# CMPSC473, Spring 2021
# Project 3: Virtual Memory Management

## Assigned: 19th March 2021
## Due: 09 April 2021, 11:59:59PM

Please read this document carefully!

[Note: There are no checkpoints and the whole assignment is due at once. However, we will be checking your incremental commits as a way of detecting possible violations of AI requirements.]

## Goals

The goals for this project are to help you:

1. Get experience working with a trace-based event-driven simulator
2. Get experience with a basic virtual memory management system (including MMU, TLB, DRAM, DRAM-resident multi-level page tables, page replacement algorithms, and swap device)
3. Get experience with simulation-based performance evaluation

## Project Overview

You will implement and experiment with a simple virtual memory manager (VMM) consisting of the following elements: (i) MMU, (ii) TLB, (iii) DRAM, (iv) DRAM-based multi-level page tables, and (iv) disk-based swap device. To do this, you will be given a **simulator base code**. If you have never worked with a simulator, do not worry. Much of the simulator functionality will be already implemented in the base code allowing you to focus on ideas related to VMM. Additionally, be sure to check out a **short video that has been uploaded in canvas announcements** - wherein we describe all you need to know to quickly become familiar with the base code.

Our base code simulates multiple processes time-sharing a single CPU with the help of the operating system which implements a round-robin CPU scheduler. The code base has been written to work for processes that *only execute non-memory instructions*. For each simulated process, the simulator reads the instructions from an "input trace file" for that process. You will extend this to simulate the time-sharing of processes that also execute memory instructions (such as loads and stores) that access virtual addresses that need to be translated into their physical counterparts by the MMU.

As input, you will receive two types of files: input files and traces. These files are present in the 'traces' folder.

# Where to find the simulator base code

The simulator base code is in the simulator.c file. You can compile it using the *make* command. Then, to run the simulator, you need to write the following command:

$ ./simulator input1.txt output1.txt
Where input1.txt is the input file and output1.txt will contain the output from running the simulator.

# Simulator Inputs

Our simulator takes two types of inputs for each simulation (henceforth also called "experiment"): (i) system parameters and (ii) a scheduling trace for each process involved in the experiment. These inputs are specified within a file that is passed as a command line argument to the simulator. So for an input file called input1.txt, the simulator would be invoked as follows, where the output will be written to output1.txt. More details on compiling and running the simulator appear under "Using the Simulator"

$ ./simulator input1.txt output1.txt

*The format of an input file is as follows:*

**# list of system parameters specified as parameter-descriptive-string parameter-value; (all latencies are in units of a "cycle")**
**# list of parameters that will remain the same across all experiments**
Non-mem-inst-length 1
Virtual-addr-size-in-bits 32
DRAM-size-in-MB 4
TLB-size-in-entries 16
TLB-latency 1
DRAM-latency 100
Swap-latency 10000000
Page-fault-trap-handling-time 10000
Swap-interrupt-handling-time 10000
TLB-type FullyAssociative
TLB-replacement-policy LRU
**# list of parameters that may vary across experiments**
P-in-bits 12
Num-pagetable-levels 3
N1-in-bits 8
N2-in-bits 8
N3-in-bits 4
Page-replacement-policy LRU
Num-procs 3
**# trace-file-name arrival-time-of-process**

process1 0
process2 3
process3 4

The processes are in order, meaning that each process has a start time later than or equal to the previous start time.

Code for parsing this file will be provided with the functions openTrace(), closeTrace(), readNextTrace() and readSysParam() in **fileIO.h/c** files.

The traces are: “input1.txt”, “input2.txt”, …., “input10.txt”.

Process trace files contain a list of addresses accessed by a program. The process trace file names will correspond to **trace-file-name**.txt from input trace files. So in the above example the “input1.txt” points to the 3 process file - “process1.txt”, “process2.txt” and “process3.txt”. ‘Mem’ or ‘NonMem’ is used to define the type of instructions.
Here is an example of the first few lines within a process trace file:

Total-num-instr 1000000
Mem 0x12f00C00
Mem 0x77f46B77
NonMem
Mem 0x7ffde001
Mem 0x12f0015C
NonMem
Mem 0x77f3c000
Mem 0x12f00C28
NonMem
NonMem
NonMem
NonMem
Mem 0x12f00010
Mem 0x12f0CC11
Mem 0x12f00100C
Mem 0x77f3CDE1
...

Code for parsing this file will be provided with the functions openTrace(), closeTrace(), readNumIns() and readNextMem() in **fileIO.h/c** files.

The process traces are available for each process in the traces: “process1.txt”, “process2..txt”, …., “process44.txt”.

# Using the Base Simulator

We have simulated the Round Robin process scheduler in the schedulingRR() function. This is the only scheduler you need to work with.

The base code simulates NonMem instructions **ONLY** and generates the following upon the completion of an experiment:

1. per-process completion time
2. total number of context switches
3. total amount of time spent in OS vs. total amount of time spent in user mode.

The current implementation works with "input1.txt" and "input10.txt" only which has all NONMEM instructions.

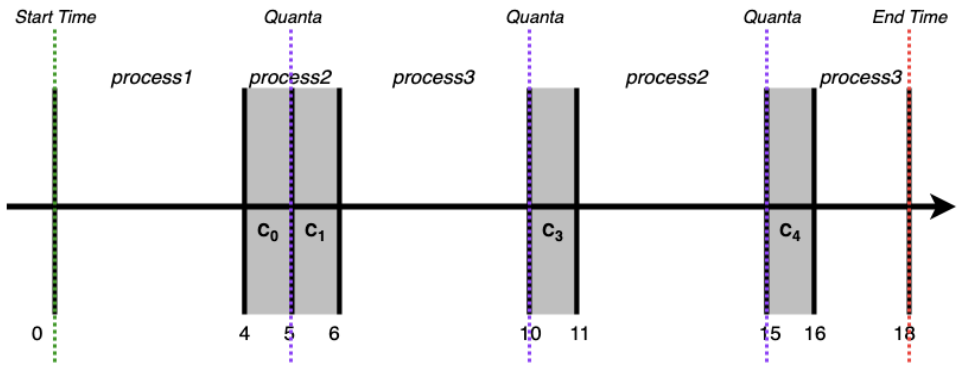
"input1.txt" has 3 processes: process1, process2 and process3. Each of these processes have 2, 2 and 3 NonMem instructions respectively.

Use the following commands to execute the simulator:
*Make*
*./simulator input1.txt output1.txt*
*[here input1.txt is the trace parameter you wish to experiment with, output1.txt is the file name where the output will be written]*

*Make clean* will delete all object files.

The execution order of the processes in "input1" are: *process1 → process2 → process3 → process2 → process3*; where *quanta = 5 and nonMemReadTime = 2*.

The below diagram shows the execution order of the 3 processes in "input1":




# What you need to do

You are required to append to this simulator to take care of all types of instructions and generate the following: *[You can use traces 2-9 as reference to test your code]*

1. per-process completion time
2. total number of context switches
3. total number of disk interrupts
4. per-process and total number of TLB misses (absolute and ratios)
5. per-process and total number of page faults (absolute and ratios)
6. per-process and total fraction of time in blocked state
7. total amount of time spent in OS vs. total amount of time spent in user mode.

To that end, you would have to implement data structures for TLB and page table and include necessary functions and helper data structures.

You must assume that the swap device has sufficient capacity to accommodate the active address spaces in their entirety in any experiment. You will assume that the capacity needs of the page tables themselves are accommodated by main memory other than the DRAM that the simulator considers (but that it has similar latency characteristics). You will ignore the main memory needs of the page tables themselves in your simulation of DRAM usage. You will assume that all the memory instructions in the trace files reference valid virtual addresses and, therefore, do not raise traps. All page table entries must be aligned to 4 Bytes. Besides the PPN, you only need to consider the valid bit within a PTE. All of these other PTE entries in a real page table are to be ignored: permissions, user/kernel, dirty, reference.

# Provided Code Structure

- **dataStructures.h** file contains the structs used in this project. You can add other structures here.
- **fieIO.h/c file** contains the helper functions for reading the traces.
- **gll.h/c** contains a linked list implementation for ease. It has been used in the existing code and you are welcome to use it for your linked list implementation.
- **simulator.h/c** contains the code for the simulator. You are expected to include your implementation in this file. There are **//TODO** instructions that give you some guidance about adding your implementation specific values in existing functions. You are expected to determine how to include your implementation yourself.
    - **init()** initializes all the data structures and list of processes. processList is the list of all processes. Any process that becomes ready to run is moved to readyProcess list. The process that should run right now is moved to runningProcess list. You should initialize your necessary variables/data structures here.
    - **simulate()** function is the simulator which loops through all the processes until all of them are done. It calls the processSimulator() function in a loop. processSimulator() simulates the running process until one of these things happen- timer interrupt, disk interrupt, page fault, process finishes. processSimulator() calls readIns() function that reads the next instruction for the running process. You have to handle “MEM” instructions in readIns() function.
    - **diskToMemory()** function moves one page from disk to memory. You have to implement it.

To implement these functions, you will need to implement TLB and page table along with MMU and swapping logic. You would also need to implement multiple page replacement algorithms(described in next section).

- Recommended order of implementation:
    - MMU, DRAM, swap device, Page table only with no TLB
    - Add TLB
    - Performance evaluation.

# Page Table Implementation (40%)

In this part of the assignment you need to implement a multi level page table with the following page replacement policies:

- **Optimal:** The page that will not be used for the longest period of time in the future is to be removed
- **LRU:** The least recently accessed page is the one to be removed.
- **Clock:** You should maintain a clock hand that rotates around memory. Each physical page is marked "unused". Each time a page is needed, the hand advances until it finds a page marked "unused", at which time that page is chosen for eviction. If it finds a page marked "used" in this process, the page is changed to be "unused". In addition, on every memory reference, the page accessed should be marked "used".

# TLB Integration (20%)

In this part of the assignment you need to implement a simulation of the Translation Look-aside Buffer (TLB) that uses the LRU replacement algorithm.

# Performance Study (40%)

After completing and testing your implementation so far, you will conduct two types of experiments. You will describe your findings in a project report (doc or pdf). We first describe these two types of experiments followed by the contents of your project report (large parts of the project report may be a google form where they will simply type in their outputs. If you can do this, it will greatly automate grading).

**Experiment type 1:** In these experiments, you will simply run your simulator with the following inputs and report the outputs 1-7 described under “What you need to do”:

- List of system parameters to use comes here. Choose LRU for page replacement policy.
- List of process mixes to consider comes here. Give them 5-6 mixes of increasing complexity.
- Ideally you would provide them a fixed format (it could even be a google form to automate grading)

**Experiment type 2:** In these experiments, you will vary certain parameters and study the impact on specified quantities. You will plot graphs to convey your findings in your report. You will use the XXX process mix for all of these experiments

- Keeping all other parameters the same as in Experiment type 1, vary p from 12 to 20 and plot the following: (i) average process completion time, (ii) per-process and total number of TLB misses (absolute and ratios), (iii) per-process and total number of page faults (absolute and ratios), and (iv) DRAM internal fragmentation. Choose the number of bits for all 3 page table levels to be as close to equal as possible.
- Keeping all other other parameters the same as in Experiment type 1, vary the number of levels in the page table from 1 to 3 and plot the following: (i) average process completion time and (ii) page table size per process. Choose the number of bits for the different levels within multi-level page tables to be as close to equal as possible.
- Keeping all other other parameters the same as in Experiment type 1, compare all outputs 1-7 with the page replacement policy being Optimal and Clock vs. LRU.

**Report:** Your report will consist of the graphs described above with a few (at most 5) sentences interpreting your findings for each graph (What trend did you expect based on your understanding from lectures/textbook? Does the trend match your expectation?)

# Handin Instruction

You will upload the commit ID that you want us to grade via canvas. The project report should be pushed to your repository (pdf/ doc format).

# Programming Rules

- IMPORTANT: You can enhance the PCB data structure.
- IMPORTANT: You need to create the MMU, TLB, DRAM, page tables, swap data structures from scratch.
- IMPORTANT: You must show your partial work on this assignment by periodically committing and submitting (i.e., pushing) your code to git. This will be used both for grading the checkpoints as well as ensuring academic integrity.

# Hints

- In addition to the usual scanning of provided code to understand how it works, we also encourage you to run it with provided traces (ones without any Mem instructions) as well as traces you construct yourself and use the observed behavior to help understand how it simulates a multi-tasked system.
- We encourage you to use the calculations you have seen in lectures and Exam 1 to create first-order models. You can use the estimates offered by these models to check whether your simulator's output appears to be in the right range or if it is way off which may be indicative of bugs in your code.