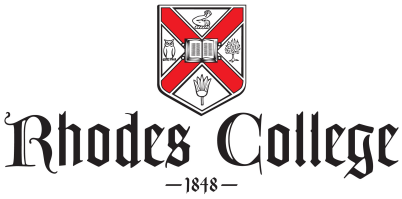
# COMP 330
# Lab 4 - Page Replacement




This project entails writing a C program to implement a virtual memory simulator. This assignment can be retrieved from GitHub Classroom to get the starter code. Submit your final program via `git`. Your work will be tested and graded on `cslogin.arc.rhodes.edu`.

## Paging System

In this laboratory assignment, you will be simulating a virtual memory paging system. Your program will be given a list of 10 processes that occupy different sizes in main memory. You will also have another input file that contains a sequence of memory accesses into the virtual memory address space. Your program will maintain a page table for each process to map virtual addresses to physical addresses.

When your program begins, you will allocate a fixed number of frames for each process. As you process the memory access trace, you will simulate the eviction and replacement process using one of the algorithms discussed in Chapter 10. The number of frames allocated to a process may change if the eviction algorithm is global, otherwise the number of frames per-process is fixed.

Your program must accept the following parameters on the command-line in the following order:

1. size of main memory in bytes
2. page size in bytes (must divide main memory size evenly)
3. page allocation algorithm (0 = equal number of pages per process, 1 = proportional to process size)
4. page eviction algorithm (0 = FIFO, 1 = Second Chance, 2 = LRU, 3 = LFU)
5. global vs. local eviction (0 = global, 1 = local)
6. period (the number of memory accesses in between printing the page tables to an output file, 0 = no printing)

## Simulating Virtual Memory

There are input files (small and large) in the GitHub Classroom repository.

`plist.txt` – This file contains a list of processes (0-9) and their memory footprint in bytes. The max size of a process is 1024 bytes. The first line contains the number of processes. The number of pages required by a process depends on the page size, passed as an input. For example, if the page size is 4 bytes and the process size is 502 bytes, then the process will consume 126 pages (2 bytes wasted on the last page due to internal fragmentation). If the main memory size is 256 bytes, then there are only 64 frames in physical memory.

`ptrace.txt` – This file contains a large number of memory accesses. Each line contains the process number followed by the virtual address. If the requested byte is not loaded in main memory, a page fault occurs. Your simulator must evict pages based on the specified strategy, bring in the page and update the page table accordingly.

Your program will not be tested with the provided input files, but they will follow the same format.

# Paging System

## Page Tables

You should maintain a page table for each process that maps virtual pages to physical frames in main memory. Each entry may need to include the following information. Depending on your implementation, you may choose to include additional information for each page table entry.

1. in-memory / on-disk bit (present/absent bit)
2. reference bit
3. frame address
4. time stamp or age

## Page Frame Allocation

As the trace file is processed, pages must be brought into main memory. For a given process, pages can be brought in without eviction until the process runs out of frames allocated to it. The number of frames allocated to a process is done with one of two techniques:

1. option 0: allocate an equal number of frames for each process
2. option 1: allocate an number of frames proportional to the process' size

**NOTE: Do not assign specific frames to processes ahead of time**. You should allow frames to be assigned as requests are made in the trace. Using this scheme, fill up the frames in memory sequentially until it is full. No complicated data structure is needed to maintain free space. After the allocated frames run out for each process, evict frames belonging to the process if local eviction is selected. If global eviction is selected, then frames in other processes are fair game. A process may run out of its allocated frames before main memory is full.

## Page Eviction

When a page containing the requested memory location is not currently loaded, a page fault occurs. It is safe to assume that the memory trace never contains an invalid address. Your program must bring the requested page into main memory. Once all frames allocated to a process are exhausted, eviction is required.

Your implementation must implement four algorithms for page eviction: *First-In, First-Out (FIFO), Second Chance* variant of FIFO, *Least Recently Used*, and *Least Frequently Used*.

1. FIFO: when a page is to be evicted, it evicts the oldest page
2. Second Chance: for this variant of FIFO, we use a *reference bit* and only replace a page if the bit is zero. You should remember where the last eviction occurred and start from there for the subsequent eviction. You should reset all reference bits every 100 memory accesses.
3. LRU: conceptually, this algorithm maintains a list representing the order in which pages have been accessed. The list consists of pointers to page table entries. The list maintains a list of pages ordered from least recently referenced to most recently referenced. On *every* reference, search the list for the page table entry for the referenced page and place the list entry at the end of the list. When eviction is needed, evict from the front of the list.

   This algorithm can be implemented without a list by recording the last reference time for each in-memory frame. When evicting, look for the timestamp with the smallest value of in-memory frames.
4. LFU: This algorithm approximates LRU. A counter is maintained for each page and is incremented on each access. When a page is to be evicted, the page with the smallest counter value is evicted. If multiple pages have the same counter value, then the page table entry's timestamp should be used to determine the oldest. Both the timestamp and counter for a page table entry should be reset when a new page is loaded into the frame.

**Global vs. Local Eviction**

The eviction algorithms listed above can either operate globally or locally. When a process needs to load a page, but there are no free frames it must evict. If the local eviction scheme is selected, then only pages in the current process are considered for eviction. Under global eviction, a page from any other process' page table may be evicted. The number of frames per process is fixed under local eviction, but may vary under global eviction.

## Experimentation and Analysis

You should run your program and verify that it works with different permutations of the program parameters. A subset of the parameters will be used to test the correctness of your solution. In particular, your program should output the following information:

1. Print the page fault rate for each process ($100\times$ # of page faults / accesses)
2. Print the total number of page faults vs. accesses and the page fault percentage
3. Average page fault percentage for all processes

Your program must print out to a file `ptable.txt`, the contents of all page table entries after every $N$ memory accesses. Print one entry per-line, with each table separated by a blank line. Separate each period with line containing five hyphens. You should overwrite this file every run.

There is a reference implementation in `/home/comp330/lab4` on `cslogin`. You may use this implementation to test your code against.

## Requirements

This program must be written in C and run on Linux. You may develop on your own computer if you prefer, but it will be graded on `cslogin`. You may use multiple files, but your main program should be in `lab4.c`. There is starter code in the Github Classroom repository. This code handles some of the drudgery of reading the input files. You will need to create your own data structures and functions to implement the remainder of the lab. The starter code will require modifications to work properly.

Submit your work by committing and pushing your final submission through Git.