LRU ALGORITHM IMPLEMENTATION

Technical Overview

José Benavente

INTRODUCTION

Operative systems utilise paging for virtual memory management. This system allows for processes to move in and out of memory staying organised in pages. These events can be represented by a scheme where pages are a set of tracks, and processes enter them one by one every time unit.

When we start keeping track of which processes want access to virtual memory every page is initialised as free, which means processes can be assigned to them. Whenever a process is assigned to a page it's called a page fault. If a process which has already been assigned to a page requests access to memory again it's simply granted access and it keeps on executing, this time with no fault. If all pages have processes assigned to them and a new, different process wants to access memory, then an algorithm is used to determine which process will be paged-out (or swapped-out) to make room for the new process. When the new process replaces an old one, a fault is registered.

This program emulates memory management implementing the LRU (Least Recently Used) page replacement algorithm. The implementation is rather simple and aims to be easy to read and understand, as it is a project I made for an Operative Systems class, which I decided to adapt and simplify as much as possible.

PROJECT STRUCTURE

The project structure is very simple. The following file structure represents a valid environment for this program to function on Windows 64 bits:

LRU/
├── logs/
├── libiconv-2.dll
├── libunistring-2.dll
└── LRU.exe

The *logs/* directory is where the files containing the steps for each execution will be stored at. Both dlls present within the project directory are used for UTF-8 checks and procedures directly tied to logging.

Simply execute *LRU.exe* and the program will run normally.

IMPLEMENTATION

The LRU algorithm implementation works as follows. The virtual memory, paging and process tracking is all wrapped into a RAM object. Several macros are defined to set the bounds which will apply to the memory environment.

The following macros can be found at the *obj_RAM.h* header file.

```
#define UNSET_VAL -1
#define REQUESTS 20
#define FRAMES 4
```

These values can be tweaked to evaluate different scenarios. **REQUESTS** represents the number of processes expected by this memory instance. **FRAMES** would be the number of page frames available in this memory. **UNSET_VAL** is internal and represents every memory spot which has not been filled by a process.

As for the relevant data regarding the logic present in *obj_RAM.c*, it all comes wrapped up in the RAM structure:

```
typedef struct {
        FILE *logFile;
        int requests[REQUESTS]
        int contents[FRAMES][REQUESTS]
        int lruMatrix[FRAMES[FRAMES]
        int faults;
} RAM;
```

This structure stores an array of requests (integer values), a matrix to represent the memory and its pages, the LRU matrix used to calculate page replacements, a faults counter, and an associated log file to register each view of the memory when requests are processed.

At runtime, this is the series of procedures carried out to instantiate the RAM object and process values with it:

```
…
RAM * ram = malloc(sizeof(RAM));
InitRAM(ram);
ProcessRAMValues(ram, sample);
…
```

After a RAM object has been allocated in memory and initialised, an array of values is passed to it and assigned to its **requests** field. These requests will automatically be processed and upon completion the program will generate a log file within its *logs/* directory. The user must make sure said directory exists beforehand.

There is a sample value set present in *main.c*. The code was left like this on purpose to allow anybody to expand on it and implement value input on their own. You may also just modify this array of integers to change the processes.