Advanced Computer Architecture Term Project Report

**Comparing the Performance of Memory Replacement Policies**

Group 5 / 201922004 권남호, 201922037 장용주, 202052006 정시우, 20202203 황선태

# Description of the Project

We have modeled the memory management structure including translation, paging, and swapping among computer architectural concepts and compared the performance of replacement policies.

Our memory model provides 32MB of virtual address space and has a 4KB page size. Therefore, it has 8192 pages, and the allocated pages are located in physical memory or swap file. The physical address space is 8MB, and 8MB of memory is allocated through POSIX malloc at the beginning of the program. For a page to be referenced, the page must be loaded in physical memory. In addition, a 32MB swap file is created, and when a page fault occurs, the page existing in the physical memory is evicted and located there. For simplicity of implementation, the location of the page in the swap file is fixedly determined by the virtual page number of the page. In order to translate virtual addresses into physical addresses, a page table was made and managed. Each page table entry has bits for valid, present, dirty and data for allocation size and physical addresses. If the page to be used is already allocated and on the physical memory, it is recorded as a page hit, otherwise it is recorded as a page fault. On page fault, if there is an empty physical memory frame, it is loaded to an empty frame and used. If there is no empty physical memory frame, an eviction occurs. At this time, when the dirty bit is 1, the file write is done to the swap file. In the code run by the user, there are four operations: malloc, free, set, get. For the workload that runs this code, you can write these 4 codes. Enter the number of mallocs, the number of free, and the number of operations first. Memory size is allocated randomly as many times as malloc set, and free also randomly frees memory addresses. If you want to write memory to the virtual address, type it as S [VA] [CHAR]. If you want to read it, write your workload like G [VA]. When the workload is finished, the number of page hits, page faults, file reads, and file writes are displayed.

The replacement policies we implemented are FIFO, clock, second chance (FIFO-based), pseudo-LRU, LRU, and NRU. The FIFO is managed as a queue, and victims are selected in the order of access. The clock policy is similar to FIFO but implemented as a circular queue instead of a queue. And the clock pointer exists, and it gives a second chance to entry through the reference bit. Note that the clock and the second chance policy give another chance to survive for the page, but these two have different mechanisms and structures to search for victims. The second-chance policy is based on FIFO queues, and a reference bit is added to the queue entry as like on the clock policy. When enqueued, this bit is set to 1. When this bit is 1 when searching for a victim from the head to the tail of the queue, it is set to 0 to provide one chance, and if it is 0, the page is selected as the victim. Instead of creating a tree structure, pseudo-LRU is implemented to create a flat array and traverse it like a tree through indexes that access it. LRU is implemented to be managed in a doubly linked list structure, and it is pushed to the tail when accessing memory, and the victim is always the entry of the head of the list. The NRU manages the modified bit and the reference bit for the page, and decides the victim according to these four priorities. File IO can be avoided as much as possible by reducing the number of times it is swapped to disk through the modified bit.

The workload used in the experiment is a workload that allocates 32MB of all memory space, fills all memory spaces with sets in alphabetical order repeatedly, and then repeats set and get in random memory space.

# Analyzing the Results

We first measured how long it took to run the same workload to measure the overhead for each policy. In the order of the longest execution time, it was NRU, FIFO, pseudo-LRU, Clock, Second chance, and LRU. It seems that this is basically due to the different complexity of the algorithm. It was tested on i9-9900X CPU, but your milage may vary.

And we compared the number of hits, and it came out as 67100647, 67100647, 67100647, 67101286, 67100647, 67100647 for FIFO, clock, second chance, PLRU, LRU, and NRU for the same workload. For a given workload, you can see that the policy giving the highest hit rate is pseudo-LRU. We can see that pseudo-LRU, which is less strict algorithm with a little randomness, shows higher performance because there are no regular memory access patterns inside the workload.

# Raw result

[page hits] [page faults] [file reads] [file writes] / Execution time

* FIFO  
    / 
* Clock  
   / 
* Second chance  
   / 
* PLRU  
   / 
* LRU  
   / 
* NRU  
   / 

# Running Manual

Codes are written for each policy. Therefore, each c file can be used after compilation. For compiling LRU policy, it needs a C++11 compatible compiler. You can create and run your own workloads, or you can download and use the workloads we used by executing the download\_workload.sh script enclosed in the attached file. You can then run it with the following command:

./[executable] < workload.txt

If necessary, you can also put the output to a file through output redirection. And among the attachments, result.txt is the correct output for the workload. Through the diff command, you can see if the memory has been correctly accessed and the value has been received.