RIASSUNTO SO

So → intermediario tra l'utente e l'hardware, rende il sistema semplice efficiente e sicuro, è *Event Driven.*

Kernel → Nucleo del sistema, svolge funzioni simili per ogni sistema.

- Gestione programmi in esecuzione
- Gestione memoria principale
- Gestione memoria secondaria

Single/MultiCore → Se c'era bisogno di più potenza si affiancavano più processori sulla stessa memoria, sistema multiprocessore.

- **Single core**: un solo programma per volta in esecuzione.
- **Multi core**: Adatti per gestire task multi-Thread (vedi sotto)

Sistemi Multi Processore → Più processori multi core possono essere combinati per fare sistemi ancora più grandi.

Thread → Due o più processi che lavorano sugli stessi dati. Un processo MultiThread è composto da più Thread di computazione detti peer-thread (context switch più efficiente perché condividono lo spazio di indirizzamento).

Multi Thread → Ogni singolo core (di un processore multi core) può eseguire in parallelo istruzioni appartenenti a più peer-thread distinti.

EVENTI

Interrupt → Eventi di natura Hardware che richiedono intervento SO

Eccezioni → Eventi di natura software causati da programmi in esecuzione. Trap e System call

Per la gestione di un evento viene salvato lo stato della computazione nel punto in cui si è interrotto il programma, viene gestito l'evento e viene fatta ripartire la computazione

Multitasking e Time sharing

Multitasking: assegnare la cpu ad altri processi mentre uno sta gestendo operazioni di I/O.

Time Sharing: dare un quanto di tempo ai processi.

Quando scade il quanto di tempo, il SO esegue delle istruzioni privilegiate (**bit di modalità: 1**).

Le **System Call:** costituiscono la vera e propria interfaccia tra processi utente e SO.

Protezione della memoria

Vengono utilizzati due registri **base** e **limite** il so carica gli indirizzi di inizio e fine dell'area assegnata al programma, ogni istruzione non può eccedere questo range

GESTIONE PROCESSI

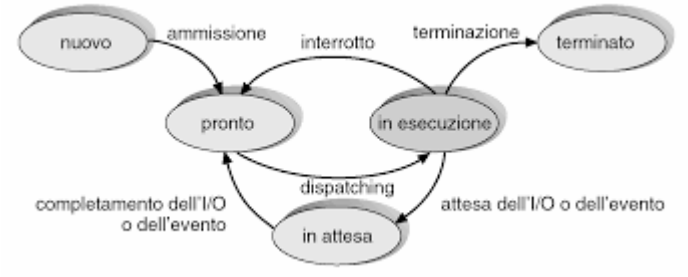
Le attività che deve il SO deve gestire per permettere il corretto ciclo di un processo sono sono Creazione, Sospensione, Sincronizzazione, Comunicazione e Cancellazione.

La struttura dati fondamentale di processo è il **PCB** (Process Control Block).

Un processo ha struttura bene definita in RAM composta da:

- **Codice da eseguire**
- **Dati**
- **Stack**
- **Heap**

Diagramma di transizione stati di un processo



ALGORITMI DI SCHEDULING

Context Switch → Significa il passaggio dell' esecuzione da un processo utente ad un altro.

I vari processi in ogni stato si trovano in varie code come la *Coda di ready, coda dei dispositivi (I/O), coda di waiting.*

CPU Scheduler → Decide quale processo mandare in esecuzione

Fork() → System Call per creazione di processi, restituisce 0 al figlio e il pid nuovo al padre.

Meccanismi di IPC (inter process comunications) → Meccanismi messi in atto da SO per permettere uno scambio di messaggi e sincronizzazioni tra i vari processi

- → **Semafori**
- → **Memoria condivisa**
- → **Code di messaggi**

SCHEDULING DELLA CPU

Con prelazione e senza prelazione ovvero il diritto di un processo di sottrarre la CPU ad un altro processo.

Dispatcher → Modulo del SO che si occupa effettivamente di fare il context switch.

DeadLock → Quando un processo aspetta una risorsa da un altro processo, che a sua voltaaspetta una risorsa in uso dal processo bloccato. Sono tutti e due fermi e non possono proseguire (MOLTO FREQUENTE CON I SEMAFORI !)

Ci sono 3 politiche di gestione del deadLock

1. Prevenire o evitare i deadLock
2. Lasciare che il DeadLock si verifichi e fornire strumenti per la risoluzione
3. Lasciare all'utente la prevenzione/gestione del deadLock

Starvation → Quando un processo è in coda di ready, ma non verrà mai selezionato per poter usare la CPU poiché verrà scavalcato da altri processi con una priorità maggiore.

Meccanismo di Aging → Permette di eliminare la starvation, ad esempio aumentando la priorità di un processo mano a mano che passa del tempo in coda di ready, fino a quando non avrà la priorità più alta e sarà per forza eseguito

Algoritmi:

FCFS: First Come First Served il primo processo che arriva gli viene data la CPU.
Non è preemptive, una volta che un processo inizia ad eseguire non la smette fino a quando non ha terminato. Meno ottimale di tutti

SJF: Shortest Job First è il migliore, viene mandato in running il processo col minor tempo di esecuzione (esiste sia preemptive che non). Non è implementabile

PRIORITA': Può essere sia preemptive che non, ogni processo in coda di ready ha una priorità e viene mandato in run quello che nella coda ha priorià maggiore.

RR: Round Robin, ogni processo ha a disposizione un quanto di tempo, una volta terminato viene generato un interrupt che lo toglie dallo stato di running rimettendolo in coda di ready, che è vista come un buffer circolare.

CODE MULTIPLE: I processi sono facilmente distinguibili in classi diverse.

- *Foreground* → Interattivi
- *BackGround* → Non interagiscono con utente
- *Batch* → La loro esecuzione può essere differita nel tempo

Dividiamo in processi tra più code e gestiamo ogni coda con l'algoritmo di scheduling che preferiamo

CODE MULTILIVELLO CON RETROAZIONE: L'assegnamento di un processo ad una coda non è più fisso, i processi si possono spostare da una coda ad un'altra, per adattarsi meglio in base alle loro priorità.
Possiamo poi gestire ogni coda con l'algoritmo di scheduling più adatto.
Esempio di 3 code QUANTO 8, QUANTO 16 e FCFS.

SOLARIS: Usa uno scheduling a priorità con code multiple a retro azione in cui i processi sono suddivisi in 4 classi.

- *Real Time* (priorità maggiore) PRIORITA' FISSA SEMPRE MAGGIORE
- *Sistema* PRIORITA' FISSA SEMPRE MAGGIORE
- *Interattiva* 50-59
- *Time Sharing* (priorità minore) 0-49

A ciascuna coda viene assegnata una priorità che deciderà quale sarà il prossimo processo ad essere eseguito
60 livelli di priorità (come avere 60 code)
Maggiore e la priorità minore sarà il quanto di tempo assegnatogli.
Se il processo esaurisce tutto il suo quanto di tempo gli viene abbassata la priorità per la sua successiva esecuzione ma gli viene alzato il quanto di tempo.
Se il processo termina prima o va ad eseguire I/O, gli viene alzata la priorità ma abbassato il quanto di tempo.

WINDOWS: simile a solaris, 32 livelli di priorità
Processi utente → fino a 15 livelli
Priorità da 16 a 31 → processi Real Time e di Sistema
Priorità 0 è riservata
Quando un processo nasce ha priorità 1.

- Se un processo usa CPU e va in wait → viene incrementata la priorità
- Se un processo usa CPU e finisce il tempo → viene abbassata la priorità

Il processo ForeGround è quello associato alla finestra attualmente attiva (se un processo passa in foreground il suo quanto viene moltiplicato per 3).

LINUX: Completely Fair Scheduling (CFS) (vedere su quaderno)
Viene distribuito equamente il tempo di CPU tra tutti i processi ready to run.
Ad ogni context switch viene ricalcolato il tempo da dare ad ogni processo.
La CPU viene data al processo che fino ad ora ha usato la CPU per meno tempo.
I processi ready to run non sono più inseriti in una coda di scheduling ma diventano i nodi di un albero di ricerca bilanciato **red-black tree.**

SINCRONIZZAZIONE DEI PROCESSI

Quando due o più processi devono accedere e modificare dati e risorse condivisi, i processi devono sincronizzarsi in modo da eseguire completamente le operazioni sui dati condivisi prima che un altro processo abbia il diritto di accedere agli stessi dati → Operazioni Atomiche

SEZIONI CRITICHE

Porzione di codice dove vengono manipolate variabili o dati condivisi. L'esecuzione di queste parti di codice deve essere fatta in maniera mutualmente esclusiva.
Problema della sezione critica: Consente di stabilire un protocollo di comportamento usato dai processi che devono usare le variabili condivise.
Processo deve chiedere permesso di eseguire una sez. critica: **entry section.**
Processo deve segnalare un eventuale uscita da una sez. critica: **exit section.**
3 proprietà che soddisfano il problema della sezione critica:

1. **MUTUA ESCLUSIONE:** se un processo è nella propria sezione critica nessun altro processo potrà eseguire una sezione critica
2. **PROGRESSO:** Se un processo lascia la sezione critica deve permettere ad un altro processo di entrare nella propria sezione critica
3. **ATTESA LIMITATA:** Qualsiasi processo che richiede di poter eseguire la propria sezione critica deve poterlo fare in un tempo limitato.

SINCRONIZZAZIONE VIA HARDWARE

TestAndSet(var1): Testa e modifica il valore di una cella di memoria.

```
Boolean TestAndset(boolean *lockvariable){     //riceve in input una variabile
        boolean tempVariable = *lockVariable;  //ne salva il valore su una variabile temporanea
        *lockVariable = true;                  //imposta il valore di lockvariable a true
        return tempvariable;                   //restituisce il valore primordiale
}
```

Per poter eseguire la propria sezione critica un processo, deve prima eseguire la entry section legata alla TestAndSet:

```
while(TestAndSet(&lock)){
        // sez critica, qua lock vale false
        Lock = true;
        // sezione non critica
}
```

Un processo entra nella propria sezione critica se e solo se il valore della variabile lock è a false. Dopo la prima esecuzione del while, quel valore varrà true e nessun altro potrà accederci e il processo continua a ciclare nel while.
Quando un processo avrà finito, nella sua exit section metterà lock a false, e il processo che stava ciclando nel while potrà finalmente progredire.
La **TestAndSet** non garantisce una corretta soluzione al problema della sezione critica (vedere spiegazione su quaderno).
Busy waiting: Quando un processo che attende per la propria sezione critica spreca tutto il suo quanto di tempo a testare il valore della variabile di Lock.

SEMAFORI
Consentono di avere sincronizzazione senza busy waiting.
Un semaforo è una variabile intera che assumiamo inizializzata a 1, su cui si può operare solo tramite due operazioni atomiche.
Prima di entrare in sezione critica testiamo il valore del semaforo e lo decrementiamo attraverso una **wait(S)** e una volta finita la sezione critica lo re incrementiamo attraverso la **signal(S)**.
Ogni semaforo è implementato usando due campi: valore e la waiting list.

Wait

```
Wait(Semaforo *S){
        S→valore--;
        if (s->valore<0){  //non gli do la risorsa ma lo aggiungo alla waiting list
                aggiungi questo processo a S→waitingList;
                sleep();
        }
}
```

Signal

```
Signal(Semaforo *S){
        S→valore++;
        if (S→valore<=0){ //c'è qualche processo in wait
                togli un processo P da S→waiting_list
                wakeup()
        }
}
```

Wait e Signal sono a loro volta sezioni critiche perché usano variabili condivise. Per eseguirle però al fine di evitare problemi, le eseguiremo disabilitando gli interrupt, in modo che vengono eseguite come operazioni atomiche.

Esempi di sincronizzazione
Produttori consumatori
Setto i semafori
Full = 0 //all'inizio non c'è nessuna cella piena
Empty = SIZE //le celle vuote all' inizio sono tutte vuote
Mutex = 1 //sez critica
Int in = 0 //punta alla prima entry libera del buffer
Int out = 0 //punta alla prima entry occupata
Item buffer[SIZE]; //buffer gestito come array circolare di dimensione SIZE

Codice produttore:

```
while(true){
    ……
    //produci un item in nextp
    ……
    wait(empty);
    wait(mutex);
    buffer[in] = nextp;
    in = (in + 1) mod SIZE;
    signal(mutex);
    sigal(full);
}
```

Codice consumatore:

```
    while(true){
    wait(full)
    wait(mutex)
    nextc = buffer[out];
    out = out +1 mod SIZE;
    signal(mutex);
    signal(empty);
    ……
    //consuma item in nextc
    ……
}
```

Problema lettori-scrittori

```
Semaphore mutex= 1, scrivi=1;
int numLettori = 0;
```

Codice scrittore:

```
wait(scrivi)
……esegue scrittura…..
Signal(scrivi)
```

Codice lettore:

```
wait(mutex);
numLettori++;
if (numLettori == 1)
        wait(scrivi);
signal(mutex);
….eseguo lettura….
numLettori--;
wait(mutex);
if (numLettori == 0)
        signal(scrivi);
signal(mutex);
```

problema dei 5 filosofi

```
Semaphore bacchetta[5]

do{
  wait(bacchetta[i]);
  wait(bacchetta[i+1 mod 5]);
  ……
  Mangia
  ……
  Signal(bacchetta[i]);
  signal(bacchetta[i+1 mod 5]);
  ……
  Pensa
  ……
}while(true);
```

GESTIONE DELLA MEMORIA

Area di Swap

Viene spostata sull' HD l'immagine di un processo per poi poter essere ricaricato in RAM successivamente.

Vengono copiati

- Dati
- Stack con eventuale Heap
- Codice

Binding

Associare per ogni variabile dichiarata nel programma un'indirizzo di una cella di memoria in RAM in cui verranno scritti i suoi valori.

Il binding può essere fatto in 3 fasi distinte:

1. **Fase di Compilazione → codice statico** (indirizzi statici)
2. **Fase di Caricamento → codice staticamente rilocabile** (indirizzi relativi rispetto ad un ipotetico zero virtuale che è l'inizio del programma)
3. **Fase di Esecuzione → codice dinamicamente rilocabile** (traduzione degli indirizzi viene fatta nell'istante in cui un istruzione viene eseguita, si fa riferimento ad un **registro di rilocazione** che contiene l'indirizzo di partenza dell'area di RAM in cui è caricato il programma. La **Memory Management Unit** si occuperà di trasformare gli indirizzi da relativi ad assoluti, usando il registro di rilocazione).

Indirizzi logici o virtuali

Indirizzi generati dalla cpu, ovvero gli indirizzi che sono usati dalle istruzioni del programma in un determinato momento in esecuzione.

Indirizzi fisici

Vengono calcolati sommando al contenuto del registro di rilocazione il valore di un indirizzo logico usato dall'istruzione in esecuzione. L'indirizzo fisico potrà essere caricato nel MAR per inidirizzare una cella di memoria.

Librerie: Collezione di subroutine di uso comune messe a disposizione dai programmatori per sviluppare un software. Possono essere

statiche: caricate in fase di compilazione , ogni programma che le usa dovrà incorporarne una copia

dinamiche: Caricate in RAM solo nel momento in cui il programma che le usa chiama una delle subroutine della libreria di cui ha bisogno, usate per più programmi.

<u>Vantaggi dinamiche</u>:

Permettono di risparmiare spazio in RAM, una sola copia per tutti i programmi

Vengono caricate solo se viene richiamata una subroutine della libreria da un programma

Il loro aggiornamento non richiede la ricompilazione del programma.

TECNICHE DI GESTIONE DELLA MEMORIA PRIMARIA

Swapping: Salvare in memoria secondaria l'immagine di un processo in esecuzione e ricaricarla

Allocazione contigua: RAM divisa in 2 parti (processi e SO, divise da un **registro limite**) quando un processo entra in RAM si alloca in partizioni fisse, il n. di partizioni definisce il grado di multiprogrammazione. Grande problema che genera un sacco di frammentazione sia interna che esterna.

Allocazione a partizioni multiple variabili: Un processo riceve una quantità di memoria pari esattamente alla sua dimensione.
Per assegnare la partizione si adotta una tra le 3 principali tecniche:

- **First fit** → Si da la prima partizione che lo contiene
- **Worst fit** → Si da la partizione più grande
- **Best fit** → Si da partizione appena maggiore a contenere il processo

Per questa tecnica ci sarà bisogno ad un certo punto di una **compattazione** della ram.

Paginazione : La RAM viene divisa in frame e il l'area di memoria allocata da un processo è anch'essa divisa in tanti pezzi di dimensioni uguali ai frame.
Metodo Base:
se un processo occupa X pagine il SO cerca X frame liberi. C'è una tabella delle pagine che riporta per ogni processo i frame in cui sono memorizzate le sue pagine.
Viene mantenuto un elenco dei frame liberi.
Gli indirizzi logici sono divisi in 2 (PAGINA,OFFSET).
La **traduzione** degli indirizzi viene fatta cercando nella PT il frame che contiene la pagina, e gli viene attaccato l'offset. Ind. Fisico (FRAME,OFFSET).

Vantaggi: Si sfrutta al meglio la RAM,eliminando la frammentazione esterna e riducendo al minimo quella interna.
Viene implementata automaticamente una forma di protezione dello spazio di indirizzamento (senza reg. limite).
Condivisione del codice e delle librerie dinamiche.

Svantaggi: Ad ogni context switch il SO deve attivare la tabella delle pagine del processo entrante e questo provoca overhead.
Il meccanismo di traduzione degli indirizzi è complesso e può portare a dei ritardi.
E' necessario un supporto hardware (TLB)

TLB: Struttura conservata all' interno dei registri della CPU dove vengono contenute alcune entry della tabella delle pagine. E' una memoria associativa (chiave-valore → numero pagina-frame) costruita a semiconduttore.
PTBR: Page Table Base Register , contiene l'indirizzo di partenza della PT, aggiornato ad ogni context switch.
La paginazione rende molto più semplice la condivisione di pagine tra più processi.

Paginazione a più livelli: Quando un'intera page table pesa più di un frame, essa viene caricata su più frame, e viene creata una page table esterna per tenere traccia dei frame occupati dalla PT
L'indirizzo logico viene ora diviso in 3 parti

1. Bit per indirizzare una entry della pt esterna, dove c'è il numero di un frame F1
2. Bit da usare come offset nel frame appena trovato(F1) e trovare una entry della PT interna, dove al suo interno c'è scritto il numero del frame F2 (che conterrà il dato
3. Bit da usare come offset nel frame F2

Page Table Invertita: L'indice di ogni entry della IPT corrisponde al numero di un frame in memoria principale.
Con la IPT si risparmia spazio ma il tempo di traduzione aumenta enormemente.
Ogni entry è composta da (PID,n. Pag)

Supporto alla paginazione della microarchitettura IA-32: Se un eventuale Directory delle pagine contenesse 1024 record (2^10 pagine), quando viene fatto un context switch viene cambiato il valore nel registro CR3, ovvero il registro usato dal SO che contiene l'indirizzo di partenza della tabella delle pagine, cambiando il suo valore viene caricata in RAM un'altra tabella delle pagine. (questo nello schema di paginazione ad un livello 1)

Nello schema a più livelli, la Directory delle pagine corrisponde alla PT esterna.

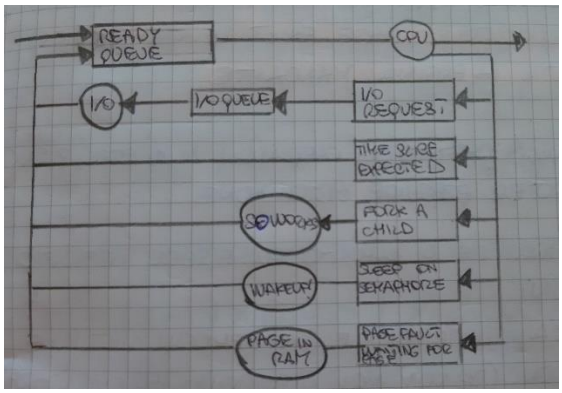
MEMORIA VIRTUALE
Insieme di tecniche che permette l'esecuzione di processi in cui codice e dati non caricati completamente in memoria primaria.
Viene caricato un pezzo di programma solo quando deve essere eseguito.
Possiamo eseguire processi che hanno uno spazio di indirizzamento logico maggiore di quello fisico.
Ottengo un aumento del grado di multiprogrammazione e inoltre i programmi partono più velocemente.
Svantaggi: Aumento del traffico tra RAM e HD.
L'esecuzione di un programma può richiedere più tempo a causa dei page fault.
C'è il rischio che il sistema vada in *Trashing.*

Paginazione su richiesta: Porto una pagina in MP solo nel momento in cui viene indirizzata una locazione che appartiene a quella pagina
Page fault: quando ad un processo serve un dato in una pagina che al momento non è caricata in RAM. Il processo che genera il page fault viene messo in waiting in uno stato che si chiama "Waiting for page".
Un modulo del SO, il **pager** provvederà a caricare la pagina mancante in RAM.
Bit di validità: Ci indica se la è o no presente in RAM. 1 se è in ram
Pure demand paging: Viene generato page fault alla prima istruzione caricata in ram.

EAT: tempo effettivo di accesso alla memoria primaria quando viene richiesto un dato
TLB → T_TLB + T_RAM
RAM → T_RAM * 2 + T_TLB
P.FAULT (dirty bit a 0) → T_PFAULT + 2*T_RAM + T_TLB
P.FAULT (dirty bit a 1) → T_PFAULT*2 + T_RAM*2 + T_TLB

Area di swap: Porzione dell' HD usata come estensione della memoria principale, è gestita con meccanismi più semplici e veloci del FS, viene usato soprattutto per ospitare pagine dei processi in RAM ma che in quel momento non vengono utilizzate.
Pagina vittima: E' la pagina che viene tolta dalla RAM dopo un page fault, se la pagina vittima contiene dati (non è codice), e questi ultimi sono stati modificati deve poi essere salvata nell' HD
Dirty bit: Associato ad ogni pagine ci indica se quest'ultima è da salvare o no in fase prelevamento dalla RAM.

Diagramma di accodamento dei processi: Con l'aggiunta di un eventuale stato di attesa ovvero, l'attesa di una pagina mancante, possiamo quindi completare il diagramma di accodamento dei processi.



ALGORITMI DI SOSTITUZIONE DELLE PAGINE

Lo scopo degli algoritmi di sostituzione delle pagine è quello di minimizzare il numero di page fault. Maggiore è il numero di frame, minore è il numero di page fault.

SOSTITUZIONE SECONDO ORDINE DI ARRIVO FIFO: Come pagina vittima viene presa quella che è da più tempo in RAM. Non è un buon algoritmo, soffre dell'**anomalia di belady** ovvero che usando più frame il numero di page fault può aumentare.
SOSTITUZIONE OTTIMALE DELLE PAGINE (non implementabile): produce il minimo numero di page p.f. e non soffre dell'anomalia di belady.
La pagina vittima è quella che sarà usata più in la nel tempo.
SOSTITUZIONE LRU (Last Recently Used): Come pagina vittima viene presa quella usata più tempo addietro.
Buon algoritmo ma difficile da implementare ,richiede supporto hardware molto complesso.
Una possibile implementazione avviene attraverso il **reference bit:** bit associato ad ogni entry della pt che indica se la pagina è stata usata (1), 0 altrimenti.

SECONDA CHANCE: In caso di page fault si esamina la pagina entrata in ram da più tempo.

- Se ha **reference bit = 0** → La seleziono come pagina vittima (siccome non è usata da molto)
- Se ha **reference bit = 1** → Il SO le da una seconda chance, azzerando il suo bit e esaminando le pagine successive. Le pagine sono in un buffer circolare, se tutte le successive avranno il reference bit a 1, la pagina precedentemente settata a zero questa verrà tolta dalla RAM

SECONDA CHANCE MIGLIORATO: Raggruppiamo le pagine in 4 classi
(reference,dirty)

1. **(0,0)** → E' la migliore per essere tolta, siccome non deve essere salvata
2. **(1,0)** → meno buona poiché ha reference a 1, ma comunque non va salvata
3. **(0,1)** → va salvata
4. **(1,1)** → peggiore candidata alla sostituzione.

Pool frame liberi: Le pagine vittime vengono prima messe in un pool di frame liberi, prima di essere salvate nell'area di swap (in modo che se servono da li a breve sono più veloci da reperire) Il SO poco per volta provvederà a prenderle dal pool e a salvarle nell'area di swap.
Allocazione uniforme: a tutti viene dato lo stesso n. di frame
Allocazione proporzionale: tiene conto delle dimensioni dei processi
Allocazione in base alla priorità: più un processo ha priorità alta, più frame avrà
Allocazione globale: Quando si verifica un page fault, la pagina vittima è scelta tra tutte le pagine
Allocazione locale: Pagina vittima scelta tra le pagine del processo che ha generato page fault.

Trashing: Se il grado di multiprogrammazione è troppo alto in base alla quantità di RAM, c'è il rischio che i processi si sottraggono i frame a vicenda, fino ad un punto in cui tutti i processi saranno in attesa di una pagina e nessuno potrà continuare ad eseguire.
La coda di ready si svuota poiché tutti i processi sono in waiting, e quindi il SO continua a mandare processi in coda di ready, peggiorando sempre di più la situazione
Soluzioni: aggiungere RAM, diminuire grado multiprogrammazione.
IL modo con cui vengono allocati e acceduti i dati influisce tantissimo sulle prestazioni del sistema.

WINDOWS
Windows adotta la *demand page with clastering,* quando viene caricata una pagina, si caricano alcune pagine adiacenti che si suppone vengano usate.
Alla creazione di un processo gli vengono assegnati due numeri

- **Insieme di lavoro minimo:** min. num di pagine che il so garantisce di allocare per quel processo
- **Insieme di lavoro massimo:** Max numero che il so allocherà

Se viene generato page fault ma non è stato raggiunto il numero massimo, viene caricata la pagina senza toglierne nessuna dalla ram. Una volta raggiunto quel numero le pagine verranno tolte.

SOLARIS
Usa paginazione su richiesta, viene assegnato un frame libero ad una pagina in caso di page fault.
Viene usato un parametro Lostfree, associato all'elenco dei frame liberi.
Ogni ¼ si secondo il SO controlla che il numero di frame liberi sia inferiorie a lostfree, se non lo è viene attivato PageOut:

1. Scandisce la pagine in ram azzerando il reference bit
2. Riscansiona le pagine, se trova pagine col dirtybit a zero, vengono considerate riutilizzabili, se hanno dirty bit a 1 vengono salvate prima di essere effettivamente riutilizzate.

Se pageout non mantiene il numero di frame liberi sotto quel livello, si sta verificando il trashing.

MEMORIE DI MASSA

Il disco è diviso in Tracce concentriche, a loro volta divise in settori.
L'insieme della tracce (su piatti diversi) genera il cilindro che ruota sul suo asse.
Una serie di testine di muovono per selezionare diversi settori e operare in lettura o scrittura.

Ogni settore memorizza un **blocco di dati**.
SeekTime: tempo di posizionamento della testina, il SO può gestire gli spostamenti della testina in modo da minimizzare questo tempo.
RotationalLatency: tempo richiesto per far ruotare il disco sotto la testina.

SCHEDULING DEI DISCHI RIGIDI
Algoritmi che gestiscono l'accesso ai dati richiesti.
FIRST COME FIRST SERVED: Vengono servite le richieste nel modo in cui sono arrivate spostando il disco di conseguenza.
C-SCAN: La testina di muove da un estremo all'altro servendo via via le richieste ma solo in un senso. Quando arriva all'estremità si sposta velocemente alla posizione zero senza servire richieste e riprenderà ad eseguire progressivamente.

Formattazione del disco: Un disco per essere usato deve essere formattato **fisicamente**, ovvero vengono associati ad ogni settore il suo numero e viene scelta la dimensione dei blocchi.
Logica: Necessaria per creare e gestire il FS sull' HD, viene creata una lista dei blocchi liberi e una directory iniziale e vengono riservate le aree necessarie al SO e viene deciso il BOOT BLOCK, che contiene il codice necessario a far partire il pc.

Gestione Area di swap: Spazio riservato sull' hd dal SO durante la formattazione fisica.
Quest'area deve essere molto veloce ed efficiente, di norma si usa allocazione contigua

RAID Redundant Array of Inexpensive Disk

Modo di configurare la memoria secondaria usando un insieme di HD al fine di aumentarne le prestazioni e l'affidabilità.
Viene visto dal SO come un solo HD

RAID 0
I dati non sono duplicati. Il disco virtuale viene mappato sui settori dei vari dischi di cui è composto il sistema suddividendo i blocchi logici del disco virtuale in **strips** di k blocchi consecutivi. In questo modo possiamo operare con letture e scritture di più blocchi in parallelo aumentando molto le prestazioni. Alte prestazioni.

RAID 1
Per ogni disco dati esiste un corrispondente disco di mirroring. Lettura in parallelo, molto veloce e affidabile.

RAID 10/01
Usa il meccanismo dello striping e del mirroring in modo combinato.
Dividiamo in strips i dati e duplichiamo i dischi. E' il migliore sotto tutti i punti di vista, solo che è molto costoso

RAID 4 (striping con parità)
Lo striping con parità consiste di avere delle strips generate attraverso delle ex-or tra le varie stip del disco, che permettono di salire ad un eventuale perdita di dati.
I dati sono suddivisi in strip tra i vari dischi e letti in parallelo.
In un disco aggiuntivo vengono salvate le strip di parità che sono aggiornate ogni volta che si modificano dei dati.
Questo però genera molto lavoro al disco delle strip che si romperà prima degli altri.

RAID 5 (Striping con parità distribuita)
E'una variante del RAID 4 ma gli strip di parità sono distribuiti su tutti i dischi del sistema.
Questo livello è il miglior compromesso tra costi prestazioni e affidabilità,è il più usato.

RAID 6
E' come il 4 ma con due livelli di parità, distribuiti su tutti i dischi (come nel 5).

FILE SYSTEM
Fornisce i meccanismi per la memorizzazione e l'accesso ai dati e agli applicativi del sistema operativo e degli utenti.
Composto da due parti
Insieme di file → file ovvero unità logica di informazione memorizzata permanentemente su un supporto di memorizzazione secondario.
Struttura di directory → Permette l'organizzazione dei file
Un file può essere visto come un entità astratta definito solo dalle operazioni che si possono fare su di esso. (creazione/eliminazione/lettura/scrittura).

Metodi di accesso: i metodi di accesso possono essere

- **Sequenziali:** i dati vengono letti/modificati in modo sequenziale
- **Accesso diretto:** Si desidera leggere o modificare un dato posizionato in un punto ben preciso del file

Directory: consentono di tenere traccia dei file. Una directory permette di risalire a tutte le informazioni di un file in quella directory a partire dal nome. Sono quindi dei file che contengono informazioni relative ad altri file.
Un file directory ha una struttura, contiene un numero di entry, una per ogni file di quella directory.
Esempio UNIX
Viene inserito a fianco del nome di ogni file (di ogni entry) un puntatore ad una struttura interna in cui sono contenute le informazioni di quel file.

NTFS: Le directory hanno una struttura ad albero di ricerca bilanciato, in cui ogni foglia corrisponde ad un file.
Ciascuna fogli contiene il nome di un file e un puntatore (file reference) alla struttura interna che contiene tutte le info associate al file.

Pathname: percorso che si deve fare tra le directory del FS per raggiungere un determinato file.

Directory con struttura ad albero: ogni directory può contenere file normali e altre directory e così via ricorsivamente.
La directory principale da cui si diramano tutte le altre si chiama **root del file system.**
Ciascun utente ha a disposizione una porzione di file system che parte dalla sua **home directory**.
Pathname relativo : definito rispetto alla current directory, inizia con il nome di una directory
Pathname assoluto: definito rispetto alla root, inizia con /

Directory con struttura a grafo aciclico: non si formano mai dei cicli tra varie directory.
I diversi collegamenti a file o directory prendono il nome di link.
Directory con struttura a grafo generale: ci possono essere dei cicli tra directory.

ACCESSO RAPIDO AI FILE
Quando dobbiamo svolgere operazioni su un file, non usiamo il pathname ogni volta che dobbiamo accedere, sarebbe super inefficiente, troppi accessi alla memoria secondaria.
Il SO richiede quindi ai programmi di aprire i file con la system call **open** che copia le informazioni relative ad un file nella **open file table** , gli accessi non passeranno più attraverso la memoria secondaria ma passeranno attraverso questa tabella che è contenuta in RAM.
Per chiudere questo "strem" si usa la **close.**
Il SO si occupa di ricopiare le modifiche effettuate sul file in MS.
Protezione file: può avvenire mediante due strategie

1. **Lista d'accesso:** per ogni file viene specificato chi ha i diritti e cosa può fare
2. **Capability list:** associa ad ogni utente la lista dei file su cui ha dei diritti

ALLOCAZIONE DEI FILE

Può essere di 3 tipi

CONTIGUA

Ogni file è allocato in un insieme di blocchi contigui sull' hd.
Per recuperare il file basta memorizzare **numero di blocco ,quantità di blocchi contingui occupati.**
Vantaggi:

- Semplice da implementare
- Accesso veloce ai blocchi
- Bastano poche info per recuperare un dato

Svantaggi:

- Dobbiamo avere spazi liberi adiacenti molto grandi per memorizzare i file
- Dobbiamo implementare una strategia di scelta del buco libero in cui mettere i file
- Soggetto a frammentazione esterna
- Dopo un po' di tempo sarà necessaria una ricompattazione del disco (molto costosa)

CONCATENATA

Catena di blocchi contenenti i dati dei file. Ogni blocco conterrà negli ultimi byte un puntatore al successivo blocco. Mi basta sapere il numero del primo blocco per risalire al dato che mi serve
Vantaggi:

- Non sono necessari blocchi contigui
- Non si verifica frammentazione esterna
- Qualsiasi blocco libero è usabile

Svantaggi:

- Perdita di byte usati per il puntatore
- Accesso ad un dato molto inefficiente, poiché devo scorrere tutta la catena di puntatori
- È molto inaffidabile, basta perdere un blocco per perdere tutto il file (per risolvere il problema si usa la doppia concatenazione)

CONCATENATA VARIANTE FAT

FAT: File allocation tabl, area all'inizio del disco in cui l'indice di ogni entry corrisponde ad un blocco.
Registra lo stato di allocazione di tutti i blocchidell'hd.
Se un blocco I punta ad un blocco N allora, nella fat all' I-esima entry ci sarà il numero N
Vantaggi:

- È sicura, se si rovina un blocco non perdiamo tutta la catena
- La gestione dei blocchi liberi è automatica

Svantaggi

- Deve essere tenuta in RAM

INDICIZZATA

Teniamo traccia di tutti i blocchi in cui è contenuto un file scrivendo il loro numero in un altro blocco del disco, il **blocco indice.**

Per recuperare i dati del file, basta memorizzare negli attributi del file il numero del suo blocco indice.

Vantaggi:

- Non sono necessari blocchi contigui
- Accesso diretto ad un byte di un file è molto efficiente

Svantaggi:

- Per ogni file (anche di piccolissime dimensioni) deve essere presente il suo blocco indice

Se non basta un blocco per memorizzare il blocco indice:

Schema concatenato: l'ultima entry del blocco indice punta ad un altro blocco indice

Schema a più livelli: Il blocco indice contiene solo puntatori ad altri blocchi inidice.

NTFS (WINDOWS)

Ogni file è caratterizzato da un elemento, il quale ha dimensione fissa.

Tutti gli elementi sono contenuti nella **Master file Table** ovvero un file memorizzato nei primi blocchi dell hd gestito solo dal so

Ogni elemento contiene informazioni relative al file, e una serie di puntatori a blocchi di dati che contengono il file.

Se il file è piccolo viene memorizzato direttamente nell' elemento

I-NODE UNIX

Ad ogni file è associato un i-node che contiene gli attributi del file e l'elenco dei blocchi di dati del file. Gli i-node sono gestiti solo dal SO.

Ogni i node contiene lo spazio per memorizzare

- 10 puntatori diretti
- Un puntatore **single direct**
- Un puntatore **double indirect**
- Un puntatore **triple indirect**

I puntatori punteranno a blocchi inidice con le varie entry dei blocchi dati.

Per leggere quindi un file dobbiamo leggere il suo INDEX NODE.

Gestione spazio libero

Il so tiene traccia dello spazio libero mediante opportune strutture

In unix mediante un **superblocco**

In windows mediante la **master file table**

Vettori di bit: un bit per ogni blocco 1 libero, 0 occupato. (mac-os)

Lista concatenata: formare una lista dei blocchi liberi

Raggruppamento: raggruppare in un blocco con tanti puntatori ai vari blocchi liberi

L'efficienza di un FS dipende dal sistema di caching che il FS ha, dove può mantenere file aperti, o quelli usati più spesso.

I LINK UNIX

In unix un file è identificato univocamente dall'index node che contiene tutte le informazioni relative al file, i suoi attributi e in quali blocchi sono memorizzati i suoi dati.
Il numero dell'index node rappresenta univocamente un file.

Hard-Link: La stringa di caratteri scritta a fianco del numero di un i-node in una directory è ciò che viene chiamati link fisico o hard link al file. In unix un file regolare può avere da uno a più link fisici. In una directory possono esistere più entry diverse, il cui numero dell' i-node è lo stesso, ma con un altro nome.
Link-counter: E' un campo di tipo int dell' i-node. Quando un file viene creato, il link counter viene inizializzato a 1. Ogni volta che creiamo un nuovo link fisico, il link counter del file linkato viene incrementato di 1.

Comando: ln *existing_file_name new_file_name*
Aumenta il link counter di 1 crea una nuova entry nella directory specificata da *new_file_name* con il numero dell' i-node e il nuovo nome.

Comando: rm *nome_file*
Recupera l'i-node di nome_file
Rimuove la entry nome_file nella directory in cui è
Decrementa illink-counter dell'i-node associato di 1
Se il link-counter ora vale 0, viene cancellato l'i-node.

Directory: Alla creazione di una nuova directory (newdir), la nuova directory nasce vuota ma con due entry ovvero
. → indica la cartella corrente
.. → riferimento alla cartella padre
Il link counter di newdir viene quindi inizializzato a due, mentre il link-counter di parentDir viene incrementato di uno.
Non si possono fare link fisici tra cartelle.

Link-simbolici: Permettono i link tra cartelle, vengono creati con il seguente comando
Ln -s existing_directory new_directory
Questo tipo di link funziona come il collegamento di Windows.
I link simbolici vengono implementati allocando un nuovo i-node, che viene associato al link simbolico. Sono distinguibili dai link fisici, quindi non si rischia di avere dei cicli di cartelle nel FS.
Non incrementa il link-counter del file sorgente, alloca solo un puntatore al link simbolico.
Sono meno efficenti dei link fisici, siccome per poterli usare bisogna leggere un index node in più.
Sono necessari per i link tra directory e per link tra file che stano su volumi (partizioni) diversi dell' HD.