

Sistemi Operativi: vmbo

Ferruccio Vitale

Contenuto della relazione

Specifica del problema	3
Analisi del problema	6
Progettazione dell'algoritmo.....	7
Descrizione moduli.....	13
Memory Management Unit (MMU).....	13
Processo.....	15
Dispositivo di I/O	18
Prima fase: analisi dei parametri	19
Seconda fase: inizializzazione ed esecuzione thread	20
Terza fase: stampa dei risultati.....	20
Testing del programma	21
Test 1. Parametri di configurazione dei moduli.....	21
Test 2. Simulazione carico eccessivo	22
Test 3. Analisi file di log di un processo	22
Test 4. Inizializzazione processi con probabilità differenti.....	23
Test 5. Evitare che un processo effettui un tipo di operazione	24
Test 6. Uso della "Reference String"	24
Test 7. Anomalia di Belady	25

Specifica del problema

Scrivere un programma multithread che consenta di valutare le performance, in termini di numero di *page fault*, di algoritmi di rimpiazzamento delle pagine per la gestione della memoria virtuale.

Il programma dovrà essere costituito da un'entità principale che operi come una Memory Management Unit (MMU), un numero arbitrario (n) di thread, ove ogni thread emuli un singolo processo (PROCESSO) e, infine, da un'entità che emuli un dispositivo di I/O (DISPOSITIVO I/O).

Il programma dovrà simulare una sessione di lavoro, nella quale sono presenti n processi che possono accedere alla memoria e generare richieste di I/O. I processi dovranno generare indirizzi di memoria casuali. Infine, il programma terminerà una volta raggiunto il numero prestabilito di accessi in memoria totali.

MMU

L'entità MMU sarà la base del sistema ed utilizzerà l'algoritmo di rimpiazzamento delle pagine oggetto di test. Essa dovrà essere istanziata dal thread principale (*main*) e sarà gestita come oggetto condiviso tra i diversi thread dei PROCESSI. In particolare essa definirà la dimensione dello spazio di indirizzamento virtuale, la dimensione della memoria fisica disponibile e la dimensione della pagina.

La MMU dovrà, inoltre, contenere una lista di tabelle della pagine ciascuna relativa ad un PROCESSO. La definizione della specifica delle tabelle delle pagine sarà lasciata allo studente, e naturalmente dipenderà dal particolare algoritmo di rimpiazzamento implementato.

I PROCESSI dovranno generare indirizzi verso la MMU invocandone un metodo pubblico.

L'accesso a tale metodo dovrà garantire la mutua esclusione (un solo PROCESSO alla volta potrà invocare il suddetto metodo. La mutua esclusione dovrà essere ottenuta utilizzando le primitive di sincronizzazione messe a disposizione dal linguaggio di programmazione eccezion fatta per il costrutto *monitor* (per chiarezza, ad esempio, in java non sarà possibile utilizzare il costrutto *synchronized*).

Il metodo in questione processerà la richiesta andando ad interrogare l'opportuna tabella delle pagine. Nel caso in cui, l'indirizzo generato sia relativo ad una pagina già mappata in memoria la MMU si limiterà ad incrementare un contatore statistico di *page hit*. D'altra parte, nel caso in cui l'indirizzo generato dal PROCESSO faccia riferimento ad un indirizzo contenuto in una pagina non mappata in memoria la MMU dovrà eseguire i seguenti passi:

- incrementare un contatore di *page fault*

- selezionare una pagina mappata da rimuovere (vittima) secondo la politica dell'algoritmo oggetto di test.
- aggiornare le opportune tabelle delle pagine
- indicare come non valida la pagina che è appena stata rimossa.
- inserire una nuova entry nella tabella delle pagine del PROCESSO che ha
- generato il *page fault* (la pagina è stata appena caricata).

PROCESSO

L'entità PROCESSO dovrà emulare un processo in esecuzione su di un sistema multi programmato. Ogni processo dovrà essere emulato attraverso l'utilizzo di un singolo thread. Lo scheduling dei thread sarà delegato alle librerie messe a disposizione dal linguaggio di programmazione (configurazione default).

Il comportamento di ogni PROCESSO può essere descritto da una macchina a stati finiti comprendente due soli stati mutuamente esclusivi:

- Richiesta indirizzo di memoria (invocazione del metodo pubblico della MMU).
- Richiesta operazione di I/O (invocazione del metodo pubblico del DISPOSITIVO di I/O).

Le transizioni tra i due stati dovranno essere gestite in maniera probabilistica. In particolare, dovrà essere definita una probabilità p (input fornito al momento del lancio dell'applicazione) di generare indirizzi di memoria ("entrare nello stato a dato che ci si trovava nello stato b" = "rimanere nello stato a") e una probabilità $q = 1-p$ di generare richieste di I/O ("entrare nello stato b dato che ci si trovava nello stato a" = "rimanere nello stato b")

I PROCESSI dovranno prevedere la possibilità di essere inizializzati con diversi valori di probabilità p .

DISPOSITIVO di I/O

L'entità DISPOSITIVO di I/O dovrà essere gestita come un ulteriore thread. Questa avrà un metodo pubblico attraverso il quale i PROCESSI potranno effettuare una richiesta di I/O.

Le richieste di I/O dovranno essere gestite con un coda di tipo FIFO. Le richieste saranno, ovviamente, bloccanti, mettendo il PROCESSO chiamante in uno stato di attesa fino al soddisfacimento della richiesta.

Il DISPOSITIVO di I/O determinerà il tempo di servizio di ciascuna richiesta in modo casuale estraendo a caso un valore intero, espresso in

millisecondi, compreso fra T_{min} e T_{max} (parametri di inizializzazione del DISPOSITIVO di I/O).

Dati di INPUT

Il programma dovrà ricevere in input i seguenti parametri di inizializzazione:

- Numero di PROCESSI
- Probabilità p di generare accessi in memoria (una p per ogni PROCESSO; la probabilità di accedere al dispositivo di I/O sarà pari a $1-p$)
- Tempi di servizio minimi e massimi del DISPOSITIVO di I/O (T_{min} e T_{max}).
- Numero totale di accessi in memoria raggiunto il quale il programma dovrà terminare.

Dati di OUTPUT

Al termine della simulazione il programma dovrà stampare a video la percentuale di *page fault* totale (relativa a tutti i processi), la percentuale di *page fault* relativa a ciascun PROCESSO, il tempo medio di servizio del DISPOSITIVO di I/O ed il tempo medio di attesa per una richiesta di I/O relativo a ciascun PROCESSO.

Ciascun PROCESSO dovrà scrivere in un proprio file di log la successione di indirizzi generata e le chiamate al dispositivo di I/O.

Analisi del problema

Il problema richiede di realizzare un simulatore elementare di memoria virtuale, utile all'analisi del comportamento e relative prestazioni di un algoritmo di rimpiazzo delle pagine. Il sistema realizzato simulerà un ambiente di lavoro multi programmato, nel quale operano più processi contemporaneamente.

Un'analisi iniziale ci suggerisce che questo problema sia di tipo **decidibile**, ovvero presuppone l'esistenza di un algoritmo che lo risolva in tempo finito e per una qualunque istanza di dati, espressa in termini di numero di processi contemporanei, dimensione della pagina e così via.

Come da specifica, il simulatore non avrà bisogno di alcun dato di ingresso per funzionare. Un insieme esteso di parametri da riga di comando permetterà di alterare alcuni aspetti e funzionalità del simulatore.

L'output del programma sarà costituito da alcuni dati statistici, prodotti a seguito del numero prestabilito di accessi alla memoria.

Progettazione dell'algoritmo

La scelta della struttura dati e del relativo algoritmo tiene conto dei parametri dettati dalla specifica del problema: l'attenzione sarà dunque focalizzata sulla politica di rimpiazzo delle pagine, sulle strutture dati necessarie ed sullo scenario che tali scelte determinano; l'analisi non terrà quindi in considerazione l'impatto che il *paging* nella memoria secondaria possa avere sull'efficienza complessiva del sistema, né dei tempi d'accesso alla memoria o di caricamento di una pagina.

Si è scelto di articolare il programma in tre fasi distinte:

1. analisi dei parametri forniti su riga di comando: di particolare rilievo, la possibilità di modificare alcuni parametri di funzionamento del simulatore;
2. inizializzazione delle strutture dati e successiva esecuzione dei singoli thread;
3. attesa del completamento dei thread, relativa deallocazione delle strutture dati e successiva stampa delle statistiche.

DESCRIZIONE DEL SIMULATORE

Un sistema dotato del meccanismo di memoria virtuale consente ad un processo di allocare una quantità di memoria superiore a quella effettivamente disponibile. La tecnica si basa sull'utilizzo di altri tipi di memoria, detta *secondaria*, che entrano in gioco quando la memoria fisica risulta piena. Il meccanismo di memoria virtuale implementato per il simulatore è quello della **paginazione**.

La paginazione prevede la suddivisione della memoria fisica in *frame*, tutti della stessa dimensione; analogamente, anche la memoria allocata da un processo viene divisa in *pagine*, la cui dimensione è pari a quella di un frame di memoria fisica: tale suddivisione avviene ad opera della MMU, in modo totalmente trasparente al processo utente.

Al pari di un sistema operativo, il simulatore terrà aggiornata una tabella dei processi attivi: ogni voce nella tabella è rappresentativa di un processo e delle pagine virtuali allocate.

Per semplicità, si suppone che la tabella dei processi e la relativa page table siano sempre residenti in memoria, ovvero che queste non vengano mai paginate a loro volta.

Perché un processo possa accedere ad un'informazione contenuta in una determinata pagina virtuale, è necessario che quest'ultima sia presente nella memoria fisica ovvero, nello specifico, che sia associata ad un frame. Quando la lista dei frame disponibili non è esaurita, sarà sufficiente associare la pagina richiesta ad un frame disponibile; nel caso contrario, si

dovrà applicare una politica di rimpiazzo delle pagine, volta a selezionare una pagina tra quelle presenti in memoria, spostarla nella memoria secondaria e associare il frame appena liberato alla pagina richiesta.

L'algoritmo di rimpiazzo delle pagine scelto è l'**enhanced second chance**, noto anche come *algoritmo dell'orologio*: tale soluzione nasce come una naturale evoluzione della tecnica FIFO¹, superando il problema di rimuovere le pagine usate più di frequente.

La realizzazione di tale algoritmo impone che ogni pagina contenuta nella *page table* del processo, oltre le comuni informazioni, debba contenere due ulteriori bit:

- bit *reference* (R): verrà posto al valore uno (1) quando si accederà alla pagina in questione, sia in scrittura che lettura: tale informazione permetterà di sapere se ad una pagina presente in memoria si sia acceduto di recente o meno;
- bit *dirty* (D): verrà posto ad uno (1) quando si accederà alla pagina in scrittura: quando una pagina dovrà essere rimossa dalla memoria, il valore uno di questo bit suggerirà al sistema operativo di fare una copia della pagina nella memoria secondaria, prima della rimozione dalla memoria principale.

La ricerca della pagina virtuale da rimuovere dovrà tenere conto delle quattro possibili combinazioni ottenute con questi due bit:

R	D	DESCRIZIONE	AZIONE
0	0	la pagina non è stata usata di recente e non è stata modificata	rimpiazza la pagina
0	1	la pagina non è stata usata di recente ma è stata modificata	effettua un write back della pagina ² e imposta ad zero il bit D
1	0	la pagina è stata usata di recente ma non è stata modificata	imposta a zero il bit R
1	1	la pagina è stata usata di recente ed è stata modificata	imposta a zero il bit R ed effettua il write back della pagina

La ricerca della pagina migliore non solo si traduce nella selezione della pagina non usata di recente, ma anche di quella che non sia stata modificata. In questo modo, le pagine utilizzate più di frequente hanno alta probabilità di rimanere nella memoria principale, a discapito delle pagine a cui si è acceduto meno frequentemente³.

¹ La tecnica FIFO sposta nella memoria secondaria le pagine più *vecchie*, ovvero quelle allocate per prima, anche quando queste vengono utilizzate frequentemente.

² La pagina di memoria viene bloccata, ovvero resa inutilizzabile, fintanto che l'operazione di I/O non termini.

³ L'algoritmo *enhanced second chance* è un'approssimazione del LRU.

Quando una pagina presente in memoria risulta *sporca*, si dovranno quindi effettuare due operazioni di I/O: il primo accesso alla memoria secondaria sarà necessario per effettuare il *page out*, ovvero copiare la pagina *sporca* sul disco, mentre una seconda operazione di lettura sarà necessaria per il *page in* della pagina che ha generato il *fault*. In questo scenario, l'algoritmo scelto, analizzando periodicamente le pagine che risultano *sporche* e provvedendo ad un preventivo *write back* nella memoria secondaria, migliora sensibilmente le prestazioni complessive, nonché il tempo medio d'accesso alla memoria.

Tale algoritmo, tuttavia, ha lo svantaggio di partizionare l'insieme delle pagine in due classi: quelle usate di recente e quelle usate con minor frequenza; quando termina la ricerca per il rimpiazzo, non è detto che la pagina identificata sia la più vecchia in assoluto, ma semplicemente una delle pagine meno utilizzate di recente.

Il caso migliore si verifica quando il primo frame analizzato ha i bit R e D posti a zero: la ricerca termina immediatamente e non sarà nemmeno necessaria un'operazione di I/O volta a fare una copia della pagina nella memoria secondaria.

Come da requisito, i processi effettuano soltanto l'operazione di lettura della memoria: è tuttavia possibile alterare questo comportamento specificando il parametro “-w” da riga di comando, per ottenere una simulazione che prevede l'uso del bit *DIRTY*.

Il caso peggiore dell'*enhanced second chance* si verifica quando tutte le pagine residenti in memoria sono sia referenziate che modificate: in questo caso, l'MMU dovrà necessariamente scorrere interamente la lista delle pagine presenti, ottenendo la medesima complessità dell'algoritmo FIFO.

Da tali considerazioni, è facilmente comprensibile che la complessità dell'algoritmo sia $O(n)$, ovvero lineare e proporzionale al numero di frame in cui è stata suddivisa la memoria.

Non meno importante è sottolineare che, essendo un'evoluzione della tecnica FIFO, tale approccio soffre dell'anomalia di Belady (si veda a tal proposito il test 7).

Nei sistemi moderni, alcune funzionalità utili all'implementazione di tale algoritmo vengono implementate direttamente dall'hardware, quale, ad esempio, l'impostazione ad uno del bit *reference* e *dirty* ogni qualvolta una pagina venga referenziata o modificata.

Un algoritmo di rimpiazzo può essere di tipo *locale* o *globale*: il simulatore proposto effettua una ricerca **globale**, ovvero quando avviene un *page fault*, l'MMU cercherà la pagina candidata alla rimozione dalla memoria tra le pagine di tutti i processi attivi e non limitandosi alle pagine allocate del processo corrente. Sebbene una ricerca locale prevenga influenze da parte degli altri processi in termini di *page fault*, un algoritmo globale risulta complessivamente più efficiente nonché più semplice da implementare.

La modalità di generazione degli indirizzi, ad opera dei processi utente, è in linea con il **principio di località**, secondo cui se la CPU sta accedendo ad uno specifico dato (o istruzione), con molta probabilità i prossimi dati (o istruzioni) saranno ubicati nelle vicinanze di quella in corso: valutare tale principio ed i suoi impatti è di fondamentale importanza per il funzionamento e le prestazioni della memoria virtuale gerarchica dei moderni sistemi.

È stato, in ultimo, scelto un approccio di tipo **pure demand paging**, il quale prevede di non assegnare alcun frame al processo fintanto che la pagina non venga referenziata.

I vantaggi di tale tecnica sono immediatamente percepibili:

- le pagine inutilizzate non vengono mai caricate in memoria, aumentando la quantità di memoria disponibile ad altri processi;
- all'avvio di un programma, viene ridotta la latenza necessaria a caricare le pagine in memoria;
- minor carico di I/O dovuto ad un minor numero di pagine da caricare.

In presenza di un sistema con poca memoria fisica disponibile, il *demand paging* assicura un grado più elevato di concorrenza di processi.

Per semplicità, si è assunto che le pagine contenenti le istruzioni del processo siano sempre residenti in memoria: soltanto le pagine contenenti dati verranno gestite dalla MMU e dalla suddetta politica di rimpiazzo.

Il sistema emulato farà uso di indirizzi a 20 bit⁴: da un punto di vista fisico, un simile spazio di indirizzamento consente di gestire una memoria fisica composta, al massimo, da 1,048,576 byte: il simulatore sarà appunto dotato di questo quantitativo, sebbene sia possibile alterare tale aspetto con l'opportuno parametro da riga di comando (-R).

Anche i processi utente utilizzano un indirizzamento a 20 bit; lo spazio d'indirizzamento virtuale sarà ovviamente superiore a quello fisico in quanto si dovrà considerare anche la memoria secondaria. Sono dunque possibili due scenari che, in assenza del meccanismo di memoria virtuale, non potrebbero sussistere:

- più processi contemporanei, ognuno dei quali alloca il massimo della memoria fisica: la somma di tutti gli spazi d'indirizzamento risulta maggiore della memoria fisica;
- un sistema dotato di una quantità ridotta di memoria fisica ma che consente ugualmente ad un processo di allocare 1Mb di memoria virtuale.

⁴ È tuttavia possibile simulare un sistema con indirizzamento fino a 32 bit.

Un indirizzo virtuale a 20 bit generato da un processo sarà suddiviso, in modo del tutto trasparente, in due parti distinte: i otto bit più significativi (da 19° al 12°) verranno usati come indice per consultare la *page table* del processo, mentre i restanti dodici bit rappresenteranno l'offset da sommare all'indirizzo di partenza del frame associato.



Una simile suddivisione consente ad ogni processo di allocare un massimo di 256 pagine virtuali (2^8), la cui dimensione è pari a 4,096 byte (2^{12}): ovviamente il numero di pagine virtuali disponibili per un processo è indipendente dal numero di frame disponibili.

Qualora il rapporto tra i frame disponibili ed il numero massimo di processi concorrenti risulti vantaggioso, il programma attiverà automaticamente la **paginazione anticipata**: l'MMU, infatti, non si limiterà a caricare in memoria la pagina mancante, causa del *fault*, ma anche un ristretto numero di pagine adiacenti⁵. In pratica il simulatore cerca di prevedere quali pagine saranno richieste in virtù della località dei dati e le carica prima di quando queste vengano effettivamente richieste.

La suddetta funzionalità è da considerarsi puramente dimostrativa ed embrionale.

Di seguito vengono riassunte le scelte progettuali che caratterizzano il simulatore.

HARDWARE	
Dimensione indirizzo	20 bit
Dimensione RAM	1,048,576 byte
Dimensione frame	4,096
MEMORIA VIRTUALE	
Tipo	paginata
Politica di rimpiazzo	enhanced second chance
Dimensione pagina	4,096
Page table	sempre in memoria
Max pagine/processo	256

⁵ Nel nostro caso, la pagina precedente e quella successiva.

DISPOSITIVO I/O	
Tmin	1
Tmax	100
PROCESSO	
Probabilità accesso	80%
Località temporale	30%
Località spaziale	<i>Loop su un vettore</i>

Descrizione moduli

Di seguito verranno descritti, con maggiore dettaglio, gli aspetti implementativi dei singoli moduli: l'MMU, il dispositivo di I/O ed il processo.

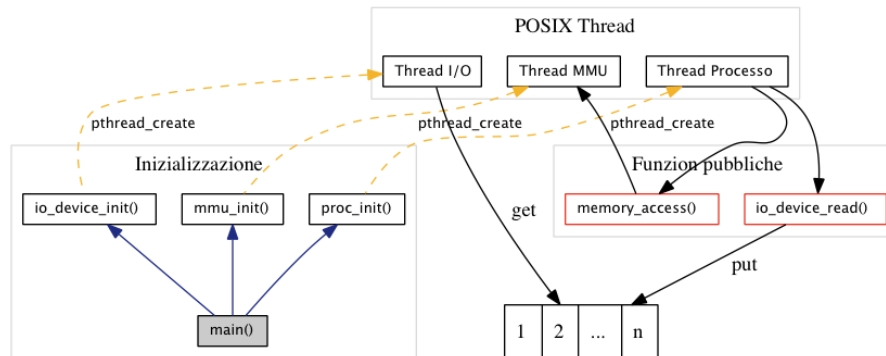


Figura 1. Moduli e loro relazione

Ogni singolo modulo del simulatore viene inizializzato direttamente dal *main()* attraverso le rispettive funzioni e sulla base dei parametri specificati da riga di comando. Terminata questa fase, inizierà l'interazione tra i singoli thread.

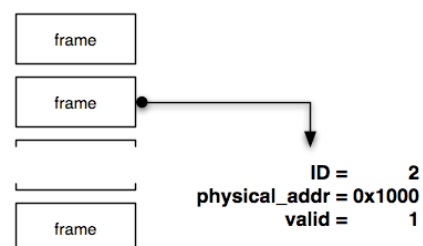
MEMORY MANAGEMENT UNIT (MMU)

Il thread che emula l'MMU è il primo ad essere istanziato dal simulatore, direttamente all'interno del *main*: questo si occupa della traduzione di un indirizzo virtuale, generato da un processo, in un equivalente indirizzo fisico di memoria.

La funzione *mmu_init* è incaricata di configurare l'ambiente del simulatore nonché di partizionare la memoria fisica in frame. Ad ogni frame verranno associate le seguenti informazioni:

- **id**: tale valore identifica univocamente un frame;
- **physical_addr**: questa variabile conterrà l'indirizzo di memoria fisica di partenza del frame; durante il ciclo di vita del simulatore, la lista dei frame non sarà ordinata per *id*: questo è il valore di base cui sommare l'offset dell'indirizzo virtuale;
- **valid**: il bit verrà posto ad uno (1) quando il frame è utilizzato;

Ad ogni frame vengono inoltre associate due ulteriori campi, *pid* e *page_id*: sebbene non siano necessari al funzionamento del simulatore, hanno il solo scopo di rendere più semplice il *debug*, nonché offrono la possibilità di mantenere



un'associazione frame \rightarrow processo \rightarrow pagina⁶.

L'MMU fa uso di due liste semplici: la prima, *free_frames*, per tenere traccia dei frame liberi (ovvero non associati ad alcuna pagina) ed *used_frames* per i frame utilizzati. Terminata l'inizializzazione della memoria, tutti i frame saranno presenti nella lista *free_frames*, condizione dettata dal *demand paging*. Un'ulteriore lista, *active_pages*, conterrà solo le pagine presenti in memoria.

La funzione *thread_mmu* viene eseguita come thread e simula l'MMU: l'interazione tra questo thread ed i processi è resa possibile dalla funzione *memory_read*, invocata direttamente dai processi che tentano un accesso alla memoria⁷. Quest'ultima si occuperà di inserire in una struttura temporanea i dati relativi al processo chiamante, nonché l'indirizzo virtuale richiesto e segnalerà alla MMU la presenza di una richiesta; l'MMU, nel vagliare la richiesta, sceglierà quale frame associare alla pagina equivalente ed, al termine, restituirà l'indirizzo di memoria fisico.

Di seguito verrà illustrato lo schema a blocchi del funzionamento della MMU.

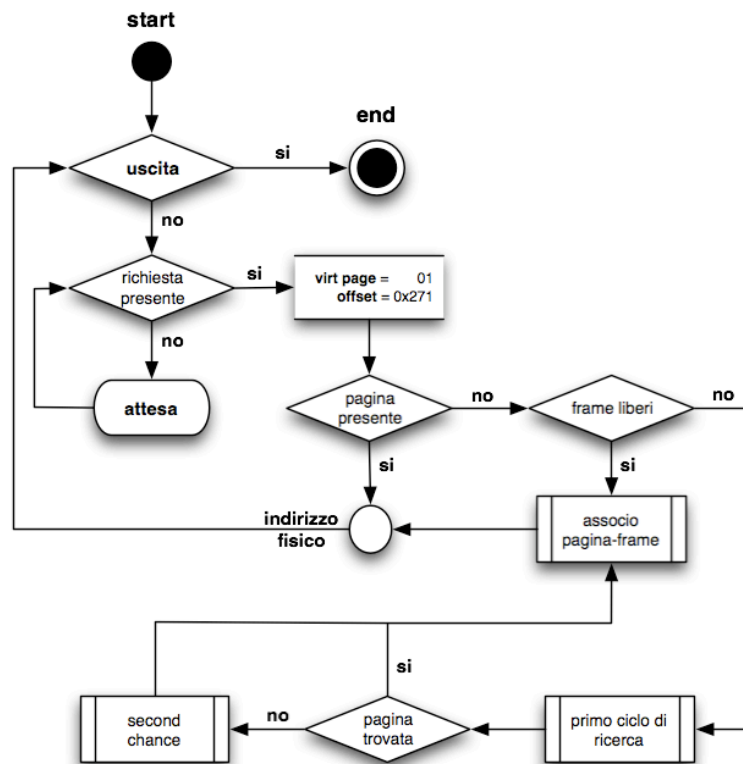


Figura 2. Diagramma a blocchi del thread MMU

L'algoritmo *enhanced second chance* viene applicato solo quando la pagina richiesta non è presente in memoria ed, al contempo, tutti i frame risultano occupati. Ha quindi inizio una ricerca della pagina candidata alla

⁶ La *page table* di un processo mantiene invece l'associazione processo-pagina-frame.

⁷ Come da specifica, la funzione *memory_read* può essere invocata da un processo per volta: il mutex *mem_read_lock* garantisce la mutua esclusione dei chiamanti.

rimozione: i due bit precedentemente descritti, *reference* e *dirty*, tengono traccia dell'uso e della modifica delle pagine.

L'algoritmo prevede che l'elenco delle pagine utilizzate sia rappresentato da una lista circolare, mentre è stata usata una lista doppiamente concatenata che, nel caso peggiore, viene visitata per intero.

PROCESSO

La funzione `thread_proc`, eseguita come thread, si occupa di simulare un processo utente.

La rappresentazione in memoria del singolo processo, nonché delle caratteristiche che lo contraddistinguono, è affidata alla struttura `proc`. Avendo come riferimento un sistema operativo reale, al proprio interno, il simulatore gestisce una lista di tutti i processi attivi, detta *process table*: ogni voce in questa tabella è di tipo `proc` e contiene le seguenti informazioni:

- **PID**: identificativo univoco del processo;
- **TID**: identificativo univoco del thread che simula il processo;
- **PAGE_COUNT**: numero di pagine virtuali allocate dal processo;
- **PAGE_TABLE**: tabella delle pagine del processo; questa informazione non deve essere accessibile al processo;
- **PERCENTILE**: probabilità del processo di effettuare un accesso alla memoria piuttosto che al dispositivo di I/O;
- **IO_COND/IO_LOCK**: condizione e relativo mutex nel quale il thread resta in attesa di completamento di una richiesta di I/O;
- **LOG_FILE**: file di log del processo;
- **PROC_STATS**: struttura per contenere le statistiche d'accesso del processo.

Nei sistemi reali, un processo ha visibilità solo di alcune informazioni contenute nella *process table* o relative al sistema ospite: per esempio, un processo potrà sapere qual è il proprio PID o la dimensione della memoria virtuale, ma non potrà tuttavia accedere alla lista delle pagine o dei frame, informazioni ad unico appannaggio del sistema operativo. Seguendo le implementazioni dei più comuni sistemi *NIX si è dunque scelto di non far gestire alla MMU la tabella delle pagine virtuali del processo, garantendo opportunamente la protezione dello spazio d'indirizzamento: l'MMU farà piuttosto uso di una propria struttura dati per gestire l'associazione delle pagine ai frame. Questa scelta trova ulteriore giustificazione nel fatto che ogni processo può allocare una quantità variabile di memoria e l'MMU, che viene inizializzata prima della *process table*, non può conoscere a priori tale quantità.

Poiché il simulatore non è stato implementato con un linguaggio orientato ad oggetti quale Java o C++, le API di accesso alla memoria ed al dispositivo di I/O devono contenere necessariamente l'identificativo del richiedente, nello specifico il PID.

La funzione `proc_init` ha il compito di creare n istanze del thread processo, per ognuno dei quali associa un numero casuale di pagine virtuali.

Il numero casuale di pagine virtuali è compreso tra 1 e 256, in modo tale che tutti i processi abbiano almeno una pagina associata e comunque non superiore al limite massimo di 256. È tuttavia possibile massimizzare tale valore per tutti i processi usando il parametro $-M$.

La funzione `thread_proc`, si occupa di scegliere se effettuare un accesso alla memoria o al dispositivo di I/O, scegliendo tra le due operazioni con una probabilità impostata dall'utente: quando la funzione `memory_access`, utilizzata per accedere alla memoria, restituirà il valore -1, i processi termineranno la propria esecuzione.

L'accesso in memoria osserva il principio di località, secondo cui:

- se accedo ad un dato, probabilmente accederò anche a quelli vicini in un tempo non lontano (**località spaziale**);
- se accedo ad un dato è probabile che debba accedervi nuovamente in breve tempo (**località temporale**).

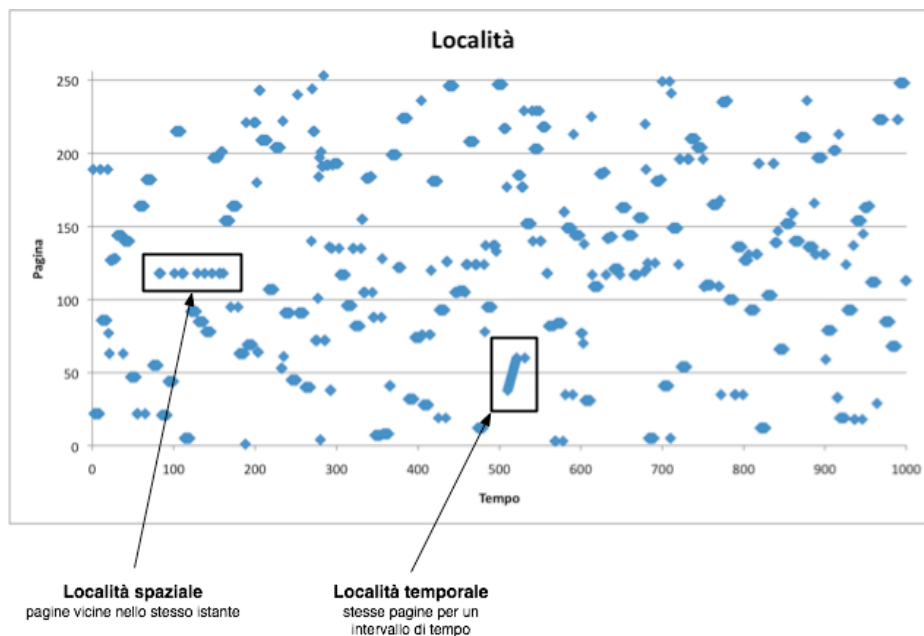


Figura 3. Campionamento località

La realizzazione della località spaziale viene offerta dalla funzione `simulate_loop` che si occupa di simulare un accesso ad un vettore di n elementi: poiché in C gli elementi di un vettore occupano indirizzi di memoria contigui, simulare un accesso ad ognuno di essi coincide ad una località spaziale.

Quando il processo effettua invece un accesso pseudo casuale alla memoria, subentra la località temporale, ovvero la possibilità di accedere ad un dato recentemente utilizzato. In quest'ottica, il processo avrà il 30% di probabilità di accedere ad un indirizzo di memoria usato in precedenza.

Il parametro “-L” permette di variare la percentuale di località temporale, di default pari a 30%; ponendo uguale a zero tale valore, si elimina l'effetto di località.

L'operazione di lettura merita ulteriori considerazioni:

1. nel caso di accesso alla memoria, l'indirizzo virtuale è generato casualmente, ma è sempre compreso nello spazio d'indirizzamento virtuale del processo: qualora l'indirizzo fuoriuscisse da tale spazio, il simulatore avrebbe il compito di generare un fault e terminare il processo;
2. per una migliore simulazione dell'algoritmo scelto, che prevede l'uso del bit *dirty*, è stata implementata l'operazione di scrittura in una zona di memoria: la funzione che realizza tale operazione è sempre `memory_access` ma con il parametro `rw` posto ad uno; non sarà tuttavia presente un parametro utile a rappresentare l'informazione da scrivere in memoria, in quanto esula lo scopo della presente analisi;
3. per entrambe le tipologie di accesso, alla memoria od al dispositivo di I/O, il processo non ha visibilità diretta dei rispettivi thread: la possibilità di effettuare una richiesta viene offerta dall'uso di due funzioni pubbliche⁸ che svolgono il ruolo di intermediario.

Sussiste una sottile differenza tra i due tipi di accesso che un processo utente può effettuare: l'MMU rappresenta un oggetto condiviso tra più processi e può assolvere una richiesta per volta; se n processi tentassero l'accesso alla memoria, soltanto uno riuscirà nell'intento mentre gli altri “ $n-1$ ” processi restano in attesa (mutua esclusione).

Di contro, l'accesso al dispositivo di I/O prevede che effettuare una richiesta d'accesso non sia bloccante, per cui se n processi tentassero l'accesso al dispositivo, tutti quanti riuscirebbero ad inserire la propria richiesta in coda: l'attesa avviene immediatamente dopo, non bloccando quindi gli altri attori della contesa, ma solo il processo che attende il risultato.

⁸ Avendo realizzato il simulatore con un linguaggio non orientato ad oggetti, una funzione è considerata *pubblica* se non è stata definita come *static* ed il prototipo è disponibile agli altri file che compongono il progetto.

Di seguito viene illustrato il diagramma a blocchi del funzionamento del thread processo.

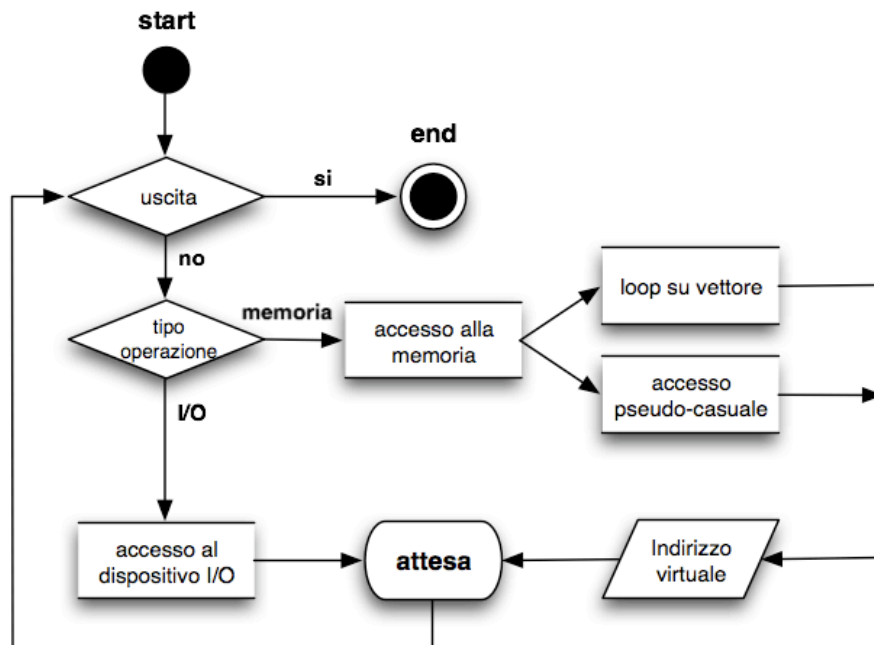


Figura 4. Diagramma a blocchi del thread processo

La prima condizione, *uscita*, viene soddisfatta quando la MMU ha effettuato il numero richiesto di accessi.

DISPOSITIVO DI I/O

Il dispositivo di I/O viene emulato dalla funzione `thread_io_device`, anch'essa eseguita come thread. Questo viene inizializzato ed istanziato immediatamente dopo aver creato il sottostrato di memoria virtuale.

Il funzionamento del dispositivo di I/O, a differenza della MMU, si basa su una lista di tipo FIFO: i processi che effettuano una richiesta a tale dispositivo vengono inseriti in coda e serviti in modo sequenziale. La richiesta è di tipo bloccante: un processo, in attesa che la richiesta venga soddisfatta, non può effettuare altre operazioni.

La funzione `io_device_read` rappresenta il metodo pubblico per inserire una richiesta in coda, al pari di `memory_access` per le letture in memoria.

La coda di richieste di I/O costituisce una risorsa condivisa tra il thread (*lettore*) e la funzione `io_device_read` (*scrittore*) ed, in tal senso, l'accesso alla FIFO va gestito all'interno di una sezione critica: il mutex `fifo_lock` viene utilizzato dalle due entità al fine di mantenere coerente lo stato delle richieste.

Il thread terminerà la propria esecuzione su richiesta dell'MMU, quando questa avrà espletato il numero di richieste previsto.

Di seguito viene illustrato il diagramma a blocchi del dispositivo.

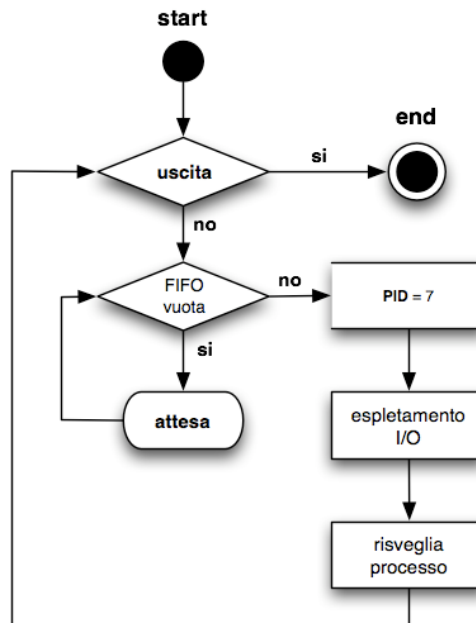


Figura 5. Diagramma a blocchi del thread I/O

Come da specifica, il dispositivo viene configurato con due parametri, T_{MIN} e T_{MAX} , rispettivamente il tempo minimo e massimo perché una richiesta di lettura venga effettuata; il thread, una volta esaminata la richiesta, si occuperà di generare un numero casuale compreso in tale intervallo ed infine segnalerà al processo l'avvenuta operazione: il processo sarà nuovamente libero di effettuare altre operazioni con le modalità precedentemente descritte.

PRIMA FASE: ANALISI DEI PARAMETRI

Il programma inizierà la propria esecuzione analizzando i parametri specificati su riga di comando; sebbene questi non siano necessari per il funzionamento del simulatore, è disponibile un insieme di parametri per modificare alcuni comportamenti predefiniti dell'applicativo.

Di seguito vengono riassunti i possibili parametri.

VALORE	FUNZIONE	DEFAULT
-v --version	Stampa la versione del programma ed esce	<i>disattivato</i>
-d --debug	Aumenta il livello di debug dell'output	
-a --anticipatory-paging	Disabilita la paginazione anticipata	
-h --help	Stampa la sinossi del programma ed esce	
-l <i>lista</i> --probabilities= <i>lista</i>	Specifica la probabilità d'accesso alla memoria per i processi	
-L <i>num</i> --locality= <i>num</i>	Specifica la località temporale	30%
-M --all-memory	Forza i processi ad allocare il massimo della memoria	<i>no</i>
-m <i>num</i> --max-read= <i>num</i>	Imposta il numero massimo di accessi alla memoria	50

<code>-p num</code>	Imposta il numero massimo di	5
<code>--max-processes=num</code>	processi concorrenti	
<code>-P num</code>	Imposta la probabilità con cui	80%
<code>--probability=num</code>	un processo effettua un accesso alla memoria	
<code>-r lista</code>	Imposta la <i>reference string</i> per	
<code>--reference=lista</code>	determinare gli accessi	
<code>-R num</code>	Imposta la dimensione della	1,048,576
<code>--ram-size=num</code>	RAM disponibile	
<code>-s num</code>	Imposta la dimensione della	4,096
<code>--frame-size=num</code>	pagina e del frame	
<code>-t num</code>	Imposta il parametro Tmin del	1
<code>--Tmin=num</code>	dispositivo di I/O	
<code>-T num</code>	Imposta il parametro Tmax del	100
<code>--Tmax=num</code>	dispositivo di I/O	
<code>-w</code>	Consente ai programmi di	no
<code>--write-enabled</code>	accedere alla memoria anche in scrittura	

SECONDA FASE: INIZIALIZZAZIONE ED ESECUZIONE THREAD

La corretta analisi dei parametri specificati su riga di comando permette di passare alla inizializzazione delle singole strutture dati ed all'esecuzione dei thread finora descritti.

La prima entità istanziata è appunto l'MMU, seguita successivamente dalla creazione del thread per il dispositivo di I/O ed in ultimo dai thread che simulano i processi utente.

TERZA FASE: STAMPA DEI RISULTATI

La terza fase ha inizio quando i processi utente hanno effettuato il numero massimo di accessi alla memoria: questa condizione determina la fine di tutti i thread istanziati, siano essi utente, MMU o del dispositivo di I/O.

Una corretta conclusione prevede quindi di effettuare una `pthread_join` di tutti quanti i thread e la relativa deallocazione delle strutture dati fino a quel momento utilizzate.

Al termine di questa operazione, il simulatore si dovrà preoccupare di stampare a video le statistiche richieste. Oltre le suddette statistiche, saranno inoltre disponibili, nella directory di lavoro, i file di log di ogni singolo processo, all'interno dei quali si potrà trovare la lista delle operazioni effettuate dai processi utente.

Testing del programma

Il *testing* del programma prevede, in prima istanza, l'analisi dell'output prodotto dal simulatore e la sua corretta interpretazione.

TEST 1. PARAMETRI DI CONFIGURAZIONE DEI MODULI

```
unixo$ ./vmbo
--> Simulatore inizializzato con indirizzi a 20 bit
--> Thread MMU avviato
    [RAM=1048576, PAGESIZE=4096, PHYS-FRAMES=256, TOTAL_READ=50, PROC=5]
--> Thread DEVICE I/O avviato [Tmin=1, Tmax=100]
--> Thread PROC avviati [NUM=5, PROB=80%, OPER=R, LOCALITY=30%]
<-- Thread MMU terminato
<-- Thread DEVICE I/O terminato
```

S T A T I S T I C H E								
PID	NUM	PROB	ACCESSI	PAGE	FAULT	ACCESSI	TEMPO	
	PAG		MEMORIA	FAULT	(%)	I/O	MEDIO	
0	92	80%	7	4	57%	2	24	
1	89	80%	2	2	100%	1	20	
2	60	80%	9	2	22%	1	55	
3	82	80%	20	6	30%	1	53	
4	235	80%	12	2	17%	1	5	
			50	16	32%	6	30	

```
Pagine virtuali allocate =          558
Memoria virtuale allocata =    2285568 (~ 2.2 Mb)
```

All'avvio del programma, i singoli moduli stampano su `stdout` il messaggio di avvenuta inizializzazione, specificando i valori con cui sono stati configurati; in assenza di parametri su riga di comando, il funzionamento del simulatore viene assicurato dai parametri di default.

L'output del simulatore è di tipo "buffered": a seguito di successive esecuzioni del programma, è possibile notare un ordine differente dei primi messaggi d'inizializzazione stampati a video.

In fase di uscita, il simulatore provvede a stampare le seguenti statistiche:

- *page fault* totali;
- tempo medio di servizio del dispositivo di I/O;
- per ogni processo:
 - numero di *page fault*
 - numero di accessi I/O e tempo medio d'attesa.

Vengono stampate due ulteriori informazioni, utili alla valutazione del carico reale che il simulatore ha sostenuto: il totale di pagine virtuali allocate da tutti i processi e l'equivalente dimensione della memoria virtuale allocata, a fronte di quella fisica disponibile.

TEST 2. SIMULAZIONE CARICO ECCESSIVO

```
unixo$ ./vmbo -w --Tmax=2 --memory-read=10000 --processes=1000 -M
--> Simulatore inizializzato con indirizzi a 20 bit
--> Thread MMU avviato
    [RAM=1048576, PAGESIZE=4096, PHYS-FRAMES=256, TOTAL_READ=10000,
    PROC=1000]
--> Thread DEVICE I/O avviato [Tmin=1, Tmax=2]
--> Thread PROC avviati [NUM=1000, PROB=80%, OPER=RW, LOCALITY=30%]
<-- Thread MMU terminato
<-- Thread DEVICE I/O terminato
```

S T A T I S T I C H E								
PID	NUM	PROB	ACCESSI	PAGE	FAULT	ACCESSI	TEMPO	
	PAG		MEMORIA	FAULT	(%)	I/O	MEDIO	
0	256	80%	22	22	100%	1	2	
1	256	80%	8	5	62%	1	2	
[...]								
999	256	80%	11	11	100%	1	2	
			10000	9746	97%	884	2	

```
Pagine virtuali allocate = 256000
Memoria virtuale allocata = 1048576000 (~ 1000.0 Mb)
```

In questo caso, è stato aumentato il numero di processi concorrenti (*--processes=1000*): rispetto al precedente esempio, l'effetto immediato è che, a parità di numero di frame disponibili, aumentano gli attori che tentano l'accesso alla risorsa condivisa. Ai processi in esecuzione è stato inoltre richiesto di allocare la dimensione massima di memoria (-M), ovvero 256 pagine virtuali, per un totale di 1Mb; il numero elevato di richieste d'accesso (*--memory-read=10000*) a così poche risorse porta inevitabilmente ad un elevato numero di *page fault* (97%).

TEST 3. ANALISI FILE DI LOG DI UN PROCESSO

Come da specifica, il simulatore si occuperà di creare un file di log per ogni thread *processo*.

Lo stralcio di file di log del processo mostra nel dettaglio le operazioni effettuate che, nel caso specifico, possono essere di lettura/scrittura alla memoria (-w) o di accesso al dispositivo di I/O.

```
NIZIO PROCESSO
=====
PID = 0
PAGINE VIRTUALI = 109
PROBABILITA' = 80%
=====

Scrittura indirizzo virtuale 431302 [pagina 105 - offset 1222]
--> La pagina virtuale 105 e' stata associata al frame 18
[PAGE FAULT] L'indirizzo virtuale 431302 corrisponde al fisico 74950

Lettura indirizzo virtuale 431322 [pagina 105 - offset 1242]
[PAGE HIT] L'indirizzo virtuale 431322 corrisponde al fisico 74970

Scrittura indirizzo virtuale 731766 [pagina 178 - offset 2678]
<-- La pagina 149 del processo 856 è stata rimossa dalla memoria (frame 249)
--> La pagina virtuale 178 e' stata associata al frame 249
[PAGE FAULT] L'indirizzo virtuale 731766 corrisponde al fisico 1022582
Write-back della pagina 178
```

```
Lettura indirizzo virtuale 5259 [pagina 1 - offset 1163]
L'indirizzo virtuale 5259 corrisponde al fisico 1163
```

```
Richiesta d'accesso a dispositivo I/O accodata
Richiesta d'accesso servita in 49 ms
```

Nel caso di un accesso alla memoria, nel file di log comparirà l'indirizzo virtuale generato ed il suo equivalente indirizzo fisico, tradotto dalla MMU. Qualora la richiesta abbia generato un *page fault*, verrà anche descritta l'operazione di associazione della pagina richiesta ad un frame, secondo le regole descritte.

Qualora sia attiva la paginazione anticipata, sarà possibile notare che, a fronte di un singolo *page fault*, verranno caricate tre pagine in memoria, ovvero due in più rispetto al necessario.

La richiesta di I/O prevede due stati: l'inserimento della richiesta nella FIFO del relativo dispositivo ed il tempo necessario perché questa venga esaudita.

Qualora venga specificato il parametro *-debug* o *-d* su riga di comando, all'interno del file di log viene stampato, dopo ogni accesso in memoria, lo stato delle pagine del processo, specificando se queste siano associate ad un frame, nonché le informazioni sui bit *reference* e *dirty*. Di seguito se ne riporta uno stralcio.

```
Scrittura indirizzo virtuale 22429 [pagina 5 - offset 1949]
--> La pagina virtuale 5 e' stata associata al frame 9
--> La pagina virtuale 4 e' stata associata al frame 10
--> La pagina virtuale 6 e' stata associata al frame 11
[PAGE FAULT] L'indirizzo virtuale 22429 corrisponde al fisico 38813
    PAGE 0 :
    PAGE 1 :
    PAGE 2 :
    PAGE 3 :
    PAGE 4 : FRAME 10 [REF]
    PAGE 5 : FRAME 9 [REF, DIRTY]
    PAGE 6 : FRAME 11 [REF]
    PAGE 7 :
    PAGE 8 :
    PAGE 9 :
    PAGE 10 :
    PAGE 11 :
    PAGE 12 :
    PAGE 13 :
```

TEST 4. INIZIALIZZAZIONE PROCESSI CON PROBABILITÀ DIFFERENTI

L'uso del parametro “-l” permette di specificare una probabilità diversa per ogni processo; come si potrà notare dall'esempio, non sarà necessario specificare tanti valori per quanti processi verranno istanziati: qualora la lista non contenga valori a sufficienza, i restanti processi ereditano una probabilità pari a 80%.

```
unixo$ ./vmbo -w -l .1:.2:.3
--> Simulatore inizializzato con indirizzi a 20 bit
--> Thread MMU avviato
    [RAM=1048576, PAGESIZE=4096, PHYS-FRAMES=256, TOTAL_READ=50, PROC=5]
--> Thread DEVICE I/O avviato [Tmin=1, Tmax=100]
--> Thread PROC avviati [NUM=5, PROB=0%, OPER=RW, LOCALITY=30%]
<-- Thread MMU terminato
```

```
<-- Thread DEVICE I/O terminato
```

S T A T I S T I C H E							
PID	NUM	PROB	ACCESSI	PAGE	FAULT	ACCESSI	TEMPO
	PAG		MEMORIA	FAULT	(%)	I/O	MEDIO
0	91	10%	9	2	22%	4	56
1	115	20%	0	0	0%	4	44
2	39	30%	2	2	100%	4	44
3	64	80%	23	6	26%	3	27
4	35	80%	16	4	25%	3	63
			50	14	28%	18	47

```
Pagine virtuali allocate = 344
Memoria virtuale allocata = 1409024 (~ 1.3 Mb)
```

TEST 5. EVITARE CHE UN PROCESSO EFFETTUI UN TIPO DI OPERAZIONE

Ricorrendo ancora una volta al parametro “-l” per impostare la probabilità di effettuare un accesso in memoria, sarà possibile evitare che un processo effettui un particolare tipo di operazione; come si potrà osservare dall’esempio, specificando il valore 0, il PID 0 e 1 non effettuano accessi in memoria ma solo al dispositivo di I/O e viceversa per il PID 3.

```
unixo$ ./vmbo -l 0:0:1
--> Simulatore inizializzato con indirizzi a 20 bit
--> Thread MMU avviato
    [RAM=1048576, PAGESIZE=4096, PHYS-FRAMES=256, TOTAL_READ=50, PROC=5]
--> Thread DEVICE I/O avviato [Tmin=1, Tmax=100]
--> Thread PROC avviati [NUM=5, PROB=0%, OPER=R, LOCALITY=30%]
<-- Thread MMU terminato
<-- Thread DEVICE I/O terminato
```

S T A T I S T I C H E							
PID	NUM	PROB	ACCESSI	PAGE	FAULT	ACCESSI	TEMPO
	PAG		MEMORIA	FAULT	(%)	I/O	MEDIO
0	177	0%	0	0	0%	1	39
1	249	0%	0	0	0%	0	0
2	66	100%	25	6	24%	0	0
3	14	80%	1	1	100%	0	0
4	256	80%	24	3	12%	0	0
			50	10	20%	1	39

```
Pagine virtuali allocate = 762
Memoria virtuale allocata = 3121152 (~ 3.0 Mb)
```

TEST 6. USO DELLA “REFERENCE STRING”

Ricorrere all’uso della *reference string* per misurare il numero di *page fault* equivale ad utilizzare una metrica che viene influenzata da fattori esterni, quale lo scheduler o dagli accessi al dispositivo di I/O.

Quando viene specificata una *reference string*, il simulatore altera il proprio comportamento ed imposterà in modo autonomo alcuni valori prestabiliti:

OPZIONE	VALORE
Numero processi	1
Accessi in scrittura	no
Accessi I/O	disabilitato
Probabilità accesso	100% memoria
Numero accessi memoria	proporzionale alla reference string

Di seguito si riporta l'esecuzione del simulatore con la seguente *reference string*: 1 2 3 4 1 2 5 1 2 3 4 5.

```

unixo$ ./vmbo --reference=1:2:3:4:1:2:5:1:2:3:4:5 --ram-size=16768 -d
--> Simulatore inizializzato con indirizzi a 20 bit
--> Thread MMU avviato
    [RAM=16768, PAGESIZE=4096, PHYS-FRAMES=4, TOTAL_READ=12, PROC=1]
--> Thread DEVICE I/O avviato [Tmin=1, Tmax=100]
--> Thread PROC avviati [NUM=1, PROB=100%, OPER=R, LOCALITY=30%]
<-- Thread DEVICE I/O terminato
<-- Thread MMU terminato

```

S T A T I S T I C H E							
PID	NUM	PROB	ACCESSI	PAGE	FAULT	ACCESSI	TEMPO
	PAG		MEMORIA	FAULT	(%)	I/O	MEDIO
0	12	100%	12	10	83%	0	0
			12	10	83%	0	0

```

Pagine virtuali allocate =      12
Memoria virtuale allocata = 49152 (~ 0.0 Mb)

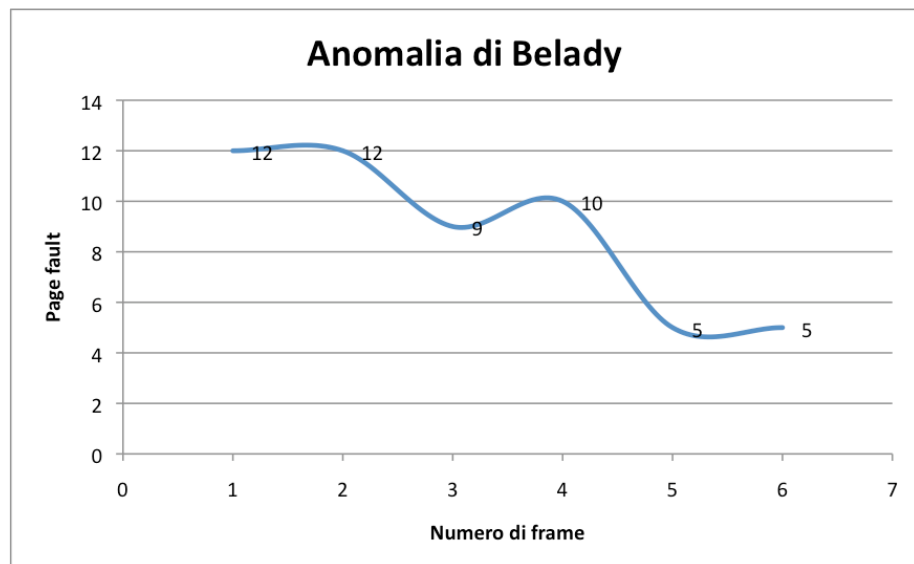
```

TEST 7. ANOMALIA DI BELADY

Per verificare che l'algoritmo proposto soffra dell'anomalia di Belady, si è scelto di effettuare alcuni campionamenti del numero di *page fault* generati al variare della dimensione della memoria fisica, mantenendo costante la dimensione del frame.

La metrica utilizzata durante i campionamenti è la *reference string*, in quanto immune da influenze esterne; di seguito si riportano i parametri specificati su riga di comando:

PARAMETRO	EFFETTO
--reference=1:2:3:4:1:2:5:1:2:3:4:5	Sequenza di pagine
--ram-size=<num>	Valore compreso tra 23Kb e 4Kb



Analizzando il grafico si può notare come, in corrispondenza di quattro frame disponibili, il numero di *page fault* aumenti rispetto a quelli ottenuti con la medesima *reference string* e tre frame.