buf, int n);`
  Si occupa di leggere `n` byte nel buffer `byf` da un certo filde.

  – `fd` è il filde del file da cui leggere;
  – `buf` è l'area in cui trasferire i byte letti;
  – `n` è il numero di byte da leggere.

  In caso di successo, restituisce un intero positivo $\leq n$ che rappresenta il numero di byte letti. Altrimenti restituisce -1;

- `int write(int fd, char* buf, int n);`
  Si occupa di scrivere `n` byte dal buffer `byf` su un certo filde.

  – `fd` è il filde del file da cui leggere;
  – `buf` è l'area da cui ottenere i byte da scrivere;
  – `n` è il numero di byte da scrivere.

  In caso di successo, restituisce un intero positivo $\leq n$ che rappresenta il numero di byte scritti. Altrimenti restituisce -1.

1.2 Processi in UNIX

Restando sull'argomento UNIX, vediamo nel dettaglio come questo sistema operativo gestisce i **processi**.

1.2.1 Stato dei processi UNIX

UNIX supporta lo *swapping* di processo, per cui il diagramma di stato del processo è simile a quello riportato in 17.3.2:
diagramma di stato 17.3.2 + zombie
L'aggiunta a questo diagramma è quella dello stato **zombie**, in cui un processo va a trovarsi prima di essere effettivamente terminato. Un processo passa allo stato zombie quando viene effettivamente terminato, prima che il suo processo padre ne raccolga lo stato (il cosiddetto *reaping*).

1.2.2 Descrittori di processo UNIX

Avevamo definito in 5.0.2 come un processo è rappresentato da un descrittore detto **PCB** (*Process Control Block*). Questo descrittore è sostanzialmente quello adottato da UNIX.

In verità, tale struttura si divide in più sottostrutture sulla base delle aree di competenza dei dati da rappresentare. In particolare, vogliamo distinguere su 2 caratteristiche ortogonali:

- Dove i dati devono essere **accessibili**, cioè parte *kernel* e parte *utente*;
- Se i dati possono essere soggetti a **swapping**, cioè parte *swappable* e parte *non swappable* o *residente*.

Vediamo quindi come dividiamo il PCB:

- **Process structure**: contiene le informazioni necessarie al sistema per la gestione del processo (a prescindere dallo stato del processo).

  In particolare, contiene:

  - Il **PID** (*Process IDentifier*);
  - Lo **stato** del processo;
  - I puntatori alle aree **dati** e **stack**;
  - Il riferimento alla **text structure**, su cui abbiamo più informazioni sotto;
  - Le informazioni sullo **scheduling** da operare sul processo;
  - Un riferimento al processo **padre** (cioè il suo PID),
  - Informazinoi relative alla gestione dei **segnali** UNIX (segnali inviati ma non ancora gestiti, maschere di segnale, ecc...);
  - Puntatori a processi successivi nelle **code** di scheduling;
  - Un puntatore alla **user structure** (la prossima che vediamo).

- **User structure**: contiene le informazioni necessarie solo se il processo è *residente* in memoria centrale (non si è fatto swap).

  In particolare, contiene:

  - La copia dei **registri** CPU;
  - Come abbiamo già visto, le informazioni sulle risorse allocate, come ad esempio la tabella dei **file aperti**;
  - ALtre informazioni riguardanti la gestione dei **segnali** (handler segnali, ecc...);
  - L'**ambiente** del processo, e quindi il direttorio corrente, gli argomenti, l'utente e il gruppo che lo hanno lanciato, le variabili di sistema (fra cui il PATH), ecc....

- **Text structure**: contiene informazioni riguardo a dove il codice di un processo si trova. Questa tabella viene implementata in quanto UNIX sfrutta il cosiddetto codice *rientrante*: più processi possono riferirsi allo stesso codice, e in tal caso il codice viene caricato una sola volta e riferito attraverso questa tabella.

Oltre a queste strutture, chiaramente, vorremo mantenere le aree *dati* e *stack* del processo, nonché ovviamente l'area *text* dove il codice puntato dalla text structure verrà effettivamente contenuto.

Vediamo quindi un diagramma che mostra come tutte le strutture riguardanti un processo appena visto sono disposite, sulla base delle 2 caratteristiche ortogonali viste prima:
schema strutture processo

1.3 Protezione e sicurezza

Veniamo quindi alla dicussione della **protezione** e della **sicurezza** nei S/O.

- La *protezione* riguarda l'insieme di attività che si occupano di garantire all'interno di un sistema il controllo dell'accesso alle risorse logiche e fisiche;
- La **sicurezza** riguarda invece la prevenzione di comportamenti dannosi da parte di programmi o utenti all'interno del S/O.

1.3.1 Protezione

Il controllo degli accessi, prerogativa della **protezione**, è suddivisibile in 3 livelli concettuali:

- **Modelli**: rappresentano le astrazioni che il nostro sistema realizza.

  In particolare, un *modello di protezione* definisce *soggetti*, e gli *oggetti* ai quali i soggetti hanno accesso e diritto di accesso, ovvero su cui possono svolgere operazioni. In questo, gli *oggetti* sono la parte passiva del sistema (risorse fisiche e logiche), mentre i *soggetti* sono la parte attiva (processi che agiscono per conto di utenti per accedere ad oggetti). Notiamo che un soggetto può avere diritti di accesso sia per gli oggetti che per altri soggetti (un soggetto può controllarne un altro).

  Come abbiamo introdotto in 5.0.3, un soggetto in UNIX è rappresentato dalla tripla:

  $$S =< \text{PID}, \text{UID}, \text{GID} >$$

  composta da **PID** (*Process IDentifier*, già visto), e **UID** e **GID** (*User IDentifier* e *Group IDentifier*), che rappresentano il proprietario del processo.

  Un soggetto può ulteriormente essere considerato come una coppia $(\textit{processo}, \textit{dominio})$, dove il dominio è l'*ambiente* di protezione nel quale il processo sta eseguendo (l'insieme dei suoi diritti di accesso per ongi oggetto). Si possono avere domini *disgunti* o domini con diritti di accesso *comuni*.

  Un dominio di protezione è unico per un soggetto, mentre un processo può cambiare dominio durante la sua esecuzione. In altre parole, il soggetto S_i può rappresentare il processo P in un certo dominio, e S_j può rappresentare lo stesso processo in un altro dominio. L'associazione fra processo è dominio può essere:

  - *Statica*: cioè l'insieme delle risorse disponibili ad un processo rimane fisso durante il suo tempo di vita;
  - *Dinamica*: l'associazione fra processo e dominio varia durante l'esecuzione del processo. In questo caso chiaramente dobbiamo prevedere meccanismi per consentire il passaggio da un dominio all'altro.

- **Politiche**: sono gli insiemi di regole attraverso le quali i soggetti possono accedere agli oggetti.

  Possono essere di più tipi:

  - **DAC** (*Discretionary Access Control*): il creatore di un oggetto controlla i diritti di accesso per quell'oggetto. Ad esempio, è il tipo di politiche usate in UNIX;

  - **MAC** (*Mandatory Access Control*): i diritti di accesso vengono gestiti centralmente. Viene usato in sistemi di alta sicurezza (enti governativi, difesa, ospedali, ecc...);
  - **RBAC** (*Role-Based Access Control*): ad un ruolo sono assegnati specifici diritti di accesso sulle risorse. Gli utenti possono appartenere a diversi ruoli.

  Qualunque sia tipo di politica, si nota un concetto base, cioè quello del *privilegio minimo*: ad ogni soggetto dovrebbero essere garantiti solo i diritti di accesso strettamente necessari alla sua esecuzione;

- **Meccanismi**: sono gli strumenti messi a disposizione dal sistema di protezione per imporre una determinata politica.

  La separazione fra politiche e meccanismi è la seguente:

  - La politica definisce *cosa* va fatto;
  - Il meccanismo definisce *come* viene fatto.

  Ci interessa assicurare la *flessibilità* del sistema di protezione: i meccanismi di protezione devono essere sufficientemente generali per consentire l'applicazinoe di diverse politiche di protezione.

1.3.2 Implementazione in UNIX

Abbiamo già detto che in UNIX il *soggetto* è rappresentato dalla solita tripla $S =< \text{PID}, \text{UID}, \text{GID} >$. Di questo UID e GID definiscono il *dominio* di protezione.

L'associazione soggetto/dominio è dinamica, in quanto è previsto il cambio da *user mode* a *system mode*. Inoltre, il programmatore ha a disposizione le primitive di tipo `exec()` su processi con bit SUID o SGID attivati (che quindi vengono elevati ai diritti di accesso del loro proprietario o gruppo proprietario).