**Paginazione**

La paginazione è un metodo di gestione della memoria che permette che lo spazio degli indirizzi fisici di un processo non sia contiguo.

**Metodo di base**

Il metodo di base per implementare la paginazione consiste nel suddividere la *memoria fisica* in blocchi di dimensione costante, detti **frame**, e nel suddividere la *memoria logica* in blocchi di pari dimensione, detti **pagine**.

Lo spazio indirizzi logico di un processo può essere allocato in blocchi anche *non contigui* di memoria fisica. Quando si deve eseguire un processo, si caricano le sue pagine nei frame disponibili, prendendole dalla memoria ausiliaria, divisa in blocchi di dimensione fissa, uguale a quella dei frame della memoria.

Ogni indirizzo generato dalla CPU è diviso in :

* Numero di pagina (p) : il numero di pagina serve come indice per la tabella delle pagine che contiene l’indirizzo base di ogni pagina nella memoria fisica;
* Scostamento di pagina (offset) : l’offset viene combinato con l’indirizzo base per definire l’indirizzo della memoria fisica che viene inviato all’unità di memoria.

Con la paginazione si risolve il problema della frammentazione esterna perché qualsiasi frame libero si può assegnare a un processo che ne ha bisogno. Però si può avere la frammentazione interna perché visto che i frame si assegnano come unità può succedere che lo spazio di memoria richiesto da un processo non è un multiplo delle dimensioni delle pagine, così l’ultimo frame assegnato può non essere completamente pieno.

Il caso peggiore si ha con un processo che necessita di n pagine più un byte: si assegnano n + 1 frame, quindi si ha una frammentazione interna di quasi un intero frame.

Il caricamento di un processo avviene come segue : viene esaminata la dimensione dello stesso espressa in pagine, ed ognuna di esse necessita di un frame. Per un processo di n pagine si necessita di n frame. Si carica la prima pagina in uno dei frame assegnati e s’inserisce il numero del frame nella tabella delle pagine relativa al processo in questione; così via per i frame successivi.

Un aspetto importante della paginazione è la distinzione tra la memoria vista dall’utente e l’effettiva memoria fisica: il programma utente vede la memoria come un unico spazio contiguo contenente solo il programma, ma in realtà il programma utente è separato in una memoria fisica contenente anche altri programmi. La differenza tra la memoria vista dall’utente e la memoria fisica è colmata dall’architettura di traduzione degli indirizzi, che fa corrispondere gli indirizzi fisici agli indirizzi logici, ed è gestita dal SO e non sono visibili all’utente. La memoria di un programma è limitata solo ai frame assegnatogli dal SO. Il sistema operativo mantiene una copia della tabella delle pagine per ciascun processo, in modo tale da fornirla al dispatcher per fornire ai processi successivi frame e pagine libere.

**Paginazione degenere (thrashing)**

Supponiamo che in un determinato istante ogni processo ha un numero minimo di frame per essere eseguito ed inoltre il sistema utilizza una politica di allocazione globale. Quindi ogni processo che accusa un page fault sottrae la frame agli altri processi così si generano dei page fault a cascata.

Questo effetto è chiamato paginazione degenere o Thrashing perché si spende più tempo nella paginazione che nella esecuzione dei processi cioè un processo impiega il suo tempo facendo quasi sempre solo swapping di pagine di memoria.

*Page fault : eccezione generata da un processo che vuole accedere a una pagina presente nel suo spazio di indirizzamento virtuale ma non nella memoria fisica.*

In generale in un sistema se la CPU è sottoutilizzata il SO potrebbe ritenere di alzare il grado di multiprogrammazione; questa scelta comporta nuovi processi nel SO che possono sottrarre pagine ai processi già presenti.

Accade quindi che i processi che in un determinato istante generano page fault vengono tolti dalla coda dei processi pronti ed inseriti nella coda dei processi d’attesa. Con il passare del tempo si svuota la coda dei processi pronti e tutti i processi entrano nella coda dei processi d’attesa che attendono le pagine.

La CPU in questo contesto risulta ancora una volta sottoutilizzata ed altri processi potrebbero essere lanciati così il sistema si “ingolfa”.

Questi effetti si possono limitare usando un algoritmo di sostituzione locale o un algoritmo di sostituzione per priorità. Con la sostituzione locale, se un processo ricade nell’attività di paginazione degenere, non può sottrarre frame a un altro processo e quindi provocarne a sua volta la degenerazione. Però se ad un processo sono concessi pochi frame rimane nella coda d’attesa e quindi aumenta il tempo di servizio medio di un page fault perché la coda si allunga. Di conseguenza, il tempo effettivo d’accesso aumenta anche per gli altri processi.

Per evitare queste situazioni, occorre sapere quali oltre che quante pagine servono.

Per cercare di sapere quanti frame “servano” a un processo si impiegano diverse tecniche ad esempio il modello dell’insieme di lavoro (working-set) che definisce il modello di località d’esecuzione del processo.

Il modello di località stabilisce che un processo, durante la sua esecuzione, si sposta di località in località. Una località è un insieme di pagine usate attivamente. Generalmente un programma è formato da parecchie località diverse, che sono sovrapponibili. Ad esempio, quando s’invoca una procedura, questa definisce una nuova località in cui si fanno riferimenti alla memoria per le istruzioni della procedura, per le sue variabili locali e per un sottoinsieme delle variabili globali.

Quando la procedura termina, il processo lascia questa località, poiché le variabili locali e le istruzioni della procedura non sono più usate attivamente.

Quindi, le località sono definite dalla struttura del programma e dalle relative strutture dati. Il modello di località sostiene che tutti i programmi mostrino questa struttura di base di riferimenti alla memoria.

Si supponga di allocare a un processo un numero di frame sufficiente per sistemare le sue località attuali. Finché tutte queste pagine non si trovano in memoria, si verificano page fault relative a tali località; quindi, finché le località non vengano modificate, non hanno luogo altre page fault. Quindi se si assegnano meno frame rispetto alla dimensione della località attuale, la paginazione del processo degenera, poiché non si possono tenere in memoria tutte le pagine che il processo sta usando attivamente.

**Interrogazione ciclica**

Il protocollo fra la CPU e un controllore avviene attraverso la negoziazione (handshaking). Il controllore specifica il suo stato per mezzo del bit busy del registro status. Pone a 1 il bit busy quando è impegnato in una operazione, e lo pone a 0 quando è pronto per il comando successivo. La CPU comunica le sue richieste tramite il bit command-ready del registro command e pone questo bit a 1 quando il controllore deve eseguire un comando.

La CPU scrive in una porta coordinandosi con un controllore tramite negoziazione in questo modo:

La CPU legge ripetutamente il bit busy fino a che esso non vale 0.

La CPU pone a 1 il bit write nel registro command e scrive un byte nel registro data-out.

La CPU pone a 1 il bit command-ready.

Quando il controller si accorge che il bit command-ready è posto a 1, pone a 1 il bit busy.

Il controller legge il registro command e riceve il comando write, legge il registro data-out per ottenere il byte da scrivere, ed effettua l’operazione di scrittura nel dispositivo.

Il controller pone a 0 il bit command-ready. Se l’operazione di I/O è stata eseguita correttamente pone a 0 il bit error nel registro status, e pone a 0 il bit busy per indicare che l’operazione è terminata.

Questa sequenza viene ripetuta per ogni byte. Durante l’esecuzione del passo 1 la CPU è in attesa attiva o in interrogazione ciclica: itera la lettura del registro status fino a che il bit busy assume il valore 0.

**Metodi di allocazione**

**Allocazione contigua :**

Ogni file deve occupare insiemi di blocchi contigui nel disco così per accedere al file occorre solo la locazione iniziale (# blocco iniziale) e la lunghezza (numero di blocchi). L’accesso al file può essere sequenziale cioè il file system memorizza l’indirizzo dell’ultimo blocco e se è necessario legge il blocco successivo; oppure può essere ad accesso casuale ovvero quando si vuole accedere ad un blocco i che inizia dal blocco b sarà b+i.

L’allocazione contigua ha due vantaggi significativi: cioè è semplice da implementare, dato che tutti i blocchi del file sono univocamente identificati da un numero e le prestazioni sono eccellenti dal momento che l’intero file può essere letto dal disco con singola operazione.

Mentre presenta alcuni problemi; una difficoltà riguarda l’individuazione dello spazio per un nuovo file.

Questo problema è un’applicazione particolare del problema generale dell’allocazione dinamica della memoria e quindi i più comuni criteri di scelta sono first-fit, best-fit e worst-fit. Simulazioni hanno dimostrato che i primi due sono più efficienti del terzo però questi algoritmi soffrono la frammentazione esterna cioè assegnando e liberando lo spazio per i file, lo spazio libero dei dischi viene frammentato in tanti piccoli pezzi e questo può essere un problema abbastanza grave perché spezzettando la memoria si creano pezzi dove il più grande non soddisfa una richiesta. Questo problema si può prevenire copiando un intero file system su un altro disco o nastro così da liberare un grande spazio contiguo e quindi alla fine si copiano i file sul disco ma ciò è pesante.

Un altro problema consiste nel conoscere la dimensione del file da creare ad esempio se un file riceve poco spazio può essere impossibile estenderlo soprattutto quando si usa il criterio di allocazione best-fit.

**Allocazione concatenata :**

L’allocazione concatenata risolve i problemi dell’allocazione contigua quindi non si ha la frammentazione esterna. Ogni file è costituito da una lista concatenata di blocchi del disco i quali possono essere sparsi in qualsiasi punto del disco stesso.

Ogni blocco contiene un puntatore al blocco successivo mentre la directory contiene un puntatore al primo e all’ultimo blocco del file.

Il primo svantaggio di questo metodo è la memoria necessaria a contenere i blocchi ai puntatori successivi.

La soluzione più comune a questo problema consiste nel raccogliere i blocchi in gruppi, detti **cluster**, e nell’allocare i cluster anziché i blocchi così i puntatori usano una quantità di spazio che si riduce in modo proporzionale alla quantità di cluster e inoltre si ha un miglioramento delle prestazioni per via del numero minore di riposizionamenti della testina però si ha una maggiore frammentazione interna perché se un cluster è parzialmente pieno si spreca più spazio rispetto ad un *blocco parzialmente pieno*.

Lo svantaggio principale è che può essere utilizzata efficientemente solo per file ad accesso sequenziale perché per trovare l’i-esimo blocco di un file occorre partire dall’inizio del file e seguire i puntatori finché non si arriva all’i-esimo blocco. Quindi, ogni accesso a un puntatore implica una lettura del disco, e talvolta un posizionamento della testina sul disco.

Un altro problema riguarda l’affidabilità perché può succedere che un puntatore viene perso o danneggiato così da rendere inaccessibili grandi quantità di dati.

Una soluzione parziale a questo problema consiste nell’utilizzare liste doppiamente concatenate oppure nel memorizzare il nome del file e il relativo numero di blocco in ogni blocco (overhead, richiesta in più rispetto al necessario).

**Allocazione indicizzata :**

L’allocazione indicizzata raggruppa tutti i puntatori in una sola locazione chiamata blocco indice.

Per accedere ad un file, si carica in memoria il suo blocco indice e si utilizzano i puntatori in esso contenuti.

E’ sufficiente, come nei casi precedenti, gestire un solo intero (l’indice del primo blocco) per ogni elemento della directory per essere in grado di individuare tutti i blocchi, qualsiasi sia la grandezza del file.

Quindi, ogni file ha il proprio blocco indice che è un array d’indirizzi di blocchi del disco.

**Soluzione di Peterson**

L'algoritmo di Peterson o algoritmo tie-breaker è un algoritmo sviluppato nella teoria del controllo della concorrenza per coordinare due o più processi o thread che hanno una sezione critica in cui deve esservi mutua esclusione. L'algoritmo assicura la corretta sincronizzazione, impedendo lo stallo (deadlock) ed assicurando che soltanto un processo alla volta possa eseguire una sezione critica (serializzazione).

//'' dichiarazione delle variabili globali comuni''

boolean in1 = false, in2 = false;

int turno;

// processo #1

Process CS1 {

while(1) {

in1 = true;

turno = 2;

while (in2 && turno == 2)

; // finché è il turno del processo #2, il processo #1 rimane all'interno del while

<sezione critica 1>

in1 = false;

<sezione non critica 1>

}

}

// processo #2

Process CS2 {

while(1) {

in2 = true;

turno = 1;

while (in1 && turno == 1)

; // finché è il turno del processo #1, il processo #2 rimane all'interno del while

<sezione critica 2>

in2 = false;

<sezione non critica 2>

}

}

Se il processo #1 viene eseguito per primo, in1 viene impostato a true prima di impostare turno a 2. Dal momento che in2 è stato inizializzato a false, la condizione dell'iterazione while non è soddisfatta e il processo #1 accede alla sezione critica.

Se nel frattempo viene avviato il processo #2, questo imposterà in2 a true e turno a 1. in1 è già stato impostato a true dal processo #1, perciò la condizione dell'iterazione while del processo #2 è soddisfatta, così che questo deve aspettare. Solo dopo che il processo #1 ha abbandonato la sezione critica in1 diventa false, e il processo #2 può accedere alla sua sezione critica.

Se il processo #1 viene nel frattempo riavviato, imposterà turno a 2, e dovrà aspettare che il processo #2 abbia abbandonato la sua sezione critica.

La mutua esclusione è garantita dal fatto che il controllo di in e di turno viene fatto in modo atomico.

L'assenza di deadlock viene garantita dal fatto che solamente una delle due condizioni while può essere vera nello stesso momento.

Il progresso dell'elaborazione è garantito dal fatto che se uno dei due processi tenta di entrare e l'altro non è in sezione critica può tranquillamente entrare.

L'attesa di un processo è limitata in quanto, se i due processi tentano di entrare in sezione critica, entrano alternamente.

**Test-and-set**

In informatica l'istruzione test-and-set viene usata per scrivere in una locazione di memoria e restituire il suo vecchio valore come una singola operazione atomica (non interrompibile). Se diversi processi possono accedere alla stessa area di memoria, e se un processo sta eseguendo una test-and-set, nessun altro processo può iniziare un'altra test-and-set finché il primo processo non ha terminato la propria.

bool Test-and-Set(bool \*target){

bool val;

val = \*target;

\*target = TRUE;

return val;

}

L'istruzione permette a più processi coordinanti di modificare una variabile condivisa senza incorrere nell'inconsistenza dei dati cioè modificandola senza il rischio che con la prelazione del sistema operativo un processo non concluda la modifica dei dati portando così a risultati imprevedibili. La zona di codice in cui il programma modifica questi dati condivisi è detta *sezione critica*. L'obiettivo è che su n processi solamente uno entri in sezione critica evitando così che altri processi cooperanti ad esso accedano a dati non ancora completamente modificati, mantenendo la coerenza dei dati.

**Algoritmo del fornaio**

L'algoritmo del fornaio è uno dei metodi di mutua esclusione che trovano applicazione pratica nella programmazione parallela per serializzare l'esecuzione delle sezioni critiche da parte di più processi o thread concorrenti.

L'algoritmo deve il nome al suo inventore Leslie Lamport, che propose l'analogia con il modello reale di una frequentata panetteria, dove i clienti strappano un numero prima di mettersi in fila ad aspettare il proprio turno. I clienti del fornaio rappresentano i task in attesa del proprio turno per eseguire la sezione critica.

do {

scelta[i]=true;

numero[i] = max(numero[0], numero[1], …., numero[n-1])+1;

scelta[i] = false;

for(j=0; j<n;j++) {

while(scelta[j]);

while((numero[j]!=0) && ((numero[j],j)<numero[i],i)));

}

Sezione critica

Numero[i]=0;

Sezione non critica

} while(true);

**Strutture RAID**

I moderni calcolatori possono avere più dischi ciò rende possibile l’aumento della frequenza con cui i dati si possono leggere o scrivere, e permette di migliorare l’affidabilità della memoria secondaria.

Esistono varie tecniche per l’organizzazione dei dischi chiamate batterie ridondanti di dischi (RAID) che hanno lo scopo di affrontare i problemi di prestazioni e affidabilità.

In passato strutture RAID composte da piccoli dischi economici erano viste come un’alternativa vantaggiosa

rispetto a costosi dischi di grande capacità; invece oggi si impiegano per la loro maggiore affidabilità e velocità di trasferimento dei dati.

**Miglioramento dell’affidabilità tramite ridondanza**

Per quanto riguarda l’affidabilità è più probabile che uno dei dischi in un insieme di n dischi si guasti rispetto ad uno specifico disco isolato presenta un guasto. Così se si memorizza una sola copia dei dati, allora ogni guasto di un disco comporterebbe la perdita di una notevole quantità di dati.

La soluzione a questo problema si ha nell’introdurre una ridondanza, cioè nel memorizzare informazioni che non sono normalmente necessarie ma che si possono usare in caso di guasto per ricostruire le informazioni perse.

Il metodo più semplice è la copiatura speculare (mirroring) cioè ogni disco logico consiste di due dischi fisici e ogni scrittura si effettua su entrambi i dischi. Se uno dei due dischi si guasta i dati si possono leggere dall’altro ma se il secondo disco si guasta prima della sostituzione del primo disco (già guasto) i dati si perdono.

**Miglioramento delle prestazioni tramite il parallelismo**

Con la copiatura speculare, se si può accedere in parallelo a più dischi, la frequenza con la quale si possono gestire le richieste di lettura raddoppia perché una richiesta si può inviare a uno dei due dischi.

La capacità di trasferimento di ciascuna lettura è uguale a quella di un sistema a singolo disco ma il numero di letture per unità di tempo raddoppia. Però attraverso l’uso di più dischi è possibile anche migliorare la capacità di trasferimento distribuendo i dati in sezioni su più dischi ciò è chiamato sezionamento dei dati è consiste nel distribuire i bit di ciascun byte su più dischi avremo così il sezionamento a livello dei bit.

Invece se distribuiamo i blocchi di un file su più dischi avremo il sezionamento al livello dei blocchi. Sono possibili anche altri livelli di sezionamento come quelli bastati sul byte di un settore o sui settori di un blocco.