1 Lezione del 18-11-25

Continuiamo a parlare della paginazione.

1.0.1 Descrittori di pagina

Come per i segmenti, possiamo associare alle pagine **descrittori di pagina** che contengono informazioni riguardo all'accesso alla pagina.

Le informazioni tipiche sono, ancora una volta, i bit di accesso in lettura (R) o scrittura (W).

1.1 Paginazione su domanda

Altre informazioni contenute nei descrittori di pagina sono quelle legate alla **paginazione su domanda**, praticamente analoghe a quanto discusson in 17.4:

- Bit **U**, detto di *uso*, aggiornato quando si usa (legge o scrive) la pagina;
- Bit **M**, detto di *modifica*, aggiornato quando si fa un operazione di scrittura sulla pagina;
- Bit **P**, detto di *presenza*, indica se la pagina è effettivamente caricata in memoria o se ne é fatto swap out.

Come per la segmentazione, il bit P associerà alla pagina un certo significato:

- Bit *P* a 1: la pagina è valida e caricata in memoria;
- Bit *P* a 0: la pagina può non essere valida, oppure può essere valida ma non caricata in memoria.

In questo caso utilizziamo il bit P per lanciare condizionalmente un'eccezione di *page fault* in caso di accesso a pagine con bit P a 0. Capire se la pagina va caricata o l'accesso è effettivamente invalido è quindi compito dell'handler predisposto dal S/O a tale eccezione.

Ricordiamo che la paginazione su domanda è un'approccio a rilocazione *dinamica*, allocazione *non contigua*, spazio virtuale *unico* e caricamento *su domanda*.

1.1.1 Gestione del page fault

Abbiamo detto che il page fault è un'eccezione, generata dal sistema, e completamente incodizionata dal programma.

In altre parole, il processo è ignaro dell'esistenza dei page fault e qualsiasi problema si verifichi deve essere risolto dal S/O. Tutto ciò che potremmo osservare dal lato processo è un aumento del tempo di *turnaround*, cioè dell'intervallo dal tempo di revoca al successivo assegnamento CPU, impiegato chiaramente a caricare la pagina richiesta.

Andiamo quindi a dettagliare come si gestisce effettivamente il page fault:

1. Eseguendo il codice del processo (p_0) in esecuzione, la CPU genera un indirizzo virtuale che corrisponde ad una pagina non caricata in memoria;
2. La MMU riceve tale indirizzo, esplora la *tabella delle pagine del processo* fino all'indirizzo, controlla il bit P è lo trova uguale a 0. Viene lanciata un'eccezione di page fault;

3. Il supporto hardware alle eccezioni (cioè lo stesso a supporto delle interruzioni) carica il puntatore alla routine di gestione del page fault e la mette in esecuzione.

   Ora, in memoria assumiamo esista una tabella, detta *tabella delle pagine fisiche*, che associa alle pagine fisiche un informazione riguardante se quella pagina è libera o meno;

4. Nel caso una pagina fisica (`pf`) sia trovata libera, si fa una copia (con DMA o altri metodi) dalla *memoria di swap* alla *memoria centrale* della pagina richiesta. L'indirizzo della pagina in memoria di swap può essere, ad esempio, memorizzato nella sezione dedicata all'indirizzo fisico del descrittore di pagina non caricta.

   Quindi si aggiorna l'indirizzo fisico del descrittore di pagina con quello di `pf`, e si mette il bit di presenza a 1;

5. A questo punto i primi 2 passaggi si ripetono, con la differenza che la MMu trova la pagina desiderata con bit P uguale a 1, la tabella delle pagine fisiche contiene la pagina effettivamente richiesta, e il processo può proseguire.

1.1.2 Rimpiazzamento di pagine

La paginazione su domanda richiede spesso che, per liberare spazio in memoria centrale per la nuova pagina, si deallochi una vecchia pagina. Questa viene detta *vittima*, e l'intero processo viene detto **rimpiazzamento**.

Dettagliamo anche questo processo, immaginando di avere un'altro processo (p_1) in esecuzione in parallelo a quello del primo esempio:

1. Quando diventa necessario caricare la pagina del processo p_0, supponiamo che p_1 sia in possesso dell'indirizzo fisico `pf` (cioè che `pf` sia indirizzo fisico in uno dei descrittori di pagina di p_1);

2. In questo caso si mette in esecuzione un dato algoritmo di *rimpiazzamento*, e si dealloca la pagina di p_1 nella memoria di swap.

   Come abbiamo detto, per ricordare dove abbiamo messo la pagina memorizziamo nella sezione dedicata all'indirizzo fisico, l'indirizzo in memoria di swap dove abbiamo memorizzato la pagina;

3. Alla fine di questo processo, il processo p_1 si trova nella stessa situazione in cui si trovava p_0 prima di accedere alla pagina: questa esiste nella tabelle delle pagine di processo, ma non punta ad un entrata valida della tabella delle pagine fisiche (bensì ad una pagina di cui si è fatto swap).

L'andamento del rimpiazzamento delle pagine porta naturalmente alla formazione di un *working set* associato ad ogni processo, cioè l'insieme di pagine su cui quel processo fa accesso con frequenza. Il working set inizialmente si espande, ma poi tende a rallentare la sua crescita.

Gli algoritmi di rimpiazzamento che consideriamo saranno quindi:

- L'algoritmo *ottimo* sarebbe quello che rimpiazza le pagine che non verranno più riferite, o almeno, che verranno riferite più tardi nel tempo. Questo però è impossibile da ottenere, in quanto implicherebbe di conoscere il futuro;

- Possiamo quindi adottare un algoritmo **FIFO** (*First In First Out*): la vittima scelta è la pagina che è da più tempo in memoria. Questo è probabilmente l'algoritmo più semplice che potremmo usare.

  Un problema dell'approccio FIFO è che esiste un sottoinsieme del working set che probabilmente resterà quasi sempre rilevante all'esecuzione del programma, e che quindi non converrà caricare. Questo sottoinsieme sarà ad esempio quello che contiene le strutture dati di base del programma, le costanti e le variabili statiche;

- Un'altro algoritmo, che risolve quest'ultimo problema, è il **LRU** (*Least Recently Used*): la vittima è la pagina meno recentemente utilizzata.

  Questo chiaramente richiede un qualche tipo di statistica sull'uso delle pagine (a questo tornano utili i bit U e M che avevamo predisposto). Un'alternativa è quella di memorizzare, anziché il bit U, un *timestamp* nel descrittore di pagina. A questo pusto basterà semplicemente aggiornare la pagina con timestamp più remoto. Questo però ha chiaramente delle problematiche in termini di overhead (non solo lo spazio che va allocato nel descrittore per il timestamp, ma il tempo impiegato ad aggiornare il timestamp ad ogni operazione sulla pagina).

Può essere utile parlare di *rimpiazzamenti locali*: quando un processo provoca un page fault, si sceglie come vittima una pagina presente nel working set di tale processo. Questo impedisce le interferenze con altri processi, e assegna la penalità associata al page fault al processo stesso che ha generato il page fault.

Se l'algoritmo di scelta delle vittime non è ottimale il sistema può andare in *trashing*. Si dice che il sistema è in trashing quando la percentuale di page fault sugli accessi in memoria supera una certa soglia. La condizione (ideale, fortunatamente non reale) di *trashing completo* è quella in cui ogni accesso in memoria provoca un page fault.

1.1.3 Rimpiazzamento second-chance

Un'altro algoritmo di selezione delle vittime è il cosiddetto algoritmo di rimpiazzamento *second-chance*, o algoritmo dell'*orologio*.

Supponiamo di fare rimpiazzamento locale, e cioè di considerare come possibili vittime solo le pagine nel working set del processo che ha provocato il page fault. Prevediamo quindi un puntatore alla vittima, che inizialmente corrisponde al puntatore alla pagina da più tempo in memoria dell'approccio FIFO. La differenza col FIFO è però data dal fatto che, prima di deallocare la pagina, si controlla il suo bit U. Nel caso questo sia impostato, si annulla e si procede con le pagine successive: questo processo si ripete finché non si trova una pagina con bit U a 0 (e questa verrà deallocata).

Il comportamento ottenuto è quindi che si cerca di evitare di deallocare pagine che sono state usate: si torna sulla prima pagina considerata solo nel caso in cui anche tutte le altre pagine sono già state usate.

1.2 Segmentazione paginata

Concludiamo l'argomento della gestione di memoria parlando della **segmentazione paginata** (o *segmentazione con paginazione*): questo è un approccio a rilocazione *dinamica*, allocazione *non contigua*, spazio virtuale *segmentato* e caricamento *su domanda*.

L'idea della segmentazione paginata è che la CPU produce sempre indirizzi:

$$x =< \text{segmento}, \text{scostamento} >$$

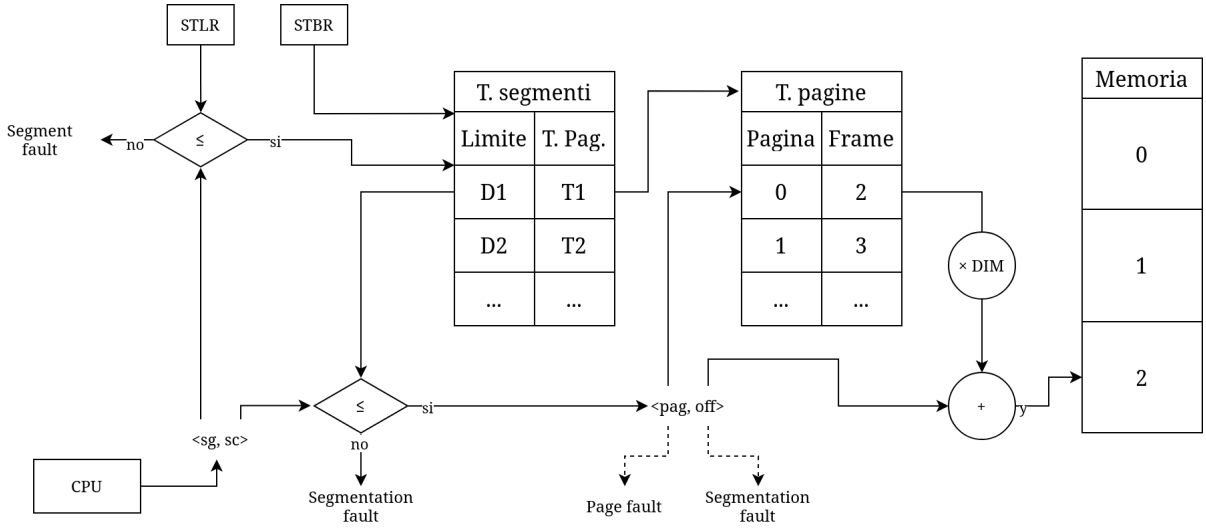
La differenza è però che lo *scostamento* (non ancora offset) viene diviso in due sezioni:

$$\text{offset} = \text{pag} \,|\, \text{offset}$$

cioè viene usato per indicizzare una pagina e l'offset all'interno di tale pagina. Quello a cui punterà il segmento sarà quindi un *descrittore di segmento*, contenente a sua volta un puntatore alla *tabella delle pagine* associata a tale segmento. Lo scostamento verrà poi usato, come nei normali approcci a paginazione, per indicizzare su tale tabella di pagine.

Questo può essere meglio schematizzato come segue:

-3cm-3cm



dove l'eccezione di *page fault*, che può evidentemente essere lanciata, è stata riportata ma non dettagliata nel funzionamento: chiaramente bisognerà mantenere da qualche parte (nella tabella dei segmenti o come costante del S/O) un numero massimo di pagine per segmento, e controllare la pagina richiesta dallo scostamento con tale numero. La stessa cosa vale per la *segmentation fault* causata dalla paginazione: qui sappiamo, come già visto in 17.5, di avere dimensione di pagina fissa.

Notiamo quindi come questo approccio può portare a 3 tipi di eccezione diversi:

- *Segment fault*: lanciata quando si richiedono segmenti inesistenti (fuori dai bound o non presenti);
- *Segmentation fault*: lanciata quando si effettuano accessi invalidi all'interno del segmento o della pagina;
- *Page fault*: lanciata quando si accede ad una pagina di segmento inesistente (fuori dai bound o non presente).

Questo è l'approccio tipicamente usato da UNIX in segmentazione. Nelle prossime sezioni, infatti, approfondiremo l'approccio che questo S/O usa alla gestione della memoria.

1.3 Gestione della memoria in UNIX

In UNIX la tabella delle pagine fisiche viene detta **core map**. Questa ha quindi il compito di mantenere i frame fisici di pagina, e informazioni riguardo a quale pagina contengono, o se questi sono liberi.

Vengono quindi stabilite 3 variabili per i limiti delle pagine libere:

- *lotsfree*: numero minimo di frame liberi per evitare sostituzione;
- *minfree*: numero minimo di frame liberi necessari per evitare lo swapping dei processi;
- *desfree*: numero minimo di frame desiderabile per un buon funzionamento del sistema.

Chiaramente vale la diseguaglianza:

$$minfree < desfree < lotsfree$$

La sostituzione delle pagine viene quindi effettuata come segue:

- Un processo di sostituzione di pagine, detto **pagedaemon**, esegue periodicamente e sostituisce le pagine solo se:

$$num\text{-}frame\text{-}liberi < lotsfree$$

- Il processo **swapper**, che si occupa invece di effettuare swap in e swap out di processi, fa lo swap out di interi processi se sono soddisfatte le seguenti condizoni:

$$num\text{-}frame\text{-}liberi < min\text{-}free \wedge num\text{-}medio\text{-}frame\text{-}liberi < des\text{-}free$$

  La seconda condizione si basa su una statistica fatta sul numero medio di frame liberi. Questo ci permette di effettuare swapping solamente nel caso in cui si hanno picchi di occupazione dei frame fisici che si prolungano per un certo periodo di tempo, mantenendo il numero di pagine libere sotto la media desiderata.

1.4 Gestione delle periferiche

Veniamo allora ad un altro concetto fondamentale dei S/O: la **gestione delle periferiche**. Viene da sé che l'ambiente delle periferiche hardware è estremamente variegato, e l'obiettivo sarà quindi quello di fornire alle applicazioni la possibilità di accedervi in maniera unificata.

Di base, prevediamo un *controllore* per ogni *periferica*: i controllori sono quelli che vengono effettivamente montati sul bus, e vengono visti dal processore come *interfacce*, a cui si accede secondo una modalità *a porte* ben definita (con la differenza di alcune interfacce che potrebbero essere montate *in memoria*).

Come primo esempio dell'eterogeneità delle periferiche disponibili ai moderni calcolatori, anche prima delle differenze tecniche nel modo in cui vi si accede, possiamo notare la grande differenza in termini di *velocità di trasferimento*: passiamo da dispositivi come tastiere e mouse vecchio stile (velocità nell'ordine del Kb/sec) a moderni dispositivi di rete Ethernet (125 Mb/sec).

1.4.1 Sottosistemo di I/O

In particolare, prevederemo un **sottosistema di I/O**, che avra il compito di nascondere i dettagli hardware dei controllori dei dispositivi.

Dal sottosistema di I/O ci aspettiamo il compimento dei seguenti compiti:

- Definizione di uno *spazio dei nomi* con cui identificare in maniera univoca i dispositivi (spazi di indirizzamento PCI, ecc... non verranno visti nel dettaglio in questo corso);
- Gestione dei *malfunzionamenti* dei dispositivi, senza che debbano occuparsene le applicazioni;
- Garanzia della *sincronizzazione* tra l'attività di un dispositivo e del processo che lo ha attivato: il programmatore dovrà essere libero di realizzare la logica del programma senza preoccuparsi della sincronizzazione esplicita con i dispositivi;
- *Bufferizzazione*, cioè disaccopiamento temporale e spaziale tra processi e periferiche (i processi forniscono *buffer* in memoria condivisa o privata, che vengono quindi riempiti dalla periferica, attraverso il S/O, in differita).

1.4.2 Organizzazione logica dell'I/O

Dal punto di vista **logico** il sottosistema di I/O è estremamente stratificato.

- A livello *utente*, avremo che i **processi applicativi** (cioè le applicazioni) si interfacciano con **librerie** che espongono funzionalità offerte da *interfacce applicative* fornite dal S/O;
- A livello *S/O*, si offre l'**interfaccia applicativa** (**I/O API**) vera e propria per la gestione delle periferiche.
  - Questa si interfaccia con una certa quantità di strutture e ruotine gestite dal S/O che esistono a priori dalle periferiche. Tali strutture formano la cosiddetta interfaccia **device independent** del sottosistema di I/O. Per tornare all'esempio di Unix, si ha che l'associazione periferica $\leftrightarrow$ file è parte dell'interfaccia device independent. Sono dello stesso tipo anche tutte le varie primitive `open()`, `close()`, `read()`, `write()`, ecc...
  - Al di sotto dell'interfaccia device independent si trova chiaramente una parte di interfaccia **device dependent**. Questa è composta da strutture e routine gestite dal S/O che esistono direttamente in funzione delle periferiche montate nel sistema.

    Parte dell'interfaccia device dependent sono i *driver*, cioè sostanzialmente dai gestori delle *interruzioni* lanciate dai dispositivi. Chiaramente i driver sono strettamente legati ai dispositivi che gestiscono, in quanto devono conoscere i loro dettagli di funzionamento.

    Sempre riconducendosi all'esempio di Unix, abbiamo che anche questo livello implementa le sue `read()`, `write()`, con la particolarità che queste si preoccupano del funzionamento effettivo della periferica. In questo, la chiamata della `read()` di livello indipendent si tradurrà in una chiamata alla `read()` di livello dependent, e lo stesso per la `write()`, ecc...
- Alla fine della gerarchia c'è il livello *hardware*, formato dall'**interfaccia di accesso ai controllori**, e quindi coi **controllori** veri e propri.

1.4.3 Bufferizzazione

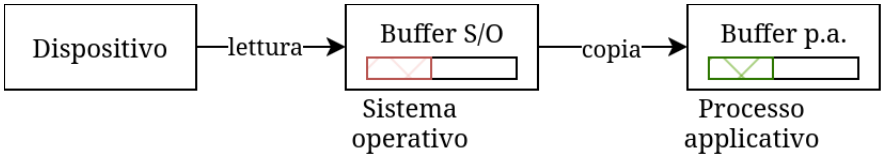
Vediamo nel dettaglio l'attività di **bufferizzazione** svolta dal sottosistema di I/O.

Quello che vogliamo fare è, ad esempio per un operazione di lettura, permettere ai processi che chiamano primitive di ingresso di fornire un *buffer*. Questi verranno quindi bloccati per la durata del trasferimento, e il S/O si impegnerà a riempire tale buffer con quanto ottenuto dalla periferica.

Il disaccoppiamento che cerchiamo è sia *spaziale* che *temporale*:

- **Spaziale**, in quanto non chiediamo al programmatore di conoscere le dimensioni di buffer ideali per la periferica, ma ci prendiamo la briga di gestire eventuali buffer intermedi e di riempire il buffer da egli fornito;
- **Temporale**, in quanto il buffer viene riempito in differita, dopo che viene fornito dal programmatore.

Questo approccio richiede la presenza di 2 buffer:



- Un buffer di **S/O**, la cui dimensione è effettivamente dettata dal dispositivo stesso, e che il sistema operativo usa per gestire immediatamente le operazioni di scrittura dalla periferica verso il bus;
- Un buffer di **processo applicativo**, dichiarato dal programmatore, all'interno del quale ci aspettiamo di trovare i dati (nel nostro esempio dopo una `read()`). Questo viene riempito attaverso operazioni di *copia* dal buffer di S/O.

1.4.4 Funzioni indipendenti

Vediamo quindi le funzioni predisposte *a priori* dei dispositivi, cioè quelle presenti al cosiddetto livello **device indipendent**:

- **Spazio dei nomi** dedicato ai dispositivi: questo deve permettere un qualche tipo di associazione:

  $$< \text{nome simbolico} > \implies \texttt{\&id\_desc}$$

  dove `&id_desc` è un puntatore a una qualche struttura dati `id_desc`, detta *descrittore di dispositivo*, che rappresenta informazioni riguardo a tale dispositivo.

  Ricordiamo che questo in Unix è effettuato attraverso la metafora dispositivo $\leftrightarrow$ file, e quindi le funzionalità di *naming* dei dispositivi sono implementate direttamente all'interno del file system. Possiamo infatti anticipare che il file system Unix è gestito attraverso descrittori di file detti *inode*, e che alcuni *inode* particolari hanno il compito di rappresentare proprio i dispositivi.

- Gestione dei **malfunzionamenti**;
- Gestione degli accessi **concorrenti** allo stesso dispositivo (questa si realizza lato S/O usando le primitive di sincronizzazione già ampiamente studiate).

1.4.5 Funzioni dipendenti

Vediamo quindi le funzioni *dipendenti* dai dispositivi, cioè quelle del cosiddetto livello **device dependent**.

Qui il nostro compito è quello di definire primitive come la `read()`

```
read(descrittore, buffer, dim_buffer);
```

che offrano un'interfaccia univoca all'accesso ai dispositivi.

Il supporto S/O delle funzioni dipendenti è dato, come abbiamo anticipato, dai **driver**. Il driver è infatti quella collezione di strutture e routine (principalmente gestori di interruzione) predisposti all'interoperazione fra S/O e dispositivo, ed è l'unico componente software che si preoccupa effettivamente delle modalità di funzionamento dell'hardware del dispositivo.

Abbiamo già detto che il driver si interfaccia in particolare con il *controllore* di dispositivo, e non con il dispositivo stesso. Un controllore generico è formato dalle seguenti componenti:

- Registri di **controllo**, su cui la CPU può scrivere, che dettano al dispositivo quali operazioni compiere;
- Registri di **stato**, da cui la CPU può leggere, che forniscono informazioni riguardo allo stato del dispositivo;
- Registri di **buffer** (o *dati*), su cui la CPU può leggere, scrivere o entrambi a seconda del tipo di dispositivo. Questi si occupano della lettura e scrittura effettiva di dati da e sul dispositivo.

Il funzionamento è quindi il seguente:

1. La CPU scrive comandi sui registri di controllo del controllore, legge lo stato dai registri di stato del controllore, e scambia informazioni attraverso i registri dati del controllore;
2. Il controllore invia *segnali* al dispositivo, e legge *dati* dal dispositivo. Segnali e dati scambiati col dispositivo sono direttamente influenzati da quanto il controllore ha ricevuto comunicando con la CPU.

In questo, il controllore si comporta effettivamente da *buffer* fra CPU e dispositivo.

I controllori hanno poi quasi sempre la possibilità di generare **interruzioni** per la CPU (attraverso un componente intermedio detto *controllore di interruzioni*, che si occupa di organizzare gerarchicamente e bufferizzare le interruzioni per la CPU). Le interruzioni sono ormai fondamentali alla gestione dei dispositivi: l'esempio tipico è quello di generare un'interruzione quando si ha un aggiornamento dei bit di stato (ad esempio per segnalare nuovi dati da leggere o la terminazione di un'operazione in scrittura).