Michael Liao
CSC 280
Spring 2022

# Memory Management Lab Report

## Compiling:

No special libraries are required to compile this C program. All you need to do is use the GNU compiler command as follows:

```
gcc pagemanager.c -o pagemanager
```

## Running:

Ensure that a text file named `data.txt` exists in the same directory where this program has been compiled, with the formatting as indicated in the directions. Then, simply run it by typing:

```
./pagemanager
```

## Summary:

In order to create simulations for multiple memory page management algorithms, I had to first understand how the data relating to the pages would be formatted so I could read it in. That was addressed with a function called `read_file()` which would take an array of structs of type `page` to fill in with repeated calls of `fscanf()` accessing a file named `data.txt`, mentioned earlier (generated using a separate program which randomly generated page references and frames available); the function also returned a value representing the actual number of pages referenced so the simulations could run efficiently and not iterate through a potentially empty array. A struct `page` has only two integers associated with it: `page_num` and `time_accessed`. These should be fairly self-explanatory; they allow the program to allocate the page number to the array and then store the `time_accessed` for future algorithms' needs.

Each type of algorithm was given its own function for organization's sake (e.g. `sim_optimal()` or `sim_fifo()`) and called in the order given in the directions for this assignment. I had to write an additional function called `reset_data()` (called at the beginning of every simulation) in order to reset the `time_accessed` properties after a function has modified them in order to ensure no data was mixed or corrupted between simulations.

Each function makes good use of for loops in order to iterate through the array of pages and operates by incrementing one of many index variables `i` to move along the list of page references. It checks the current page reference indicated by that variable against a separate array `frames` (of type struct `page`, naturally)containing only the pages currently in use by the computer. If `frames` contains the page reference, there is no need to cause a page fault, and the simulation moves on, incrementing `i`. Otherwise, a page fault must occur, and the index variable `k` is used to track which frame is currently being modified as the current page reference replaces it. A variable called `page_faults` is incremented whenever this occurs so we can track our relevant data. Other for loops are used for different algorithms to analyze the current page references in our `frames` array and target them more specifically for page faults (using a `longest` and `oldest` variable for the Optimal algorithm and the Least Recently Used (LRU) algorithms respectively and comparing the calculated values against other page references in use), though the simplest algorithm — First In First Out (FIFO) — required only management of the index variable `k` to ensure it never moved out of bounds. Once the entire list is processed, by checking the index `i` against the number of pages (found in the `read_file()` function and passed by pointer to the simulation functions), it can then print the results and exit, allowing the program to move onto the next simulation.

**Results & Analysis:**

With a dataset of 2000 page references and 17 frames available, these were the results:

```
optimal page faults: 420
fifo page faults: 934
lru page faults: 924
```

With these numbers, it is clear that the Optimal algorithm performed the best. It does exactly what it says on the tin with just under half the page faults of the other two algorithms, even with multiple trials of varying available frames.

The FIFO and LRU algorithms performed about the same across multiple trials, although generally speaking the LRU algorithm was able to perform slightly better than FIFO, beating it by anywhere from about 10 to 20 page faults.

To speak of the elephant in the room, the optimal algorithm is functionally impossible to implement in real circumstances. It requires a neatly ordered list of pages which extend into the future, and even computers aren't able to prophesize yet. Although it certainly is the best option, it is unfeasible to expect a computer to

queue up a list of all its page references before submitting it to its memory management system so it can work on it optimally, as the time saved by doing so will certainly not outweigh the time required to delay all its memory-related operations.

As a result, I would advocate then for computers to use the LRU algorithm in order to optimize that extra page fault advantage over the FIFO algorithm. It may not perform as well as the optimal algorithm, but it does well for what it is.

Investing further time into optimizing this process would be an endeavor for those who have quite a bit of leisure time. At the current stage of memory management, as far as I and other ordinary people are concerned, it works well enough that I may not notice the difference between LRU and FIFO. Should an algorithm achieve that benchmark by speeding up to the point where the slowdown is generally unnoticeable, I believe that it is sufficient for most everyday uses. In some rare or extreme cases, it may be important to optimize even further, but that is a task that will take much more effort and resources to research and put into practice.

**Shortcomings:**

This conclusion being reached, this simulation is not perfect, as is the case with many things. The range of page numbers for references affects the data greatly, just as the random integer I generated for the available page frames would affect how many page faults are required. A standardization of such would help ensure the data's consistency across multiple trials; for example, a fixed 32 page frames available and only ever 64 programs running at any given time. That being said, the data can still be compared according to relative scales between amounts of page faults, and I believe my conclusions are not too devalued by this shortcoming.