Università degli studi di Salerno

Dipartimento dell'Ingegneria dell'Informazione ed Elettronica e Matematica applicata

Corso di Laurea in *Ingegneria Informatica* / L-8

Sistemi Operativi

Appunti delle lezioni di *Gennaro Francesco Landi*

I Sessione di Laurea *A.A. 2023-2024*

mi viene da piangermi
-elli

Indice

Capitolo 1

I Sistemi Operativi

1.1 Ruolo del S.O. in un Sistema di Elaborazione

Un insieme di elaborazione si può suddividere in quattro componenti:

- *hardware*: fornisce al sistema le risorse fondamentali, è composto dall'unità centrale d'elaborazione (CPU), dalla memoria e dai dispositivi di I/O
- *sistema operativo*: controlla l'hardware e ne coordina l'utilizzo da parte dei programmi applicativi per gli utenti
- *programmi applicativi*: definiscono il modo in cui si usano le risorse fornite dall'hardware per l'automazione di algoritmi
- *utente*: utilizza l'ambiente fornito dal so

Un **Sistema Operativo** è:

- un insieme di programmi che agisce da intermediario tra l'utente e l'hardware
- virtualizzatore del processore *(macchina astratta)*

Il ***ruolo*** *del S.O.* è fondamentale dal punto di vista dell'utente e del calcolatore, in quanto:

- **Punto di vista dell'utente**: funge da interfaccia essenziale tra di lui e la macchina, consentendo di interagire con il computer in modo intuitivo e senza dover conoscere i dettagli tecnici interni, come i driver o l'organizzazione della memoria. Inoltre, semplifica le attività quotidiane, come l'apertura di file, e gestisce situazioni di errore, cercando di recuperarle quando possibile e prevenirle, ad esempio evitando stack overflow o crash delle applicazioni.
- **Punto di vista del calcolatore**: il sistema operativo agisce come un supervisore e gestore delle risorse del sistema di elaborazione. Ha una visione completa e dettagliata delle risorse disponibili e offre agli utenti una visione virtuale di esse. Ottimizza e facilita l'uso di queste risorse, garantendo che vengano allocate in modo efficiente per soddisfare le richieste degli utenti e dei processi in esecuzione. In sostanza, il sistema operativo assicura che il computer funzioni in modo fluido e efficiente, gestendo risorse e interazioni con gli utenti.

Definizione di Sistema Operativo

Un sistema operativo svolge diverse funzioni cruciali per il funzionamento di un computer. Gestisce l'unità centrale, inclusi processore e memoria, il flusso di input/output, i programmi applicativi e i file attraverso il file system. Inoltre, si occupa dell'interfaccia con l'utente, fungendo da interprete dei comandi tramite una shell, e gestisce la sicurezza del sistema controllando l'accesso alle risorse.

La sua definizione si basa sulla necessità di fornire un sistema di elaborazione utilizzabile per eseguire programmi e risolvere i problemi degli utenti. Il sistema operativo è il programma centrale sempre in esecuzione nel computer, noto come kernel, con altri due tipi di programmi: i programmi di sistema *(es. drive per i dispositivi I/O)*, correlati al sistema operativo ma non parte del kernel, e i programmi applicativi, che non influenzano il funzionamento del sistema.

I moderni sistemi operativi non si limitano al kernel, ma includono anche ***middleware***, che fornisce servizi aggiuntivi per lo sviluppo di applicazioni.

1.1.1 Organizzazione di un Sistema di Elaborazione

Un moderno calcolatore *general-purpose* è composto da:

- una o più **CPU**;
- da un certo numero di **controllori di dispositivi** connessi attraverso un canale di comunicazione comune *(bus)* che permette l'accesso alla memoria condivisa dal sistema.
  Ciasqun *controllore di dispositivo* dispone di una propria memoria interna *(buffer)* e di un insieme di *registri specializzati* ed è responsabile del trasferimento dei dati tra i dispositivi periferici ad esso connessi e della propria memoria interna *(buffer)*.
  Ogni *controllore di dispositivo* ha un **driver del dispositivo** che gestisce le specificità del controllore e funge da interfaccia uniforme con il resto del sistema

1.1.2 Le Interruzioni

Nel contesto dell'organizzazione di un sistema elaborativo, sia la CPU che i controllori possono eseguire operazioni in parallelo, competendo per l'accesso ai cicli di memoria. Le **interruzioni** svolgono un ruolo fondamentale in questo processo.
L'hardware della cpu dispone di un filo chiamato linea di richiesta di interruzione *(interrupt-request line)* che controlla dopo l'esecuzione di ogni istruzione. Quando arriva una richiesta di interruzione, la CPU salva il suo stato attuale, inclusi il Program Counter (PC) e altri registri principali, in un'area di memoria dedicata. Successivamente, legge il numero di interruzione e carica l'indirizzo di una routine di servizio delle interruzioni (ISR) ed inizia ad eseguire le istruzioni di questa ISR. Una volta completata l'ISR, la CPU ripristina il suo stato precedente e continua l'esecuzione del programma interrotto.
Le *cause di* ***interruzione*** possono essere:

- *Segnali che la CPU riceve da dispositivi esterni tramite il* ***bus di sistema***, esempio un dispositivo I/O segnala che il dato richiesto è pronto;
- *Situazioni anomale/erronee nell'esecuzione di un programma* ***(trap)***, per esempio la divisione per 0;
- *Esecuzione di un'istruzione di* ***Software Interrupt*** *da parte del programma*, ovvero quando viene chiamata una ***system call***.

1.1.3 Funzionamento di un SO

Attività del Sistema Operativo:

1. Programma di avviamento (**bootstrap program**)
2. Il **kernel** inizia ad offire servizi al sistema e agli utenti
3. Gestione delle interruzioni ed eccezzioni (**traps** o **exceptions**)
4. Gestione di chiamate di sistema (**system call**)
5. Multiprogrammazione che consente il multitasking mediante la gestione del *file system* e della *memoria virtuale*

La CPU ha una **duplice modalità di funzionamento *(dual mode)***: **modalità utente** e **modalità kernel**. Per indicare quale sia la modalità attiva, l'architettura della CPU deve essere dotata di un **bit di modalità: *kernel (0)* o *user (1)***.
La **dual mode** consente la protezione del sistema operativo e degli altri utenti dagli errori del programma di un utente, solo in *modalità di sistema* la CPU può eseguire istruzioni di I/O e può accedere a tutte le risorse del sistema, invece in *modalità utente* la CPU può eseguire solo un sottinsieme delle istruzioni e può usare un sottoinsieme delle istruzioni.

1.1.4 Architettura degli elaboratori

Componenti di un sistema di elaborazione:

- **CPU**: componente hardware che esegue le istruzioni
- **Processore**: chip che contiene una o più CPU
- **Unità di calcolo *(core)***: unità di elaborazione di base della CPU
- **Multicore**: include più unità di calcolo sulla stessa CPU
- **Multiprocessore**: include più processori

I sistemi **monoprocessore** contengono una sola CPU con un unico nucleo di elaborazione *(core)*, invece i sistemi **multiprocessore** permettono una *multielaborazione simmetrica*, in cui ogni processore è abilitato all'esecuzione di tutte le operazioni di sistema.
Aggiungere nuove CPU a un multiprecessore ne aumenta la potenza di calcolo, ma ne peggiora le prestazioni, in quanto aumenta il *troughput* ma è necessario un lavoro supplementare *(overhead)* per garantire che tutti i compinenti funzionino correttamente.
Una soluzione per dimunuire l'overhead è quella di fornire a ciascuna CPU la propria memoria locale accessibile per mezzo di un bus locale piccolo e veloce, questo sitema è chiamato **NUMA** *(Non-Uniform Memory Access)* e consiste nel ditrubuire il carico di lavoro in modo più equo tra i processori riducento l'effetto collo di bottiglia che si verifica nei sistemi **SMP** *(Symmetric Multi-Processing)* in cui tutti i processori condividono la stessa memoria centrale.

1.1.5 Gestione delle Risorse

Il sistema operativo svolge numerose funzioni essenziali per gestire efficientemente le risorse di un sistema informatico. Queste funzioni includono la gestione dei processi, della memoria, dei file, della memoria di massa, della cache e dell'I/O.

Inoltre, la **sicurezza** e la **protezione** sono aspetti fondamentali del sistema operativo, con meccanismi di protezione che controllano l'accesso alle risorse del sistema e *identificatori utente* (**user ID**) che consentono di identificare univocamente gli utenti e le loro autorizzazioni di accesso. La sicurezza è importante per difendere il sistema da possibili minacce sia interne che esterne.

Infine, la **virtualizzazione** è una tecnica che consente di creare ambienti di esecuzione isolati e indipendenti all'interno di un singolo hardware fisico. Questo permette ai sistemi operativi di funzionare come applicazioni all'interno di altri sistemi operativi, consentendo un'efficienza e una flessibilità maggiori nell'utilizzo delle risorse hardware disponibili.

La **macchina virtuale (*VM*)** e il gestore della macchia virtuale sono strumenti utilizzati per gestire e monitorare gli ambienti virtuali all'interno del sistema operativo. In sintesi, tutte queste funzioni lavorano insieme per garantire un'efficiente e sicura gestione delle risorse del sistema informatico.

Capitolo 2

I Processi

Un processo è un programma in esecuzione, la sua struttura è generalmente divisa in:

- **Sezione di testo**: contiene il codice eseguibile
- **Sezione Dati**: contiene le variabili globali
- **Heap**: memoria allocata dinamicamente durante l'esecuzione del programma
- **Stack**: memoria temporaneamente utilizzatta durante le chiamate di funzioni

Un programma diventa un processo quando il file eseguibile è caricato in memoria.

Ogni **processo** è rappresentato nel sistema operativo da un ***blocco di controllo (PCB**, Process Control Block)* che contiene le informazioni connesse ad un certo processo:

- Un processo durante l'esecuzione è soggetto a cambiamenti del suo ***stato***, definito dall'attività corrente del processo stesso. Un processo può trovarsi in uno tra i seguenti stati:
  - **Nuovo**: Si crea il processo e attende di essere *ammesso*;
  - **Pronto**: Il processo attende, in una coda di attesa *(wait queue)*, di essere *accegnato* ad un'unità di elaborazione;
  - **Esecuzione**: Le sue istruzioni vengono eseguite;
  - **Attesa**: Il processo attende che si verifichi qualche evento *(come il comokletamento di un I/O o la ricezione di un segnale)*;
  - **terminato**: Il processo ha terminato l'esecuzione, a questo punto viene rimosso da tutte le code e vengono deallocati il suo ***PCB*** e le varie risorse.

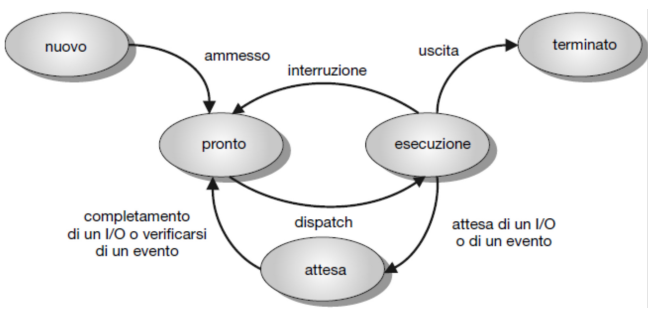

Figura 2.1: Diagramma di transizione degli stati di un processo

- **Programm Counter**: contiene l'indirizzo della successiva istruzione da eseguire per tale processo
- **Registri della CPU**: i registri variano in numero e tipo secondo l'architettura del calcolatore, essi comprendono accumulatori, registri indice, stack pointer, ecc.

  Quando si verifia un'interruzione della CPU si devono salvare tutte queste informazioni insieme al *PC*, in modo da permettere la corretta esecuzione del processo in un momento successivo, quando viene rischedulato.
- **Informazioni sullo scheduling della CPU**: comprendono le priorità del processo e tutti gli altri parametri di scheduling
- **Informazioni sulla gestione della memoria**: le tabelle delle pagine, valore dei registri di base e di limite
- **Informazioni di accounting**: comprendono la quota di uso della CPU e il tempo d'utilizzo della stessa, i limiti di tempo, i numeri del processo, ecc
- **Informazioni sullo stato della CPU**: comprendono la lista dei dispositivi di I/O assegnati a un determianto processo, l'elenco dei file aperti, ecc

Un ***thread*** è la più piccola unità di elaborazione che un sistema operativo può pianificare, è un sottoinsieme di un processo. L'utilizzo dei *thread* consente di sfruttare al meglio le capacità dei processori *multi-thread* e di migliorare l'efficienza dell'elaborazione corrente, utilizzando più *thread* per processo ed aggiungendo tutte le informazioni dei vari *threads* nel **PCB**.

2.0.1 Scheduling dei processi

L'obiettivo della multiprogrammazione consiste nell'avere sempre un processo in esecuzione in modo da massimizzare l'utilizzo della CPU. L'obiettivo del *time-sharing* è di commutare l'uso della CPU tra i vari processi così frequentemente che gli utenti possano interagire con ciascun programma mentre è in esecuzione. Per raggiungere questi obiettivi, lo **scheduler dei processi** selezione un processo da eseguire dall'insieme di quelli disponibili.

La maggior parte dei processori si può caratterizzare come avente una prevalenza di I/O (*I/O bound)* o come avente una prevalenza d'elaborazione (*CPU bound*). Un sistema di buone prestazioni presenta una cominazione bilanciata di processi CPU-bound e I/O-bound, se bilanciato:

- ready queue sempre vuota *(prevalenza di I/O boud)*
- I/O queue sempre vuota e quindi dispositivi inutilizzati *(prevalenza CPU bound)*

Code di scheduling

Ogni processo è inserito in una **coda di processi** pronti e in attesa di essere eseguiti *(ready queue)*. Questa coda generalmente viene memorizzata come una lista concatenata.

Una comune rappresentazione dello scheduling dei processi è data da un **diagramma di accodamento**, sono presenti due tipi di code: la *ready queue* e la *wait queue*.

1. Un nuovo processo si colloca inizialmente nella ready queue, dove attende finché non è selezionato per essere eseguito *(dispatched)*;

2. Una volta che il processo è assegnato alla CPU ed è nella fase di esecuzione, si più verificare uno dei seguenti eventi:

   - il processo può ammettere una richiesta di I/O e quindi essere inserito in una coda di I/O
   - il processo può creare un nuovo processo figlio e attenderne la terminazione
   - il processo può essere rimosso forzatamente dall CPU a causa di un'interruzione ed essere reinserito nella code dei processi pronti.

3. Un processo continua questo ciclo fino al termine della sua esecuzione, a questo pnto viene rimosso da tutte le code e vengono deallocati il suo PCB e le varie risorse.

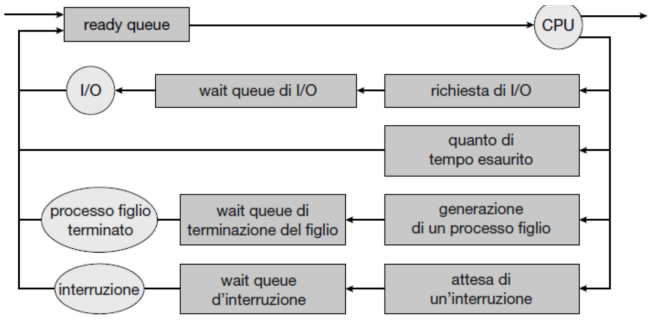

Figura 2.2: Diagramma di accodamento dei processi

2.0.2 Operazioni sui Processi

Durahte la propria esecuzone, un processo può creare numerosi nuovi processi. Il processo creante si chiama processo **padre** mentre il nuovo processo si chiama processo **figlio**, ciascuno di questi processi può creare a sua volta altri processi formando un ***albero*** di processi. In UNIX si usa la *fork()* per creare un nuovo processo, ogni processo ha un identificatore univoco ***PID*** *(Process Idetifier)*.

L'esecuzione del processo padre dopo la creazione del figlio può continuare in modo concorrente oppure attendere la terminazione del figlio.
Un nuovo processo può ottenere risorse in diversi modi:

- condivise con il padre
- solo un sottoinsieme delle risorse sono condivise col padre
- indipendenti dal padre *(ottenute dal sistema)*

Ci sono due possibilità per quel che riguarda lo **spazio d'indirizzi del nuovo processo**:

- il processo figlio è un duplicato del genitore *(ha gli stessi programmi e dati del genitore)*
- nel processo figlio si carica un nuovo programma

Entrambi i processi (genitore e figlio) continuano l'esecuzione all'istruzione successiva alla chiamata di sistema ***fork()***, con una differenza:

- la chiamata di sistema *fork()* restituisce il valore **zero** nel nuovo processo figlio

- riporta l'identificatore del processo figlio *(pid maggiore di zero)* nel processo genitore

Un processo **termina** quando finisce l'esecuzione della sua ultima istruzione ed inoltra la richiesta al SO di essere cancellato usando la **chiamata di sistema *exit()***.

Un processo genitore può porre termine all'esecuzione di uno dei suoi processi figli per diversu motivi:

- il processo figlio ha ecceduto l'uso di alcune tra le risorse che gli sono state assegnate
- il compito assegnato al processo figlio non è più richiesto
- il processo genitore termina ed il SO non consente ad un processo figlio di continuare l'esecuzione *(processo zombie, in questo caso i processi figli vengono "adottati" dal processo **init**)*

2.1 Comunicazione tra processi

I processi eseguiti concorrentemente nel sistema operativo possono essere *indipendneti* o **cooperanti** se possono o non possono influenzare/essere influenzati da altri processi in esecuzione nel sistema.

Per lo scambio di dati e informazioni i processi *cooperanti* necesitano di un meccanismo di ***comunicazione tra processi*** (**IPC**, interprocess communication). I modelli fondamentali della comunicazione tra processi sono due:

- a **memoria condivisa**: si stabilisce un'area di memoria condivisa dai processi cooperanti, che possono così comunicare scrivendo e leggendo da tale zona;
- a **scambio di messaggi**: la comunicazione ha luogo tramite scambio di messaggi tra i processi cooperanti.

In un sistema possono essistere entrambi i modelli.

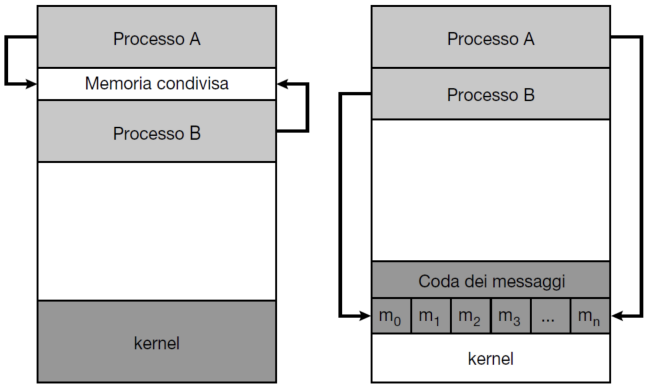

Figura 2.3: Modelli di comunicazione tra processi

2.1.1 IPC in sistemi a memoria condivisa

Di solito l'area di memoria condivisa risiede nello spazio degli indirizzi del processo che la alloca e gli altri processi che desiderano usarla per comunicare dovranno ammetterla nel loro spazio degli indirizzi *(attachment)*.

Caratteristiche memoria condivisa:

- **Il *SO* non controlla** il tipo e la collocazione dei dati sono determinati dai processi e l'accesso concorrente alla memoria;
- Condivisione di *variabili globali*;
- Condivisione *buffer di comunicazione*

Problemi memoria condivisa:

- *Identificazione dei processi comunicanti*, soprattuto in una condivisione di memoria in sistemi distribuiti, è difficile identificare i processi che accedono alla memoria;
- *Consistenza degli accessi*, evitare che più di un processo stia utilizzando l'area di memoria in lettura o scrittura;
- Richiede qualche *meccanismo di sincronizzazione* dei processi per assicurare che accedano uno alla volta *(**mutua esclusione**)*.

Un esempio di utilizzo di questo modello di comunicazione tra processi è il problema *produttore/consumatore*. Il **produttore** produce informazioni che sono consumate da un processo **consumatore**. Il *produttore e consumatore* possono condividere un **buffer**, ovvero un'area di memoria condivisa in cui il produttore inserisce le informazioni e il consumatore le preleva.

Il **buffer** utilizza una struttura dati appropiata *(es. coda circolare)* e ne esistono due tipi:

- buffer **illimitato** *(unbounded-buffer)*: non c'è un limite teorico alla dimensione del buffer e il **consumatore** deve aspettare se il buffer è vuoto e il **produttore** può sempre produrre nuovi elementi
- buffer **limitato** *(bounded-buffer)*: la dimensione del buffer è fissa e il **consumatore** deve aspettare se il buffer è vuoto e il **produttore** deve aspettare se il buffer è pieno

◉ Esempio di pseudo-codice del produttore, dove **buffer**, **in** e **out** sono variabili "condivise" con il consumatore

```
item next_produced;
while(True) {
    next_produced = /* Produce un elemento */ ;
    /* ATTENZIONE: manca meccanismo di sincronizzazione */
    while ((in+1)%BUFFER_SIZE == out)
        /* aspetta perché il buffer è pieno */
        ;
    buffer[in] = next_produced;
    in = (in+1)%BUFFER_SIZE; -> CODA CIRCOLARE
}
```

◉ Esempio di pseudo-codice del consumatore, dove **buffer**, **in** e **out** sono variabili "condivise" con il consumatore

```
item next_consumed;
while(True) {
    /* ATTENZIONE: manca meccanismo di sincronizzazione */
    while (in == out)
        /* aspetta perché il buffer è vuoto */
        ;
    next_consumed = buffer[out]
    out = (out+1)%BUFFER_SIZE;
    /* Usa l'elemento ottenuto dal buffer */
}
```

2.1.2 IPC in sistemi a scambio di messaggi

Nel primo modello è compito del programmatore realizzare la condivisione di un'area di memoria, invece nel secondo modello è il **SO** a *fornire ai processi appositi strumenti per lo scambio di messaggi.*

Lo scambio di messaggi è un meccanismo che permette a due o più processi di communicare e di sincronizzarsi senza condividere lo stesso spazio di indirizzi. Questo modello è particolarmente vantaggioso negli ambieni distribuiti, dove i processi possono risiedere su macchine diverse connesse da una rete.

Un meccanismo per lo scambio di messaggi prevede due operazioni: *send(message)* e *recive(message).*

Contenuto di un messaggio:

- Identificativo del processo mittente e del processo destinatario
- Informazioni da trasmettere *(payload)*
- Eventuali altre informazioni di gestione dello scambio *(es. priorità)*

Dimensione messaggio:

- **Fissa**: facile implementazione a livello SO, ma minore flessibilità per gli sviluppatori delle applicazioni
- **Variabile**: difficile implementazione a livello SO, ma maggiore flessibilità per gli sviluppatori delle applicazioni

Scambio "*message oriented*" o "*byte oriented*":

- *Byte oriented*: il canale di comunicazione è visto come un flusso continuo di byte (es. inviando due messaggi da 100 bytes, possono essere letti come un unico messaggio da 200 bytes, o come 4 messaggi da 50 bytes)
- *Message oriented*: viene preservata la separazione dei messaggi inviati sullo stesso canale

Indetificazione del canale:

- **Comunicazione diretta**: ogni processo ha un suo canale di comunicazione. Il mittende di un messaggio deve solo specificare il PID del destinatario;
- **Comunicazione indiretta**: i processi creano delle ***mailbox* (porte)** che rappresentano i canali di comunicazione.
  - Le operazioni **send** e **recive** specificano la *mailbox* su cui operare;
  - *Un processo* può avere *più mailbox*;
  - *Il mittente ed il destinatario devono condividere una mailbox per poter comunicare.*

Se tre processi condividono la stessa mailbox, uno invia un messaggio ad A e gli altri due lo ricevono, sorge il problema di sapere quale processo riceverà il messaggio. Esistono tre soluzioni:

1. si fa in modo che un canale sia associato al massimo a due processi;
2. si consente l'esecuzione di un'operazione di *recive()* ad un solo processo alla volta;

3. si consente al sistema di decidere arbitrariamente quale processo riceverà il messaggio tramite un algoritmo detto *round robin*.

Sincronizzazione, lo scambio dei messaggi può essere:

- **sincrono** *(bloccante)* per il mittente o per il destinatario (**Invio** bloccante o **Ricezione** bloccante);
- **asincrono** *(non bloccante)*:
  - il processo che invia riprende la sua esecuzione senza aspettare che il destinatario riceva il messaggio
  - il ricevente del messaggio acquisice un messaggio, se è disponibile, oppure l'indicazione che non ci sono messaggi, successivamente in tutti i casi riprende l'esecuzione.

Bufferizzazione, i messaggi scambiati tra processi risiedono in code temporanee (sia comunicazione diretta che indiretta), ci sono tre modi in cui i Sistemi Operativi possono implementare le code:

1. **Capacità zero**: il mittende deve bloccarsi finché il destinatario riceve il messaggio *(rendezvous)*;
2. **Capacità limitata**: c'è un numero limitato di messaggi, il mitende deve bloccarsi se la coda è piena;
3. **Capacità illimitata**: non c'è un limite al numero di messaggi, il mittente non si blocca mai

Nel primo caso si parla di **bufferizzazione esplicita o assente**, negli altri due di **bufferizzazione automatica**.

Possono esserci diverse politiche di ordinamento delle code dei messagi nella mailbox e nei processi in attesa: **LIFO**, **priorità**, **scadenza**.

I messaggi devono essere scambiati lungo un **canale di trasmissione** *(communication link)*, realizzabile in molti modi, ma trattiamo la sua *realizzazione logica*.

Realizzazione Memoria condivisa

Operazioni fondamentali:

1. *Allocazione di un'area di memoria codivisa*

   Ogni area di memoria condivisa è identificata da un **segment_id** (intero), per crearla si utilizza la syscall *shmget* in questo modo:

   - *segment_id = shmget(IPC_PRIVATE, size (byte), S_IRUSR, S_IWUSR);*
   - restituisce -1 in caso di errore
   - IPC_PRIVATE genera un nuovo identificatore
   - size, dimensione in byte dell'area di memoria da allocare
   - S_IRUSR | S_IWUSR, permessi di acesso di lettura e scrittura solo ai processo dello stesso utente che sta creando l'area di memoira
   - il processo che crea l'area di memoria deve comunicare in qualche modo il suo *segment_id* ad altri processi che lo devono utilizzare

2. *Collegamento dell'area di memoria condivisa allo spazio di indirizzi di un processo*

   Per usare un'area di memoria condivisa, un processo deve collegarla al suo spazio di indirizzamento:

   - *ptr = shmat(segment_id, NULL, 0);*
   - il valore restituito è un puntatore di tipo *(void *)* che può essere usato per accedere all'area di memoria, deve però prima essere convertito in un altro tipo *(casting esplicito)*;
   - restituisce *(void *)-1* in caso di errore;
   - NULL indica che il SO è libero di scegliere l'indirizzo a cui collegare l'area di memoria;
   - 0 indica i valori di default per i flag

3. *Scollegamento dell'area di memoria condivisa dllo spazio di indirizzi di un processo*

   Quando un processo ha finito di usare un'area di memoria condivisa, deve scollegarla dal suo spazio di indirizzamento:

   - *shmdt(ptr);*
   - Il valore restituito è 0 in caso di successo, -1 in caso di errore;
   - ptr è il puntatore che era stato restituito da *shmat*

4. *Deallocazione dell'area di memmoria condivisa*

   Un'area di memoria condivisa resta collegata **anche quando termina il processo che l'ha creata** (a differenza della memoria allocata con la *malloc*)

   - *shmctl(segment_id, IPC_RMID, NULL);*
   - 0 in caso di successo, -1 in caso di errore
   - *segment_id* è l'identificativo dell'area di memoria
   - IPC_RMID è l'operazione da effettuare *(remove)*
   - NULL è il puntatore ad un parametro che non è usato per l'operazione IPC_RMID

Implementazione scambio di messaggi

Unix offre diversi meccanismi, tra cui le **pipe anonime**, un canale di comunicazione:

- a **comunicazione indiretta**: un processo deve creare una pipe
- **byte oriented**: il sistema non preserva la delimitazione dei messaggi
- **Invio NON bloccante, Ricezione bloccante**, ma la ricezione potrebbe terminare con un messaggio incompleto.

Creazione di una pipe anonima

- La syscall **pipe** crea una pipe anonima:
- *int fd[2];* retval = pipe(fd);
- retval = 0 in caso di successo, altrimenti -1

- Il parametro di uscita fd è usato per ricevere due **file descriptor** che serviranno per accedere alla pipe:
- *fd[0]* per **leggere** le informazioni dalla pipe
- *fd[1]* per **scrivere** le informazioni sulla pipe
- i **file descriptor** possono essere usati solo nel processo che li ha creati, perchè ogni processo ha uno spazio dei file descriptor separato
- C'è un'**importante eccezione** quando un processo esegue la *fork* il processo figlio riceve una copia di tutta la memoria e quindi anche dei file descriptor del padre.
- Quindi il processo padre **deve creare la pipe prima di eseguire la *fork()***
- non è possibile usare una pipe anonima per far comunicare due processo che non siano legati da una relazione **padre-figlio** *(dirette o indirette)*.

Chiusura di una pipe anonima

- Un proceso può chiudere un file descriptor con la syscall: *close(fd);*
- è possibile chiudere separatamente il **fd** di lettura e di scrittura;
- ogni proceso che usa i *fd* deve chiuderli quando ha terminato di usarli;
- i *fd* vengono comunque chiudi automaticamente quando il processo termina *(a differenza della memoria condivisa che non viene deallocata)*.

Scrittura su una pipe

- *write(fd_wr, message_ptr, size);*
- *fd_wr* è il file descriptor di scrittura della pide (fd[1]);
- *message_ptr* è il **puntatore** all'area di memoria che contiene il messaggio;
- *size* è la dimensione del messaggio in byte;
- il valore di ritorno è il numero di byte inviati, in caso di errore sarà minore di size.

Lettura di una pipe

- *read(fd_rd, message_ptr, size);*
- *fd_rd* è il file descriptor di lettura della pipe (fd[0]);
- *message_ptr* è il **puntatore** all'area di memoria in cui verrà inserito il messaggio letto;
- *size* è la dimensione del messaggio in byte;
- il valore di ritorno è il numero di byte ricevuti:
  - -1 in caso di errore
  - 0 se la pipe è terminata *(tutti i processi hanno chiudo il fd di scrittura)*
  - **in caso di successo il numero di byte ricevuto può comunque essere minore di size, il programma deve gestire questa situazione**.

Capitolo 3

I Threads

Un thread è l'unità di base d'uso della cpu e comprende:

1. un identificatore di thread (id),
2. un contatore di programma,
3. un insieme di registri,
4. una pila (stack).

Condivide con gli altri thread che appartengono allo stesso processo la **sezione del codice**, la **sezione dei dati** e altre **risorse di sistema**, come i *file* aperti e i segnali. Un processo tradizionale, chiamato anche processo pesante (*heavyweight* process), è composto da un solo thread. Un processo multithread è in grado di svolgere più compiti in modo concorrente (*lightweight process*). Ogni thread è rappresentato da un **thread control block** (***TCB***)

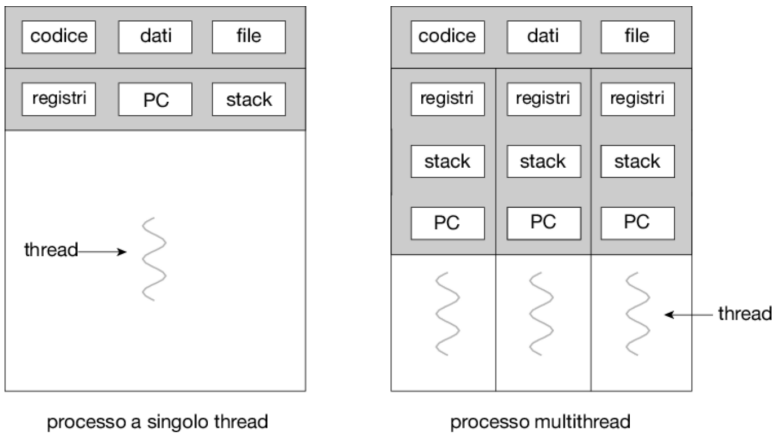

Figura 3.1: Processi a singolo thread e multithread

all'interno del ***PCB*** del processo a cui il thread appartiene. Il TCB contiene il PC e i registri (quando il thread non è in esecuzione) e altre informazioni che servono al sistema operativo per gestire il thread (ad esempio: priorità, stato del thread).

In un sistema multithread, quando viene creato un processo viene anche creato un primo thread detto **thread principale**. Questo thread può crearne altri durante l'esecuzione del programma ed ogni thread ha il suo ***status:*** pronto, in esecuzione, attesa.

Un processo termina la sua esecuzione (e quindi tutti i thread attivi in quel momento) quando il suo thread principale ritorna dal main, oppure uno qualsiasi dei thread richiama la system call exit

3.1 Vantaggi

Context switch: Thread vs Process

Nel cambio di contesto tra due thread, così come nel cambio di contesto tra due processi nei sistemi a singolo thread, il processore deve:

- Salvare in un'area di memoria (TCB o PCB) il suo stato
- Eseguire il codice per scegliere il prossimo thread/processo da portare in esecuzione
- Caricare da un'area di memoria (TCB o PCB) uno stato precedentemente salvato

Tuttavia, nel passaggio da un processo all'altro, il processore deve anche cambiare le informazioni che usa per la gestione dello spazio di indirizzamento, le informazioni nel TCB, nella cache ecc.

Quindi, il passaggio tra due thread dello stesso processo è significativamente più veloce del passaggio tra due processi diversi.

I vataggi della programmazione multithread

I vataggi della programmazione multithread si possono classificare su quattro categorie pricipali:

1. **Tempo di risposta**: Un programma può continuare la computazione anche se un suo thread è bloccato (e.g. attesa I/O)
2. **Condivisione delle risorse**: I processi possono condividere risorse soltanto attraverso tecniche come la memoria condivisa e lo scambio di messaggi. Queste tecniche devono essere esplicitamente messe in atto dal programmatore. Tuttavia, i thread condividono per default la memoria e le risorse del processo al quale appartengono. Il vantaggio della condivisione del codice e dei dati consiste nel fatto che un'applicazione può avere molti thread di attività diverse, tutti nello stesso spazio d'indirizzi.
3. **Economia**: Assegnare memoria e risorse per la creazione di nuovi processi è costoso; poiché i thread condividono le risorse del processo cui appartengono, è molto più conveniente creare thread e gestirne i cambi di contesto. È difficile misurare empiricamente la differenza nell'overhead richiesto per creare e gestire un processo invece che un thread, tuttavia la creazione dei thread richiede in generale meno tempo e meno memoria, e il cambio di contesto tra thread è più rapido.
4. **Scalabilità**: I vantaggi della programmazione multithread sono ancora maggiori nelle architetture multiprocessore, dove i thread si possono eseguire in parallelo su distinti core di elaborazione. Invece un processo con un singolo thread può funzionare solo su un processore, indipendentemente da quanti ve ne siano a disposizione. Il multithreading su una macchina con più processori incrementa il parallelismo.

I svantaggi dei thread:

- **Difficoltà di ottenere risorse private**: per ottenere memoria privata per un thread bisogna utilizzare meccanismi particolari
- **Pericolo di interferenza**: la condivisione delle risorse accentua il pericolo di interferenza, dunque gli accessi concorrenti devono essere sincronizzati (thread safety)

Nella **programmazione multicore** le applicazioni devono essere progettate tenendo conto di diversi fattori:

- **Separazione dei task**: individuazione dei task che possono essere eseguiti in parallelo
- **Bilanciamento**: i task devono eseguire compiti che richiedono all'incirca tempo e risorse paragonabili
- **Suddivisione dei dati**: devono essere definiti dei dati specifici per i vari task
- **Dipendenza dei dati**: l'accesso ai dati condivisi deve essere fatto in modo sincronizzato (thread safety)
- **Test e debugging**: a causa dei diversi flussi di esecuzione test e debugging sono più difficili

Tipi di parallelismo:

- **per dati**: distribuzione dei dati su più core di elaborazione e l'esecuzione della stessa operazione su più core;
- **per attivà**: Ogni thread realizza un'operazione distinta e thread differenti possono operare sugli stessi dati o su dati diversi.

3.1.1 Modelli di supporto al multithreading

I thread si distinguono in thread a livello **utente** e thread a livello **kernel**: i primi sono gestiti interamente all'interno dello spazio utente del processo, ciò significa che il SO viene coinvolto soltanto tramite syscall; invece i secondi sono gestiti direttamente dal kernel del SO, il kernel assegna le risorse e schedula i thread per l'esecuzione.

Kernel-Level multithreading: gestione dei thread affidata al *kernel* tramite *chiamate di sistema*.

- Il kernel definisce e gestisce le strutture dati per i thread (TCB)
- Lo scheduler del kernel determina il passaggio da un thread all'altro
- Le operazioni sui thread (es. creazione) sono richieste tramite system call
- Possibile solo se il sistema operativo supporta il multithreading nel suo kernel (ma oggi questo è vero per tutti i SO comunemente usati)

User-level multithreading: thread all'interno del processo **gestiti da una libreria di funzioni**.

- le strutture dati per i thread (TCB) sono definite e gestite dalla libreria nella memoria normale del processo
- una funzione della libreria determina il passaggio da un thread all'altro
- le operazioni sui thread sono realizzate tramite normali chiamate di funzione
- un processo può attivare più thread anche senza il supporto del SO

La combinazione di questi due meccanismi consente di realizzare diversi **modelli di esecuzione** di un'applicazione di multithreading.

In questi modelli cambia il modo in cui i threads visti dall'applicazione (*user-level threads*) sono associati ai threads visti dal sistema operativo (*kernel-level threads*).

- **Modello molti-a-uno**: realizza molti threads utente associati ad un unico thread kernel principale.

  Questo modello è vantaggioso in quanto tutti i thread sono gestiti dalle librerie a livello dell'applicazione ma l'intero processo rimane bloccato se un thread invoca una chiamata di sistema di tipo bloccante perchè un solo thread può accedere al kernel.

- **Modello uno-ad-uno**: mette in corrispondenza di ciascun thread a livello utente un thread a livello kernel, è vantaggioso perchè offre un grado di concorrenza maggiore ma compromette le prestazioni dell'applicazione, per questo viene aggiunto un limite di thread creabili.

- **Modello molti-a-molti**: una soluzione ibrida tra i due modelli, il numero massimo di thread a livello utente associabili ai thread a livello kernel dipende dal tipo di architettura.

3.2 Memoria condivisa e sincronizzazione tra i processi

La programmazzione multicore ha introdotto la possibilità per due o più processi di eseguire istruzioni in parallelo. Tuttavia l'esecuzione contemporanea contribuisce a problematiche che riguardano l'integratià dei dati condivisi da più processi.

Uno dei meccanismi di comunicazione tra procesi o threads prevede l'uso di strutture dati condivise per la **memoria condivisa**:

- thread dello stesso processo -> tutta la memoria è condivisa
- processi diversi -> il SO deve fornire delle system call per condividere un'area di memoria

Un tipico problema è quello descritto dal modello produttore consumatore per l'accesso contemporaneo ad un buffer limitato. Si potrebbero verificare situazioni in cui più processi accedono e modificano gli stessi dati in modo concorrente ed i risultati dipendono dall'ordine degli accessi (**race condition**).

Una **race condition** si verifica quando i processi hanno accesso concorrente ai dati condivisi ed il risulato finale dipende dal particolare ordine in cui si verificano gli accessi. Le race conditions possono portare a valori corrotti dei dati condivisi.

3.2.1 Probelema della sezione critica

In un sistema composto da più processi ciascuno potrebbe avere un segmento di codice, detta **sezione critica**, in cui il processo può modificare variabili comuni. Il *problema della sezione critica* consiste nel progettare un protoccolo dove ogni processo deve chiedere il permesso per entrare nella propria sezione critica. La sezione che realizza la richiesta è detta **sezione d'ingresso**, la sezione critica può essere seguita da una **sezione d'uscita** e la restante parte è detta **sezione non critica**.

Una soluzione del problema della sezione critica deve soddisfare i tre requisiti:

1. **Muta esclusione**: Se un processo è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica;

2. **Progresso**: Se nessun processo è in esecuzione nella propria sezione critica e e qualche processo desidera entrare nella propria sezione critica, deve essere effettuata una scelta che non si può rimandare indefinitamente per chi deve entrare per primo;

3. **Attesa limitata**: Se un processo ha già richiesto l'ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.

Una *prima soluzione* sarebbe quella di fare in modo che un processo running conservi la CPU per tutta la durata della sezioni critica, disabilitando le interruzioni. Ovviamente la soluzione è banale ma svantaggiosa ed è **adatta solo per le sezioni critiche interne al kernel**.

Soluzioni software al problema della sezione critica non funzionnano bene con le modere architetture elaborative, il *supporto hardware* include invece istruzioni hardware come *test_and_set* e le operazioni atomiche (*non interrompibili*).

Una *seconda soluzione* è quella di utilizzare la funzione *test_and_set* dove viene garantito che durante l'esecuzione nessun altro processo possa accedere alla stessa sezione. Se un processo è già in esecuzione della sezione critica, un altro processo rimarrà a girare nel ciclo d'ingresso della sezione critica. Per questo, il meccanismo viene chiamato **spinlock**.

La mutua esclusione attraverso gli spin lock è garantita ma non soddifa il requisito dell'*attesa limitata*. I processi in attesa utilizzano continuamente la CPU per verificare se la condizione è vera (**attesa attiva** o **busy waiting**).

L'**attesa attiva** è impiegata *solo in piccole sezioni del kernel di un sistema multiuprocessore* ma per i *programmi eseguiti in modo utente* è preferibile un meccanisco che *consenta al kernel di mettere il processo che aspetta in stato di Waiting* e riportarlo nello stato *Ready solo quando la condizione attesa si è verificara.*

I Mutex

La soluzione hardware precedente non è quindi accessibile ai programmatori di applicazioni, per questo in alternativa si implementano strumenti software per risolvere il problema. Il più semplice tra questi strumenti è il **mutex** (***mut****ual* ***ex****clusion*).

In pratica:

- **ad ogni area di memoria condivisa viene associato un mutex diverso** e le operazioni di *inizio e fine* della sezione critica divenato (*Wait(mutex) - Signal(mutex)*)

- il sistema operativo garantisce che le operazioni sul mutex si comportino come operazioni atomiche. Perciò:

  - se due processi cercano di entrare nella sezione critica, uno troverà il mutex libero ed andrà avanti, l'altro troverà il mutex bloccato e verrà sospeso

  - quando il primo processo esce dalla sezione critica, la sua *Signall* risveglia l'altro processo che può entrare nella sezione, il mutex resta bloccato

  - quando il secondo processo esce dalla sezione critica, la sua *Signal* rende il mutex libero

Produttore-consumatore con i mutex

```
typedef struct {
 MutexType mutex;
 int primo, ultimo, cont;
 Info elem[MAX];
} Buffer;
```

BUG
- ACCESSO AD UNA VARIABILE CONDIVISA FUORI DALLA SEZ. CRITICA

```
void produttore(Buffer *p) {
 Info nuovo;
 while (True) {
  nuovo = . . .; // crea info
  while (p->cont==MAX)
   ; // aspetta buffer
  Wait(p->mutex);
  p->elem[p->ultimo] = nuovo;
  p->ultimo = (p->ultimo + 1) % MAX;
  p->cont++;
  Signal(p->mutex);
 }
}
```

PROBLEMA
ATTESA ATTIVA

```
void consumatore(Buffer *p) {
 Info val;
 while (True) {
  while (p->cont==0)
   ; // aspetta info
  Wait(p->mutex);
  val = p->elem[p->primo];
  p->primo = (p->primo + 1) % MAX;
  p->cont--;
  Signal(p->mutex);
  . . . // usa l'info
 }
}
```

BUG

Le variabili condition

Occorre un meccanismo che consenta di mettere in attesa un processo fino a che si verifica una condizione sulla struttura dati:

- il meccanismo deve consentire di controllare se la condizione è vera all'interno della regione critica
- ma se il processo deve essere messo in attesa, il meccanismo deve garantire che durante l'attesa la regione critica **non** venga mantenuta bloccata, altrimenti gli altri processi non potranno rendere vera la condizione
- per poter gestire quest'esigenza sono stato introdotte le **variabili condition**

Le operazioni primitive definite su una variabile condition sono:

- ***Wait(condvar, mutex)***: esegue una *Signal* sul mutex e sospende il processo chiamante in attesa della variabile condition *"condvar"*
- ***Sigal(condvar)***: se ci sono processi in attesa su *condvar*, ne risveglia uno, il processo risvegliato esegue *implicitamente* una Wait su mutex
- ***Broadcast(condvar)***: se ci sono processi in attesa su *condvar*, li risveglia tutti, i processi risvegliati eseguono implicitamente una *Wait* sul mutex

Note:

- **Un processo può effettuare la Wait su una condition solo se ha effettuato la Wait sul mutex associato**, e quindi si trova nella sezione critica
- Il funzionamento della *Signal* (o Broadcast) garantisce che quando viene eseguita l'istruzione successiva alla chiamamta di Wait, il processo si trova nella sezione critica
- Se il processo che esegue la *Signal* (o Broadcast) sulla variabile condition si trova nella regione critica, può essere sicuro della muta esclusione fino a quando non esegue la Signal sul mutex

---

- Vediamo come è possibile usare le variabili condition per implementare l'attesa di una condizione. Una prima soluzione:

```
Wait(mutex); // entra nella sezione critica
if (!condizione)
        Wait(condvar , mutex); // aspetta la condizione
. . . // continua con la sezione critica
```

- **Sfortunatamente, in questo codice c'è un race**: nell'intervallo di tempo tra quando il processo viene risvegliato dalla condition e quando acquisisce il mutex, e quindi rientra nella sezione critica, un altro processo potrebbe rientrare nella sezione critica e rendere la condizione di nuovo falsa. Perciò il modo corretto di realizzare l'attesa è:

```
Wait(mutex); // entra nella sezione critica
while (!condizione)
        Wait(condvar , mutex); // aspetta la condizione
. . . // continua con la sezione critica
```

Scegliere tra l'uso di *Signal* o *Broadcast*:

- usare **Signal** solo se si è assolutamente sicuri che la condizione che si è verificata consente ad un solo processo di andare avanti con la sua operazione, e che uno ***qualunque*** dei processi in coda può proseguire
- usare **Broadcast** se la condizione verificata può consentire il proseguimento di più processi, oppure se i processi in attesa **non sono equivalenti e ciascuno può proseguire solo in un sottoinsieme dei casi**
- nel dubbio, **usare Broadcast**, si rischia un programma meno efficiente ma con meno probabilità d'errore

Produttore-consumatore con variabili condition

```
typedef struct {
    MutexType mutex;
    ConditionType non_pieno,
                  non_vuoto;
    int primo , ultimo , cont;
    Info elem[MAX];
} Buffer;
```

```
void produttore(Buffer *p) {
 Info nuovo;
 while (True) {
  nuovo = . . .; // crea info
  Wait(p->mutex);
  while (p->cont==MAX)
    Wait(p->non_pieno , mutex);
  p->elem[p->ultimo] = nuovo;
  p->ultimo = (p->ultimo + 1) % MAX;
  p->cont++;
  Signal(p->non_vuoto);
  Signal(p->mutex);
 }
}
```

```
void consumatore(Buffer *p) {
 Info val;
 while (True) {
  Wait(p->mutex);
  while (p->cont==0)
   Wait(p->non_vuoto , mutex);
  val = p->elem[p->primo];
  p->primo = (p->primo + 1) % MAX;
  p->cont --;
  Signal(p->non_pieno);
  Signal(p->mutex);
  . . . // usa l'info
 }
}
```

3.3 Errori nella sincronizzazione: deadlock

Errori nella sincronizzazione dell'accesso a risorse codivise possono portare, oltre alle condizioni di race, ai seguenti problemi:

- **livelock**: un singolo processo monopolizza una risorsa condivisa non consentendo agli altri processi di avanzare
- **deadlock**: tutti i processi che condividono un inseme di risorse sono bloccati; ciascun processo deve aspettare che un altro processo rilasci una risorsa, ma poiché tutti sono in attesa, nessuno può avanzare e rilasciare le risorse che sta usando

La *causa* del **deadlock** è la **catena circolare**, un modo per evitare i deadlock è gestire l'assegnazione delle risorse in modo da avere la **certezza** che un deadlock non si possa verificare (***deadlock prevention***.

Per garantire la deadlock prevention bisogna applicare delle regole nel modo in cui i processi chiedono l'accesso alle risorse, se tutti i processi seguono queste regole si dimostra che non possono essere costruite catene circolari di attesa.

Regole (*allocazione monotona*):

1. le risorse sono ordinate secondo un criterio di ordinamento globale: $R_1 < .. < R_i < .. < R_n$;
2. non importa qual è il criterio di ordinamento, basta che ci sia un criterio condiviso da tutti i processi;
3. un processo non può richiedere una risorsa R_i se posside già una risorsa $R_j > R_i$;
4.

Capitolo 4

Scheduling della CPU

Lo scheduling della CPU è alla base di sistemi operativi multiprogrammati: attraverso la commutazione della CPU tra i vari processi, il SO può rendere più produttivo il calcolatore.

In effetti sono i **thread**, e *non i processi*, l'oggetto dell'*attività di scheduling*. Ciononostante, le locuzioni di *scheduling dei processi* e *scheduling dei thread* sono spesso considerate equivalenti.

Quando parliamo di *sheduling* di un processo che deve essere "eseguito su una CPU" stiamo sottintendendo che il processo sarà in esecuzione su un core della CPU.

L'**obiettivo dello scheduling** è realizzare la turnazione dei processi sul processore in modo da:

- *massimizzare* l'utilizzazione della CPU
- creare l'illusione di evoluzione contemporanea dei processi in sistemi ***time-sharing***

Ciclicità delle fasi d'elaborazione e di I/O

Il successo dello scheduling della CPU dipende dalla proprietà dei processi di essere sempre in un *ciclo di elaborazione* (svolto dalla CPU) e d'*attesa di completamento delle operazioni di I/*O.

Dunque, l'**esecuzione di un processo** è una sequenza alternata di:

- picchi (*cicli*) d'esecuzione di CPU - **CPU burst**
- di attesa di I/O - **I/O burst**
- Infine la *richiesta du terminare l'esecuzione*

Questa distribuzione dell'operazioni di CPU o I/O sono utili per la scelta di un appropriato algoritmo di scheduling della CPU:

- Un processo *CPU-bound* ha pochi picchi lunghi
- Un processo *I/O-bound* ha molti picchi brevi

4.0.1 Scheduler della CPU

Ogni volta che la CPU passa nello stato d'inattività, il SO sceglie per l'esecuzione uno dei processi presenti della *ready queue*. In particolare, è lo **scheduler a breve termine** che, tra i processi in memoria pronti per l'esecuzione, sceglie quello cui assegnare la CPU.

La *coda dei processi pronti* non è necessariamente una cosa in ordine d'arrivo (FIFO). Tuttavua, concettualmente tutti i processi della ready queue sono posti nella lista d'attesa per accedere alla CPU. Generalmente gli elementi delle code sono i *process control block* (PCB) dei processi.

Scheduling con e senza prelazione

Le decisioni riguardanti lo scheduling della CPU vengono prese nelle seguenti circostanze:

1. un processo passa **dallo stato di esecuzione → allo stato di attesa**(*per esempio, richiesta di I/O*
   stato di pronto(*es, quando si verifica un interrupt*)
2. **stato di attesa → stato di pronto**(*es, completamento di un'operazione di I/O*)*quandounprocesso***termi**
   La schedulazione dei casi 1 e 4 è detta **non-preemptive** (*senza sospensione dell'esecuzione*) perchè non danno alternative in termini di scheduling: si deve comunque scegliere un nuovo processo da eseguire, sempre che ce ne sia uno nella ready queue per l'esecuzione.

   Un scelta si può invece fare nei casi 2 e 3. Lo schema dei scheduling in questi casi è **preemptive** (*con prelazione, sospensione dell'esecuzione*).

3. La schedulazione **non-preemptive** è tipica della realizzazione del **multi-tasking**

- La schedulazione **preemptive** è alla base del **time sharing**
- Concetto di *time sharing*:
  - **Quanto di tempo** (*time slice*): intervallo di tempo massimo d'uso consecutivo del processore consentito a ciascun processo
  - **Pre-rilascio** (*pre-emption*): un processo può essere sospeso prima che termini il quanto di tempo
- La schedulazione **preemptive** richiede l'**uso** di *meccanismi di sincronizzazione.*

Caricamento del processo sulla CPU

Il **dispatcher** (*caricatore della CPU*) è il modulo del SO che dà il controllo della CPU ad un processo selezionato dallo *schedulatore a breve termine.* Questa funzione comprende:

- **contex switch**: sospensione del processo in esecuzione ed attivazione del processo da mettere in esecuzione
- *passaggio alla modalità utente*
- *salto* alla corretta locazione del programma utente per *ricominciare l'esecuzione*

4.0.2 Criteri di scheduling

Prima di scegliere l'argoritmo da usare in una specifica situazione occorre considerare le caratteristiche dei diversi algoritmi. Le caratteristiche usate per tale confronto possono incidere pesantemente sulla scelta dell'algoritmo migliore. Alcuni criteri importanti sono:

- **utilizzo della CPU**: la CPU deve essere attivata il più possibile, dovrebbe variare dal 40% al 90%
- **Frequenza di completamento - Troughput**: numero di processi completati per unità di tempo
- **Tempo di completamento - turnaround time**: intervallo che va dal momento dell'immissione del processo nel sistema al momento del completamento, *comprende tempi di esecuzione, tempi di attesa nelle varie code*

- **Tempo di attesa**: somma dei tempi speci in attesa nella coda dei processi pronti, solo tempi d'attesa non di esecuzione
- **Tempo di risposta**: tempo che intercorre dalla formulazione della richiesta fino alla produzione della prima risposta, si conta il tempo necessario per iniziare la risposta, non per emetterla completamente.

In generale si ottimizza:

- il **valor medio**
- il **valore minimo/massimo**
- la **varianza**

4.1 Algorimti di Scheduling

Il termine di paragone considerato sarà il **tempo di attesa medio** e la **varianza**. Descriviamo questi algoritmi di scheduling nel caso di un solo core di elaborazione disponibile, ovvero di una singola CPU con un singolo core di elaborazione, per cui il sistema è in grado di eseguire un solo processo alla volta.

- **Fist-Come, First-Served** (*FCFS*): non-preemptive.
  - processi schedulati in ordine di arrivo
  - il primo ad entrare in coda è il primo ad essere arrivato
  - i processi lasciano la CPU solo di spontanea volontà
  - essendo non-preemptive è svantaggioso per i sistemi *time-sharing*
  - c'è un effetto di ritardo a catena (convoy effect) mentre i processi brevi (*I/O-bound*) attendono che quello grosso (*CPU-bound*) rilasci la CPU
- **Shortest-Job-First** (*SJF*): algoritmo **ottimale**, fornisce il minor tempo d'attesa medio per gruppo di processi. Li ordina in base alla durata del prossimo picco, due modalità:
  1. **non-preemptive**: l'algoritmo permette al processo corrente di finire il suo uso della CPU
  2. **preemptive** (*shortest-remaining-time-first - SRTF*): quando un processo arriva nella coda dei processi pronti con un tempo di computazione minore del tempo che rimane al processo corrente in esecuzione, l'agorirmo ferma il processo.

  SJF è tipicamente usato per schedulatori a **lungo termine**, negli schedulatori di *breve termine* può essere usato cercando di predire il tempo di esecuzione
  - SJF ottimale, ma ideale in quanto non è possibile conoscere la lunghezza del successivo picco di CPU.
  - Si può stimare usano una **media esponenziale** delle lunghezze dei picchi di CPU precedenti:
    * t_n = lunghezza attuale dell'n-mo piccho di CPU
    * τ_{n+1} = valore previsto per il prossimo picco di CPU, τ_0 costante
    * $\alpha, 0 \leq \alpha \leq 1$ è il peso della nostra predizione, solitamente $\alpha = 1/2$

Quindi: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

- **Schedulazione a priorità**:
  - Si associa una priorità numerica a ciascun processo, la CPU è allocata al processo con priorità più alta
  - **preemptive o non-preemptive**
  - SJF è un algoritmo con priorità dove la priorità è l'inverso del prossimo picco previsto di CPU

  Le priorità posssono essere definite in base a fattori *interni* o *esterni*.

  Un problema potrebbe essere la presenza di un blocco indefinito (*starvation*), un processo che viene sempre superato da processi a priorità più alta.

  La soluzione è l'invecchiamento (aging), accresce gradualmente la priorità di un processo che attende nel sistema per un lungo periodo

- **Circolare - Round Robin** (*RR*): *FCFS + preemption* per alternare i processi (*coda FIFO circolare*)
  - Ogni processo possiede un quanto di tempo (**time splice**) q, generalemente $q = 10 - 100ms$, di utilizzo della CPU.
  - se ci sono n processi della coda allora:
    - ciascun processo ottiene 1/n del tempo della CPU
    - ciascun processo deve attendere più di $(n - 1)q$ unità di tempo
  - Nuovi processi vengono aggiunti alla fine della coda dei processi pronti

  Se q è grande le prestazioni sono simili a FCFS, con q piccola si ha l'impressione di parallelismo.

  Però è come se ogni processo abbia a disposizione una cpu n volte più lenta.

  Per la scelta di q, quanto di tempo, si segue questa regola:
  - l'80% dei picchi di CPU deve essere più breve del quanto di tempo

- **Code multilivello**: è adatto in situazioni in cui i porcessi possono essere divisi in gruppi.

  La coda dei processi è ripartita in code separate, ed ogni coda è gestita con il relativo algoritmo di scheduling:

  1. **system process**:
  2. **interactive process** (*foregrounf*): RR
  3. **interactive editing process** (*foregrounf*): RR
  4. **batch process** (*backgrounf*): FCFS
  5. **student process**:

  Ci deve essere una schedulazione anche tra le code:
  - tipicamente una schedulazioe preemptive a priorità fissa
  - la coda dei processi in foreground ha priorità assoluta su quelli in background

  - **possibilità di *starvation***

  Altrimenti, vi è una *partizione del tempo tra le code*: ciascuna coda ha una certa quantità di tempo di CPU che può schedulare fra i processi in essa contenuta. Ad esempio:

  - il **foreground** ha l'80% del tempo di CPU per la schedulazione **RR**
  - il **background** riceve il 20% della CPU da dare ai suoi processi sencondo l'agoritmo **FCFS**

- **Code multilivello con retroazione**: un processore può muoversi tra le varie code, **prevenendo la *starvation***.

  Uno schedulatore con coda a più livelli con *feedback* è definito dai seguenti parametri:

  - numero di code
  - algoritmo di scedulazione per ciascuna coda
  - metodo utilizzato per determinare quando promuovere un processo verso una coda a prorità più alta, per esempio se ha atteso troppo
  - metodo utilizzato per determinare quando degradare un processo in una coda a più bassa priorità, per esempio quando è troppo CPU-bound
  - metodo utilizzato per determinare in quale coda entrerà un processo quando avrà bisogno di un servizio

  Un esempio con tre code potrebbe essere:

  - Q0 - RR con quanto di tempo 8ms
  - Q1 - RR con $q = 16ms$
  - Q2 - FCFS

  Schedulazione:

  - Un nuovo processo "pronto" entra inizialmente nella cosa Q0
  - i processi lunghi affondano automaticamente nella coda Q2 e sono serviti in ordine FCFS utilizzando i cicli di CPU lasciati dalle code Q0 e Q1 (quando sono vuote)

- **Schedulazione multiprocessing**: con più CPU la schedulazione diviene più complessa.

  Si ipotizza di avere processori omogenei con suddivisione del carico (load sharing), osserviamo due *approcci di schedulazione*:

  1. **Multiprocessamento asimmetrico** (*Master-Slave*): solo un processore (master) prende le decisioni relative allo scheduling. GLi altri processori fanno solo elaborazione
  2. **Multiprocessamento simmetrico** (*SMP*): ciascun processore schedula se stesso selezionando un processo dalla coda comune dei processi pronti o da una cosa specifica per se stesso

Capitolo 5

Il File System

Il File Sytem è la parte del SO che si occupa delle informazioni immagazzinate nei dispositivi di memoria di massa.

Le informazioni sono organizzare in due entità diverse:

- un insieme di **file** contenenti informazioni
- una struttura delle **directory** che organizza i file in una struttura gerarchica e fornisce informazioni sui file stessi

Il file sytem forinisce gli strumenti per:

1. la **creazione** e **manipolazione** dei *file* e delle *directory*
2. garantire l'**affidabilità** dei file quando si verificano guasti
3. permette la **protezione** dei file e la **condivisione** tra gli utenti (anche per l'accesso concorrente)

Il termine "**file system**" è usato in due fasi diverse:

1. La parte del SO che implementa le **operazioni** sulla memoria di massa
2. L'insieme delle **strutture dati** presenti sul dispositivo di memoria di massa, che servono al sistema operativo per realizzare queste *operazioni*.
    - le strutture dai vengono create con la **formattazione** della memoria di massa
    - un SO in genere supporta diversi File System, nel senso che al momento della formattazione si può scegliere quale File System utilizzare (FAT32 - File Allocation Table, NTFS per Windows; Ext2, Ext3, Ext4 per Linux)

Il File system risiede *permanentemente* in memoria di massa.

Per migliorare l'efficienza dell'I/O, i trasferimenti tra memoria centrale e dischi si eseguono per **blocchi**. Ciascun ***blocco*** è composto da *uno o più settori*. La dimensione dei blocchi varia, da 32 a 4096 byte (solitamente 512 byte).

5.1 Scrittura del file system

Il File System è generalemnte composto da molti livelli distinti: il livello superiore si basa sui servizi offerti dal livello inferiore. I vari livelli sono:

1. **File System logico**: Il livello logico del file system gestisce la struttura gerarchica dei file e delle directory e i metadati associati ai file. Le sue funzioni includono:

   - *Gestione dei metadati*: mantiene informazioni sui file come dimensione, permessi, timestamp, e così via.
   - *Struttura delle directory*: organizza i file in una struttura a directory, permettendo una navigazione logica e intuitiva attraverso il sistema di file.
   - *File Control Blocks (FCB)*: strutture dati che contengono gli attributi dei file. Ogni file ha un FCB che memorizza informazioni come nome del file, dimensione, posizione dei dati sul disco, permessi di accesso, ecc.
   - **Modulo di organizzazione dei file**: Questo modulo gestisce l'allocazione dei file sul disco e la traduzione degli indirizzi logici in indirizzi fisici. Le sue componenti includono:
     - *Metodo di allocazion*e: sistema che determina come i file vengono distribuiti sul disco (ad esempio, allocazione contigua, a liste concatenate, indicizzata).
     - *Traduzione degli indirizzi*: converte gli indirizzi logici (utilizzati dai programmi) in indirizzi fisici (dove i dati sono realmente memorizzati sul disco).
     - *Gestore dello spazio libero*: mantiene traccia dei blocchi di memoria che sono disponibili per essere utilizzati. Questo è spesso realizzato attraverso una lista dei blocchi liberi.
   - **File System di base**: Il file system di base interagisce direttamente con il controllo dell'I/O per eseguire operazioni fondamentali sui dati. Le sue funzioni principali includono:
     - Comandi di base: invia comandi per leggere e scrivere blocchi fisici di dati sulla memoria di massa.
     - *Gestione dei blocchi fisici*: organizza e mantiene la coerenza dei dati a livello di blocchi fisici.
   - **Controllo dell'I/O**: Questo livello si occupa della gestione diretta dell'hardware del dispositivo di memoria. Include:
     - Driver dei dispositivi: software che permette al sistema operativo di comunicare con l'hardware.
     - *Gestore degli interrupt*: gestisce le interruzioni che si verificano quando un dispositivo richiede attenzione (per esempio, quando termina un'operazione di lettura o scrittura).
     - *Trasferimento dei dati*: responsabile del trasferimento dei dati tra la memoria centrale (RAM) e la memoria di massa (come hard disk o SSD).

5.1.1 Partizioni e volumi

Un'unità hardware di memoria di massa può essere suddivisa in più **partizioni**. Il file system tratta ciascuna partizione come se fosse un'unità indipendente.

Motivi per creare partizioni:

1. il disco è più grande della dimensione massima supportata dal file system
2. una parte del disco può essere usata per scopi particolari (*es area di swap*)

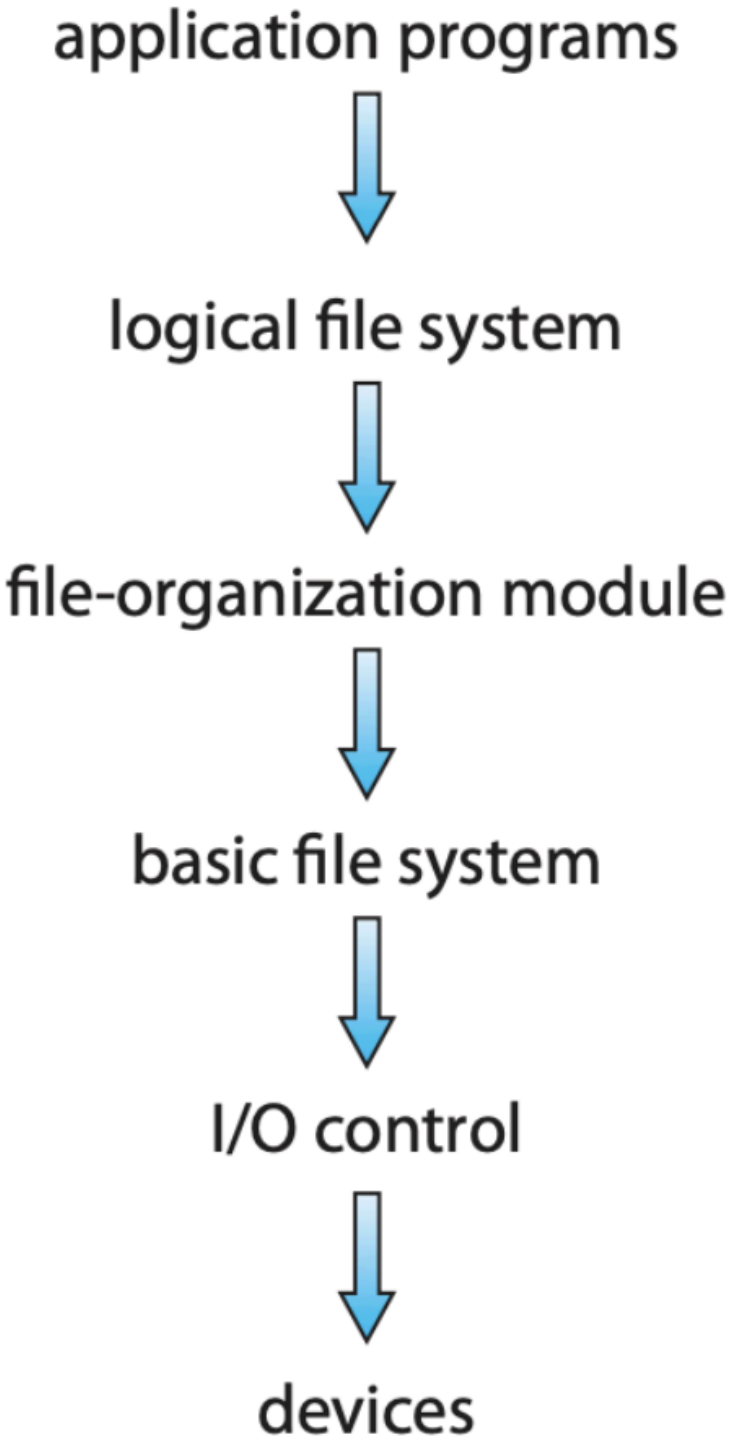

Figura 5.1: Struttura del file system

3. maggiore comodità di amministrazione (*tenere separati i file di sistema da quelli degli utenti*)

Dal punto di vista dell'utente, la memoria di massa si presenta divisa in unità logiche dette **volumi** (*es. C:, E:, D:*). Fisicamente, un volume è formato da un insieme di **blocchi** di dimensione fissa, che possono essere *letti/scritti* con una singola operazione di I/O del dispositivo.

Il file system organizza questi blocchi in strutture dati che consentono di realizzare i file e le directory.

Tipicamente, un *volume corrisponde ad una partizione*, ma è possibile avere volumi formati da più partizioni, anche su dischi diversi:

- **RAID** (*Redundant Array of Indipendent Disks*): tecnica che memorizza le informazioni di un unico volume su più partizioni in maniera ridondante.

  Utilizzata per migliorare le prestazioni, usando più dischi in parallelo, e per l'aumento dell'affidabilità (possibilità di ricostruire le informazioni memorizzate nel volume in caso di guasto fisico di uno dei dischi)

- **Esempio**: con la tecnica **RAID 5** è possibile usare **n** dischi insieme per realizzare un volume che contiene una capacità uguale a **n-1** dischi, usando una rappresentazione ridondante, con la proprietà che il danneggiamento di uno qualsiasi degli n dischi non comporta una perdita di informazioni.

Dunque, tipicamente un disco può contenere diverse partizioni ognuna delle quali può avere un file system diverso.

Una partizione senza file system è detta **raw partition**, usata per esempio per lo **swap** nei sistemi Unix.

Un'altra partizione priva di *file system*, ma con un formato proprio, è quella di **boot**:

- poichè all'avvio i driver del file system non sono ancora stati caricati
- consiste in un insieme di blocchi che vengono caricati in memoria centrale e contengono *istruzioni per l'avvio del SO*

Una volta scelto il SO la partizione radice viene **montata** (mount). Essa contiene il *kernel* del SO ed eventuali file di sistema.

Altre partizioni possono essere montate in modo automatico o su richiesta.

Il SO inserisce informazioni della partizione montata nella ***tabella di montaggio***.

5.2 Realizzazione del file system

Un file system richiede la definizione di diverse strutture dati, alcune in memoria di massa alte in memoria centrale.

Strutture dai in memoria di massa:

- **Boot control block**: contiene informazioni necessarie per avviare il SO contenuto nel disco
- **Volume control block**: contiene dettagli riguardanti il volume:
  1. Numero di blocchi fisici e dimensione
  2. Contatore dei blocchi liberi e puntatori ai blocchi

3. Contatore dei FCB presenti nel volume e puntatori ad essi

- **Struttura della directory**: organizza i file
- **File control blocks**: memorizza gli attributi dei file

Organizzazione delle directory

I file contentui in un volume sono raggruppati e organizzati usando la struttura delle directory.

In tutti i sistemi ad ogni volume è associata una directory radice (**root directory**), a seconda del SO ci possono essere ulteriori livelli di organizzazione:

- **struttua a un livello**: tutti i file sono contenuti nella directory radice
- **struttura a due livelli**: è previsto un solo livello di directory al di sotto della directory radice, per esempio una directory per ciasun utente
- **struttura gerarchica/ad albero**: è possibile avere più livelli di sotto-directory e ciascuna può contenere file e/o altre directory

5.2.1 Metodi di accesso ai file

Un **file** ha un nome che lo identifica univocamente nella directory a cui appartiene. L'*informazione fondamentale associata* a ciascun file è l'**insieme dei *blocchi*** che contengono i dati che lo compongono. Tipicamente il file system associa altre *informazioni accessorie* dette **attributi**.

Il File System può offrire diversi metodi d'accesso per leggere o scrivere i dati contenuti in un file:

1. **accesso sequenziale**: i dati sono esaminati in *sequenza*, semplice da implementare in qualcunque file system.
2. **accesso diretto**: è possibile accedere ai dati in qualsiasi ordine, specificano la **posizione** del dato da leggere o scrivere (es. RAM).
3. **accesso indicizzato**: si accede ai blocchi di dati specificando una ***chiave***, che è un sottoinsieme del blocco, definita al momento della crezione del file.

Nei **sistemi multi-utent**e, tra le informazioni mantenute dal file system ci sono quelle relative al ***controllo di accesso*** sui file, che specificano eventuali restrizioni sulle operazioni che ciascun utente è autorizzato a compiere sul file.

Ci sono due modi fondamentali con cui sono gestite queste informazioni:

1. attraverso insiemi di **permessi**, specificando la lista di utenti che possono accedere ma è poco flessibile.

   Per ciascun file è definito un numero fisso di classi di utenti (es. proprietario del file, membri dello stesso gruppi), a ciascuna delle classi corrisponde un bit per ogni operazione che indica se l'operazione è consentita.

   La rappresentazione è **compatta ed efficiente**, ma **poco flessibile**.

2. attraverso **Access Control Block** (*ACL*): definisce i gruppi di utenti che possono accedere.

   Ogni file ha una struttura dati dinamica che indica una lista di gruppi e per ciascun gruppo le operazioni consentite o delle operazioni vietate. Generalmente i file e le sotto-directory

ereditano la *ACL* della directory che li contiene, ma possono specificare delle variazioni. **Massima flessibilità**, ma **maggiore complessità implementativa** e maggiore *onere computazionale*.

5.3 Realizzazione del File System

Una specifica realizzazione del file system risponde alle seguenti domande:

1. Quali strutture dati sono usate per associare ad un file l'insieme di blocchi di dati che gli sono assegnati ? **FCB**.

   File Control Block è una struttura che contiene le informazioni necessarie al SO per gestire i file. Alcune di queste strutture dati sono caricate in memoria centrale quando un processo accede al file.

   - L'invocazione di una chiamata di sistema per l'apertura/chiusura di un file comporta l'uso dei FCB.
   - Il SO controlla la tabella dei file aperti per vedere se il file è già stato aperto.

     In caso negativo aggiunge il *FCB* alla tabella del SO.

     In ogni caso:

     (a) Aggiunge alla tabella per processo chiamante un nuovo elemento che punta alla tabella del SO

     (b) Increment il contatore della aperture del file

     Alla chiusura del file il SO decrementa il contatore e rimuove l'elemento di quel file della tabella del processo chiamato. Se il contatore di apertura vale 0 il FCB viene rimosso dalla tabella del SO (*per questo quando chiudiamo un'applicazione con un file aperto se il contatore non è 0 dice che è utilizzato su un'applicazione*).

2. Quali strutture dati sono usate per realizzare le directory ? **Struttura lineare** o **alberi / tabelle hash**.

3. Quali strutture dati sono usate per gestire lo spazio libero/occupato all'interno del volume ? **Lista dello spazio libero**, per ragioni di efficienza la struttura dati non è necessariamente una lista concatenata.

   Creazione di un file:

   (a) Si cerca nella lista il numero dei blocchi necessari

   (b) Si allocano al file (FCB)

   (c) Si rimuovono dall lista

   Eliminazione di un file:

   (a) I blocchi associati al file vengono deallocati

   (b) Si aggiungono alla lista

5.3.1 Metodi di allocazione

Un metodo di allocazione indica come i blocchi del disco sono assegnati ai file, in modo da garantire un utilizzo ed un acesso rapido.

Esistono tre metodi principali:

1. **Allocazione contigua**: alloca ciascun file in un intervallo di blocchi contigui del disco.

   Il *FCB* contiene il blocco iniziale e il numero di blocchi.

2. **Allocazione concatenata**: ogni file è composto da una lista concatenata di blocchi.

   Ogni blocco contiene il puntatore al successivo, il *FCB* contiene il puntatore al primo e all'ultimo blocco della lista.

   Però è *estremamente inefficiente*, in quanto bisogna scorrere la lista.

   L'allocazione **concatenata con *FAT*** è una sua variante:

   - I puntatori al blocco successivo non sono all'interno dei blocchi stessi ma in una tabella separata detta *File Allocation Table* (FAT)
   - *Vantaggio*: il SO mantiene una cache della FAT in memoria centrale rendendo più veloce lo scorrimento della lista

3. **Allocazione indicizzata**: ogni file posside un ***indice*** dei blocchi memorizzato in un blocco dedicato.

   Questo *blocco dedicato* contiene un array di puntatori ai blocchi del file. Il *FCB* memorizza per ogni file un puntatore al suo blocco indice.

   Vantaggio: rende veloce l'accesso diretto, il *problema* è che un singolo blocco indice può contenere un numero limitato di puntatori ai blocchi.

   Si può utilizzare uno **schema combinato**:

   - il FCB contiene 15 elementi per indicizzare i file
   - i primi 12 sono puntatori diretti a dei blocchi
   - il 13° punta ad un blocco contenente un indice
   - il 14° punta ad un indice a due livelli
   - il 15° punta ad un indice a tre livelli

   Il *vantaggio* è l'accesso diretto molto veloce per file piccoli, degrada però logaritmicamente al crescere della dimensione del file.

Ripristino: verifica della coerenza

In caso di arresto improvviso della macchina, le strutture dati del file system potrebbero essere lasciate in uno stato inconsistente.

Il **verificatore della coerenza** - un programma di sistema - confronta i dati delle directory con i blocchi dati dei dischi, tentando di correggere ogni incoerenza ripristinando le strutture dati danneggiate.

Nei file system "annotati" (***journaling file system***) le strutture dati sono modificare in maniera *ridondante* e scrivendo in *journal* quali modifiche il sistema effettua, prima di effettuarla, per garantire la possibilità di effettuare correttamente e rapidamente il ripristino.