Traccia TPSIT - Algoritmi sulla memoria

TPSIT - classe 3CII

1.1 Prerequisiti

Abbiamo visto la memoria virtuale, come questa permetta ad ogni processo di indirizzare più memoria di quanta realmente se ne abbia nel computer, di come viene gestito il ciclo di calcolo degli indirizzi e come questi influenzino il programmatore se non abbiamo a che fare con dispositivi fisici come la MMU.

Continuiamo a scontrarci con i limiti fisici: la memoria fisica **NON É** illimitata. La segmentazione e la paginazione, fortunatamente, ci viene in contro (assieme allo swapping.... domandina alla classe su quali sono i cicli dello swapping).

Trashing

cit: Il problema di Azzolin che aspetta il foglio di carta per continuare a scrivere (per i comuni mortali: I/O waiting, al posto di busy waiting, metto in coda d'esecuzione e poi procedo a continuare: context switch, interrupt, vettore delle interrupt, kernel mode/ supervisor mode per la cpu). Ma il $p = P(E)$, con E::"buon compromesso tra massima multiprogrammazione e minimo numero di page fault"?
grafichino sul primo quadrante $f(x) = \frac{1}{x}$; ordinata grado multiprogrammazione, ascissa nro di page fault.
considerazione: se ho troppi processi in memoria, ho prestazioni pessime, se ho troppi pochi processi in memoria, ho prestazioni pessime.

Come pagino?

RICORDIAMO però che PAGING è diverso da SWAPPING:

- Swapping: scambio di interi processi da/per il backing store;
- Swapper: processo che implementa una politica di swapping (scheduling di medio termine);
- Paging: scambio di gruppi di pagine (sottoinsiemi di processi) da/per il backing store;
- Pager: processo che implementa una politica di gestione delle pagine dei processi (caricamento/scaricamento).

La confusione nasce dal fatto che in alcuni S.O. il pager viene chiamato "swapper" (es.: in Linux si chiama `kswapd`). L'una non esclude l'altra!!

Che paginazioni abbiamo quindi:

- *on-demand* o *a richiesta*: cerco di vedere cosa mi serve solo quando mi serve (Euristica?[1] WorkingSet? -> difatto segue il prepaging, forse).
  Considerazioni:

[1] digli anche come si fa.... a grandi linee

  - problema di performance: si vuole un algoritmo di rimpiazzamento che porti al minor numero di page fault possibile; per misurare le performance uso EAT:

    $(1-p)*accessoRAM+p*(overheadPF+t_{swapout}+t_{swapin}+overheadRESTARTitem)$

  - l'area di swap deve essere il più veloce possibile → meglio tenerla separata dal file system (possibilmente anche su un device dedicato) ed accedervi direttamente (senza passare per il file system). Blocchi fisici = frame in memoria.
  - La memoria virtuale con demand paging ha benefici anche alla creazione dei processi.

- prepaging: faccio il mago! Cerco di vedere cosa posso fare in anticipo (e noi ne sappiamo qualcosa con la speculative execution, o no, Francesco?)
  come per le cache:

  - Località spaziale:
    tutto l'intorno della memoria di un processo (frame precedente e antecedente) mi serviranno...[2] Questo ci serve solo nello swap di pagine che sono locali al processo, e non globali al sistema;

  - Località temporale:
    se i riferimenti sono vicini, probabilmente mi servono ancora!

Resident Set: quale parte del programma e dei dati devono stare in memoria affinché il processo possa essere eseguito (con il *Working Set*: : $WS(t,\Delta)$ avremo quello del passato!).

1.2 Algos:

Con *stringa di riferimenti*: : R_α si intende quella stringa che, dato un momento i-esimo, e la funzione $\Psi(R_\alpha, i) \to N$ mi restituisce il riferimento al momento i, tale che corrisponde al carattere in posizione i-esima rispetto a R_α. Con **PF** indichiamo il *page fault*.

OPT - OPTimal

Voglio in ogni maniera avere il miglio assoluto...

Se si potesse prevedere quali sono le mie page fault e le prossime richieste di accesso alla memoria, eliminerei sempre la pagina con il riferimento più lontano in assoluto. Ma noi non siamo dei maghi, quindi non possiamo riuscire a prevedere il futuro se non guardando bene nel passato...

Esempio Quanti page fault avvengono considerando un numero di page frame della RAM da uno a sette e 4 frame?

Abbiamo R_α : $1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6$. Avremo quindi quanto segue:

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	1	1	1	1	1	1	1	1	1	1	1	7	7	7	7	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
			4	4	4	5	6	6	6	6	6	6	6	6	6	6	6	6	6
PF	PF	PF	PF			PF	PF					PF				PF			

Considerazioni: come possiamo vedere, abbiamo solo 8 **PF**. É attuabile? NO! Non posso sapere il futuro! Rimane un ottimo modello di performance.

[2] domanda alla classe: ci servono realmente? NO! qui parliamo di frame che non sono correlati tra loro!

NRU - Not Recently Used

sono quasi un mago...
mi servino 2 bit:

- ***dirty bit***: modifica del blocchetto di memoria
- ***refecenced bit***: ho riferito la memoria

si creano 4 classi ($classe_n$):

1. not ref, not mod;
2. not ref, mod;
3. ref, not mod;
4. ref, mod.

Apparentemente la $classe_1$ è inesistente, ma esistono delle combinazioni per cui può accadere.

Elimino quale sezione di memoria? Ovvio, quella che sarà in $classe_1$! Perchè? beh, easy: quello che non abbiamo minimamente usato speriamo sia quello che veramente non ci serve più. É migliorabile? NO! rimane un abbozzo per OPT. É simulabile altrimenti OPT? CERTO! vedi *clock* e/o *second chance*.

FIFO - First In First Out (o algoritmo del frigorifero....)

quello che scade per prima se ne va! meglio di così si muore... forse.

Note all'esempio: disegniamo una matrice, in orizzontale abbiamo $\tau(i)$ funzione del momento che descrive la memoria nel momento i-esimo, nelle colonne il numero di frame a disposizione nella memoria. Dato quindi $\Psi(R_\alpha, i)$, nella cella più in alto della colonna i-esima della matrice avrò:

- o il nuovo frame caricato (**PF**), che farà shiftare in basso i precedenti frame fino a dover fare lo swap out del frame più in basso;
- o il frame è già in memoria, quindi non abbiamo nulla da fare.

Esempio Quanti page fault avvengono considerando un numero di page frame della RAM da uno a sette e 4 frame?

Abbiamo $R_\alpha : 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6$. Avremo quindi quanto segue:

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	2	3	4	4	4	5	6	2	1	1	3	7	6	6	2	1	1	3	3
	1	2	3	3	3	4	5	6	2	2	1	3	7	7	6	2	2	1	1
		1	2	2	2	3	4	5	6	6	2	1	3	3	7	6	6	2	2
			1	1	1	2	3	4	5	5	6	2	1	1	3	7	7	6	6
PF	PF	PF	PF			PF	PF	PF	PF		PF	PF	PF		PF	PF		PF	

Considerazioni: FIFO si comporta in maniera molto differente rispetto NRU, e come possiamo vedere non tiene minimamente contro del passato! Ci rifacciamo quindi alle strategie di sostituzione locali e globali.

LRU - Least Recently Used

Una buona osservazione del passato ci permette di sapere quando iniziamo a non aver più necessità di qualcosa: vediamo per l'appunto le page che sono meno usate nel tempo passato: quella che avrà il riferimento più vecchio sarà quella da sostituire. Si può pensare d'usare anche NFU (Not Frequently Used), che lavora in maniera analoga.

Esempio Quanti page fault avvengono considerando un numero di page frame della RAM da uno a sette e 4 frame?

Abbiamo $R_\alpha : 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 4, 6, 3, 2, 1, 2, 3, 6$. Avremo quindi quanto segue:

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3
		1	2	3	4	2	1	5	6	6	1	2	3	7	6	3	3	1	2
			1	1	3	4	2	1	5	5	6	1	2	2	7	6	6	6	1
PF	PF	PF	PF			PF	PF				PF	PF	PF			PF			

Considerazioni: Nonostante sia il meglio possibile, non è assolutamente di facile implementazione, in quanto decisamente dispendioso: per quanto possibile, con l'hw a basse prestazioni e senza necessità di microcodici per attuarlo, viene simulato con l'ausilio del "second chance". L'implementazione reale necessiterebbe di tenere traccia dei tempi di riferimento dei frame, ed ad ogni riferimento gestirli in un aggiornamento della coda, al fine di mantenere la stessa. Il costo per attuare una strategia simile è di utilizzare una lista doppiamente linkata e si hanno almeno 6 operazioni per i reinserimenti nella coda dei vari frame.

Second chance o Clock (FIFO improved)

Come abbiamo visto, FIFO non tiene traccia dei riferimenti storici delle pagine... ma se facciamo nuovamente riferimento ad una pagina già nella memoria, forse è il caso di "rimetterla" sopra la pila! In questa maniera, se ad esempio è un riferimento ad un'area di una libreria, come per la chiamata alla funzione `printf`, sappiamo che per certo ci serve nuovamente in futuro. L'applicazione dell'algoritmo avviene all'interno del TLB e viene sempre fatta attraverso l'ausilio del *reference bit*.
Questo algoritmo, ogni volta che bisogna considerare quale pagina eliminare dalla memoria, guarda il reference bit: se quando passa sopra è a 1, allora non sarà la vittima per questo giro, se è a 0, è tra le vittime: di queste, eliminerà quella più vecchia.

Esempio Quanti page fault avvengono considerando un numero di page frame della RAM da uno a sette e 4 frame?

Abbiamo $R_\alpha : 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 4, 6, 3, 2, 1, 2, 3, 6$. Avremo quindi quanto segue:

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	2	3	4	4	4	5	6	6	6	6	3	7	6	6	6	1	1	1	1
	1	2	3	3	$\underline{\underline{3}}$	2	5	5	5	$\underline{\underline{5}}$	2	3	7	7	$\underline{\underline{7}}$	6	6	6	6^1
		1	2	2^1	2^1	1	2	2^1	2^1	2^1	1	2	3	3^1	3^1	3	3	3^1	3^1
			1	1	1^1	4	1	1	1^1	1^1	$\underline{\underline{6}}$	$\underline{\underline{1}}$	2	2	2^1	2	2^1	2^1	2^1
PF	PF	PF	PF			PF	PF				PF	PF	PF			PF			

Considerazioni: Nonostante sia un piccolo miglioramento, parliamo sempre di 4 **PF** in meno!

Ma a lato pratico cosa cambia e perchè usiamo *clock* al posto di *second chance*? Semplicemente, l'algoritmo di clock è più semplice da implementare a livello HW poichè il secondo sfrutta la ciclicità dei numeri data dal numero finito dei bit, mentre con il secondo, nasce la necessità di creare una vera e propria coda in memoria! Un miglioramento si può avere sfruttando l'idea di **NRU**, quindi applicare una FIFO con clock classificando ulteriormente le pagine, quindi definendo se sono riferite o meno.

Non è comunque il meglio che possiamo ottenere, in quanto questo è un abbozzo dell'algoritmo LRU:

1. come *clockSTD*, cerco una pagina che abbia $< 0, 0 >$, se la trovo fine;
2. se al primo giro, non trovo nessuna pagina a clock $< 0, 0 >$, passo a cercare una pagina con $< 0, 1 >$, se la trovo fine;
3. se non trovo nulla, fai un altro giro (torna al punto 1).

Working Set - WSClock

L'idea di base è la medesima di LRU: a noi serve sapere il passato. Sappiamo che se abbiamo un buon compromesso tra numero di pagine caricate in memoria e numero di frame, allora dire che il numero maggiore di page fault che avremo sarà all'inizio dell'esecuzione del processo. Una volta passata questa fase, la località di riferimento sarà caricata in memoria e non avremo più un'alta percentuale di PF in esecuzione. Il concetto di Working Set si rifà quindi all'idea di base del Resident Set. Definiamo quindi come

$$WS(t, \Delta) = \{x : x = \Psi(R_\alpha, i), i : [t - \Delta + 1; t]\}$$

e diciamo che il WS è un'approssimazione della località del processo, ossia è l'insieme di pagine "attualmente" riferite. Questo significa che l'insieme restituito dalla funzione WS sarà costituito dal al più Δ elementi.

Esempio Abbiamo $R_\alpha : 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6$. Il $WS = (10, 5)$ sarà dato da $\{1, 2, 5, 6\}$. Notare come nel tempo l'insieme può aumentare o diminuire: se avessimo calcolato $WS = (8, 5)$, l'insieme sarebbe stato di cardinalità 5 a $t = 8$, mentre è diminuita a $t = 10$.

Esempio Abbiamo $R_\alpha : 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 4, 7, 3, 2, 1, 2, 3, 6$. Applichiamo il modello del WS con $\Delta = 4$ per esprimere le sostituzioni di pagina. Facciamo attenzione a togliere le pagine man mano che escono dal working set, e segnando PF quando bisogna (ri)mettercele:

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	1	1	1	2	1	1	1	1	1	1	1	1	2	3	2	1	1	1	1
	2	2	2	3	2	2	2	2	2	2	2	2	3	6	3	2	2	2	2
		3	3	4	3	4	5	5	5	6	3	3	6	7	6	3	3	3	3
			4		4	5	6	6	6			7	7		7	6			6
PF	PF	PF	PF		PF	PF	PF				PF	PF	PF		PF	PF			PF

Considerazioni: Il WS dà il meglio di sè con la sostituzione locale dei riferimenti di un processo, non con una sostituzione globale dei processi, in quanto, come scopriremo prossimamente, in quanto un processo, durante l'esecuzione, migra da una località all'altra.

WSClock

Applichiamo quindi l'algoritmo del clock che abbiamo già visto per il WS. Manteniamo un contatore T per ogni processo e fissiamo una finestra temporale τ per calcolare il nostro possibile WS.

- abbiamo le pagine organizzate in una lista, questa la scorriamo ogni volta che applichiamo il WSClock[3];
- ogni entry contiene i reference (R) e dirty (M) bit e un registro *Time of Last Use* (TLU), che viene copiato dal contatore durante l'algoritmo. La differenza tra questo registro e il contatore si chiama età della pagina.
- ad un page fault, si guarda prima la pagina indicata dal puntatore:
  - se $R = 1$, si mette $R = 0$, si copia $TLU = T$ e si passa avanti;
  - se $R = 0$ e età $\leq \tau$, è nel working set: si passa avanti;
  - se $R = 0$ e età $> \tau$, se $M = 0$ allora si libera la pagina, altrimenti si schedula un pageout e si passa avanti;
  - ma se facciamo un giro completo?
    - se almeno un pageout è stato schedulato, si continua a girare (aspettando che le pagine schedulate vengano salvate, alternativa è segnarsi quella più vecchia e swapparla come col TVC);
    - altrimenti, significa che tutte le pagine sono nel working set. Soluzione semplice: si rimpiazza una quasisiasi pagina pulita. Se non ci sono neanche pagine pulite, si rimpiazza la pagina corrente.

[3] l'operatore di resto ci da una grande mano!