Reti e sistemi operativi
A.A. 2013/2014

19 Maggio 2014

Homework 3 (facoltativo)

- Consegna: entro **3 Giugno 2014 ore 24**
- Formato: **archivio (.zip oppure .tar.gz)** che contiene *una directory* al cui interno devono trovarsi: tutti i file sorgenti (.c e .h) ed un file .txt con le istruzioni su come compilare ed eseguire il software (specificare qui nome, cognome e numero di matricola).
- Si richiede: **codice funzionante e ben commentato**.

Esercizio 1

Implementare in C una versione semplificata di un gestore di memoria virtuale.

Il sistema dovrà simulare un sistema hardware/software dotato di memoria centrale di $size_{mem} = 256$ byte, memoria di massa utilizzabile per le operazione di swapping di $size_{disk} = n * size_{mem}$ byte, ove n numero di processi in esecuzione nel sistema, e dimensione delle pagine di memoria virtuale (ovviamente identica ai frame di memoria fisica) di $size_{page} = 4$ byte. Per simulare i processi che eseguono gli accessi in memoria, creare n thread, ognuno definito da un ID univoco (**process_id** $= 0, \ldots, n-1$) che eseguono una sequenza di accessi alla memoria. Ognuno dei thread abbia un spazio di indirizzamento virtuale di dimensione pari alla memoria centrale ($size_{mem}$), ovvero ogni thread ha la sensazione di avere a disposizione l'intera memoria, a partire dall'indirizzo 0 sino all'indirizzo 255. I thread potranno eseguire 2 semplici operazioni in memoria: (a) allocare e inizializzare *tutta* la memoria virtuale a loro disposizione e (b) leggere un byte alla volta. Tali operazioni andranno effettuate chiamando le funzioni:

- **void allocateAllMemory(int process_id)**: alloca $size_{mem}$ byte di memoria virtuale per il processo con identificativo **process_id**, inizializzando ogni byte con un valore corrispondente al suo indirizzo virtuale.
- **char readByte(int process_id, int virtual_address)**: legge un byte all'indirizzo virtuale **virtual_address**, restituisce il byte letto.

Memoria centrale e disco andranno simulati attraverso due buffer (array) condivisi tra tutti i thread di dimensioni $size_{mem}$ e $size_{disk}$ byte, rispettivamente. I thread accederanno a tali buffer solamente attraverso le funzioni **allocateAllMemory()** e **readByte()**. Ad ogni thread andrà associato una page table che conterrà per ogni pagina:

- Indice del frame in memoria centrale (valido solo se bit di validità è settato a 1).
- Bit di validità: se 1, la pagina è presente in memoria centrale, se 0 la pagina ha subito uno swap out.

Il sistema dovrlò inoltre disporre di una *frame table* globale che conterà per ogni frame di memoria centrale:

- ID del processo che sta usando quel frame di memoria centrale, oppure -1 se non ancora usato da nessun processo.
- Indice nella page table corrispondente (nota: questa implementazione non permette di condividere frame tra più processi).
- Istante dell'ultimo accesso a tale frame in memoria centrale, oppure 0 se non vi è stato alcun accesso.

Il sistema utilizza la paginazione su richiesta (la pagina viene portata in memoria solo se richiesta) utilizzando una politica di allocazione globale, ovvero i frame sono tutti a disposizione per ogni processo. Per semplicità si supponga inoltre che:

- Non sia implementato il bit di modifica, per cui prima di ogni operazione di swap in sarà sempre necessario eseguire un'operazione di swap out se il frame non è libero.
- Ogni pagina verrà salvata a disco sempre nella stessa posizione, esclusiva e dipendente dall'indice della pagina e dall'identificativo del processo (in fin dei conti si è detto che ogni processo ha memoria virtuale di dimensione $size_{mem}$ byte mentre la porzione di disco utilizzata per lo swapping ha dimensione $size_{disk} = n * size_{mem}$).

In pratica, **allocateAllMemory()** dovrà:

- Settare tutti i bit di validità della page table corrispondente a 0.
- Riempire tutti i frame corrispondenti in memoria secondaria (ovvero, a disco) inizializzando ogni byte con un valore corrispondente al suo indirizzo virtuale

mentre **readByte()** dovrà:

- Estrarre dall'indirizzo **virtual_address** l'indice i di pagina
- Consultare l'i-esima entry della page table associata al thread di identificativo **process_id**.

- Se il bit di validità è settato a 1, si avrà un page hit: si utilizzerà l'indice del frame sommato all'offset per accedere al dato, aggiornando l'istante di accesso nella corrispondente entry in frame table.
- Se il bit di validità è settato a 0, si avrà un page miss: in tal caso si dovrà scegliere in memoria centrale un frame da sostituire. Consultando la frame table, si cerchi un frame libero (ovvero ID del processo che sta usando quel frame di memoria centrale settato a -1). Se non è presente alcun frame libero, si utilizzi l'algoritmo LRU per la scelta (ovvero viene scelto il frame con istante di accesso minore), effettuando lo swap out a disco del frame scelto e resettando il corrispondente bit di validità nella page table del precedente proprietario del frame. Si effettui quindi lo swap in da disco a memoria, aggiornando la entry in frame table (ID del processo, indice nella page table corrispondente e istante di accesso) e la entry nella page table. Infine, si acceda al dato, come nel caso precedente.

readByte() dovà inoltre incrementare due contatori globali: **n_access** per ogni chiamata e **n_page_miss** per ogni page miss. Per testare il sistema, si facciano partire n thread, ognuno dei quali dovrà implementare il seguente frammento di codice (in pseudo-codice):

```
allocateAllMemory( process_id );
for (round = 0; round < m; round++)
{
  for(i = 0; i < size_mem; i++)
  {
    if( i != readByte( process_id, i ))
    {
      // ERRRORE
      return -1;
    }
    usleep(1000);
  }
}
```

Utilizzare, ad esempio, $n = 4, 16, 32, \ldots$ e $m = 10$ (entrambi devono essere dei parametri modificabili). Al termine, si stampi a video il contenuto dei contatori **n_access** e **n_page_miss**.

ATTENZIONE: gli accessi alle risorse condivise andranno protette con un **mutex lock**, evitando race condition.

Appendice

Per ottenere l'istante corrente dal clock di sistema, utilizzare la seguente funzione:

```
#include <stdlib.h>
#include <sys/time.h>

unsigned long int getTime()
{
```

```
  struct timeval tv;
  gettimeofday( &tv, NULL );
  return (unsigned long int)tv.tv_sec*1000 + (unsigned long int)tv.tv_usec/1000;
}
```

Se pensate che manchino delle specifiche al problema, o se avete dei dubbi, contattatemi via mail: pretto@dis.uniroma1.it