CSC 456 - Prog. 3, Simulation

Anthony Morast, Elizabeth Woody,
Zachary Pierson

Shared Memory

Separate documentation will be compiled and provided for the shared memory portion of the program.

Simulation

Project Description

This program is designed to simulate a the memory management unit, the process scheduler, and a few page replacement algorithms. The inputs and outputs of this program are designed to allow a user who has little knowledge of how these units work to get a decent understanding of these hardware components. Much of this program is run with randomly generated data, such as the processes, the page accesses, etc.

The process scheduler will output a few key values at each time step in execution time, this always includes the time step and the process id of the process currently running. Depending on the algorithm other information may also be displayed such as the burst time remaining or the priority of the process. For algorithms where it matters (shortest-job-first and priority scheduling) both preemptive and non-preemptive versions of the process scheduling algorithms are ran and the output is displayed. After the output containing process information is displayed, average turnaround time and average response time are output so the user can compare the algorithms.

As for the page replacement algorithms each algorithm, first-in first-out, optimal, least recently used, least frequently used, second chance, and clock, are all implemented and ran upon execution of this program. The replacement algorithms are ran and the output at each page access is stored in an array. At the end of execution a table that displays the frames' contents at the end of each page access attempt is output followed by the number of page faults and page hits so a user can compare the algorithms.

Finally, the Memory Management Unit (MMU) will simulate the Transition Lookaside Buffer (TLB), page tables, and paging. Every time a process accesses memory, a TLB check is performed to determine if the page had already been loaded. Each process has it's own page table that has information about what frames have been allocated to it. Every time a new process is started, available frames in memory will be allocated to it. In this simulation the user will be able to determine, at every step, what operation the Memory Management Unit should perform. This allows the user to play around with different options and configurations of the MMU.

Function and Class Description

main.cpp:

- *main(int argc, char *argv[])*
    - This is the entry point of the program. This handles and displays a main menu to the user for further interactions with other simulations.

- *proc_pageReplace_sim()*
    - This is the entry point for the process and page replacement simulations. This method handles the parsing and validation of the command line parameters and the generation of of random components such as process properties and page accesses.
- *PrintHelp()* -
    - This method is invoked by running the executable with '-help' as the first and only command line parameter. The function outputs a some information about using the program, what it does, and what the parameters are for. Note: there is also a usage message output if there are incorrect parameters passed to the program which is different from this message.

page_replacement.h:

- *class SecondChanceInfo*
    - *public*
        - *int page_contained* - the page currently in this frame
        - *int reference_bit* - the value of the reference bit corresponding to this frame
    - This class is used by the second chance and clock page replacement algorithms as they need more information than, say, the least frequently used page replacement algorithm.
- *void fifo(int num_accesses, int *reference_string, int num_pages, int num_frames)* -
    - A method that implements the First In First Out algorithm for page replacement. Naturally this is implemented with queue to keep track of what page should be replaced next. The first page that was inserted into the frames will be the first page to get replaced.
- *void optimal(int num_accesses, int *reference_string, int num_pages, int num_frames)* -
    - A method that implements the optimal page replacement algorithm. This algorithm looks into future page accesses and determines which page isn't used for the longest period of time. That page is then replaced with the incoming page. This method will produce the least page faults, hence the name optimal.
- *void lru(int num_accesses, int *reference_string, int num_pages, int num_frames)* -
    - A method that implements the Least Recently Used algorithm for page replacement. This works by looking at previous page accesses to determine which page has not been accessed in the longest period of time. That page is then replaced with the new one.
- *void lfu(int num_accesses, int *reference_string, int num_pages, int num_frames)*
    - A method that implements the least frequently used page replacement algorithm. This is done by looping through the number of accesses and determining if the current page in the reference string is in a frame. If not some page replacement logic is applied.

- *void second_chance(int num_accesses, int *reference_string, int num_pages, int num_frames)* -
    - A method that implements the second chance page replacement algorithm. This is done by looping through the number of accesses and determining if the current page in the reference string is in a frame. If not some page replacement logic is applied.
- *void clock_alg (int num_accesses, int *reference_string, int num_pages, int num_frames)*
    - A method that implements the clock page replacement algorithm. This is done by looping through the number of accesses and determining if the current page in the reference string is in a frame. If not some page replacement logic is applied.

process.h:

- *class Process*
    - *public*
        - *Process()* - Constructor
        - *int arrival_time* - the time the process arrives to be scheduled
        - *int burst_time* - how much CPU time is needed
        - *int process_id* - the processes identification number
        - *int priority* - the processes priority
        - *int finish_time* - the time when the process finished
        - *int response_time* - first response - arrival time
        - *bool first_service* - to determine if this is the first time the process has been serviced
        - *unsigned int mem_req* - The amount of memory a process requests
        - *unsigned int* table* - The page table for this process
        - *unsigned int table_size* - The size of the page table
        - *unsigned int frag* - The amount of internal fragmentation (bytes)

process.cpp:

- *Process()*
    - This Constructor simply initializes the page table to NULL and sets the size of the page table to 0;

process_scheduler.h:

- *class ProcessScheduler*
    - *public*
        - *void round_robin (Process *processes, int quantum, int num_procs)*
        - *void priority (Process *processes, int num_procs, bool preempt)*
        - *void shortest_job_first (Process *processes, int num_procs, bool preempt)*
    - *private*
        - *Process *burst_sort (Process *processes, int num_procs)*
        - *Process *priority_sort(Process *processes, int num_procs)*

        - *Process *arrival_sort (Process *processes, int num_procs)*
        - *float calculateTurnaround (Process *processes, int num_procs)*
        - *float calculateResponse (Process *processes, int num_procs)*

- ***ProcessScheduler Implementations:***
    - *void round_robin (Process *processes, int quantum, int num_procs)*
        - The round_robin function takes a list of processes that are to be executed, a time quantum which is how long each process gets the CPU and a count of the number of processes to be scheduled. The method uses this information to simulate the processes being scheduled using the round robin process scheduling algorithm. The process being executed along with the CPU time are output at each clock cycle and at the end of the method the average turnaround time and average response time are displayed for comparison purposes.
    - *void priority (Process *processes, int num_procs, bool preempt)*
        - This method implements the priority process scheduling algorithm on the list of processes passed to the method. A count of the processes and a boolean indicating whether or not preemption will be used in the algorithm are also passed as parameters. The process being executed along with the CPU time and the process' priority are output at each clock cycle and at the end of the method the average turnaround time and average response time are displayed for comparison purposes.
    - *void shortest_job_first (Process *processes, int num_procs, bool preempt)*
        - This method will implement the shortest job first scheduling algorithm on the processes passed in as an argument. The process being executed along with the CPU time and remaining burst time are output at each clock cycle and at the end of the method the average turnaround time and average response time are displayed for comparison purposes.
    - *Process *burst_sort (Process *processes, int num_procs)*
        - This method will sort the processes passed as a parameter based on burst time remaining, lowest burst time remaining first, and return the sorted array of processes. This method was used by the shortest job first algorithm.
    - *Process *priority_sort(Process *processes, int num_procs)*
        - This method sorts the array of processes based on priority, the highest priority values are placed at the beginning of the array, and returns the sorted array. This method was used by the priority sort algorithm.
    - *Process *arrival_sort (Process *processes, int num_procs)*
        - This method sorts the process array based on the arrival time of the processes, the lowest arrival times are placed at the beginning of the list, and returns the sorted array. This method was used to make scheduling more easy since if the process arrival time was less than the

CPU time it would not be considered for scheduling since it “wasn’t there”.

- *float calculateTurnaround (Process *processes, int num_procs)*
  - This method calculates and returns the average turnaround time of an array of processes. That is it sums each process’ finish time minus the process’ arrival time then divides by the number of processes.
- *cfloat alculateResponse (Process *processes, int num_procs)*
  - This method calculates the average response time of an array of processes by summing the times at which the processes were first serviced by the CPU (response time is a field in the Process class). Then the sum is divided by the number of processes and the result is returned.

memory_management.h:

- *int virtualSpace* - virtual address space defined in number of bits
- *int physicalSpace* - Physical address space defined in number of bits
- *int pageSize* - Size of page, defined in number of bits
- *int numPages* - Number of pages also defined in number of bits
- *int numFrames* - Number of Frames also defined in number of bits
- *int tlbSize* - Number of entries in the TLB
- *unsigned int* TLB* - TLB page entries
- *vector<unsigned int> freeFrames* - Vector of free frames available
- *vector<unsigned int> referenceString* - Vector containing recent page accesses

memory_management.cpp:

- *MMU (vector<Process>& procs)*
  - The constructor simply makes a call to *Setup()* that literally sets up the Memory Management Unit variables.
- *~MMU (vector<Process>& procs)*
  - The destructor deallocates memory that was used for the TLB
- *void Setup(vector<Process>& procs)*
  - The function prompts the user for member class variables and does error checking along the way. The information that the user provides will be used to calculate the number of pages and frames. The TLB is also dynamically allocated and initialized.
- *void View_System_Info(vector<Process>& procs)*
  - Prints information about the Memory Management Unit to the terminal.
- *void View_Process_Info(vector<Process>& procs)*
  - Displays process information including the page tables for each process and their amount of internal fragmentation.
- *unsigned int* Config_Page_Table(unsigned int memReq, unsigned int& pageAlloc, unsigned int& fragmentation)*

    - This function creates a page table for a process if there are enough free frames available. Free frames are assigned to the process and taken out of the freeFrames vector.
- *void Terminate_Proc(Process& proc, unsigned int id)*
    - This function gives back the frames that it was allocated to the MMU. The page table is then dealocated and set to NULL.
- *void Terminate_All_Procs(vector<Process>& procs)*
    - Makes successive calls to *Terminate_Proc()* on all processes that are currently running.
- *bool Access_Memory(Process& proc, unsigned int mem)*
    - The process provided attempts to access memory at location *mem*. The page number and offset are extracted from *mem*. A check for invalid memory access and TLB is then preformed.
- *bool TLB_Hit(unsigned int page)*
    - This function checks the TLB if there is a hit. The TLB simulation implements the Least Recently Used algorithm to replace contents.
- *void Access_Multiple_Memory(vector<Process>& procs)*
    - This function makes successive calls to *Access_Memory* on each of the running processes.
- *int Total_Fragmentation(vector<Process>& procs)*
    - This function simply adds up all of the internal fragmentation of all the processes.
- *bool Create_Proc(Process& proc)*
    - Prompts the user to specify how much memory the process is requesting. The page table for the process is then created.
- *void Create_Multiple_Procs(vector<Process>& procs)*
    - Prompts the user for number of processes to be created along with upper and lower bounds for requesting memory. Processes that are successfully created are pushed to the procs vector.
- *bool prompt_mem_access(int& low, int& high)*
    - This function prompts the user for bounds on memory access. This function also provides error checking.
- *bool prompt_proc_req(int& numProcs, int& low, int& high)*
    - This function prompts the user for information about creating multiple processes. Error checking is also handled in this function.
- *int custom_menu_prompt()*
    - Displays options that the user will be able to choose from. Error checking will have to be handled in the calling function.
- *void memory_management_sim()*
    - This is the main body of the MMU simulation. A menu will show the user what operations are available to choose from.
- *int random_int(int min, int max)*

- This function simply grabs a random number that falls between the minimum and maximum bounds specified.

How to Compile

There is a makefile included in the main directory that will compile the simulation code into an executable in the main directory named *simulation.* There is a separate makefile in the shared_memory directory that will compile that code into an executable.

How to Use

Run the executable called simulation. This will bring you to the main menu. from here you have 3 options:

```
                Simulation Main Menu
=========================================================
1. Run process schedular and page replacement simulations
2. Run Memory Management simulation
3. Exit program
> 
```

1. The Process Scheduler and Page Replacement simulations takes four command line arguments:

   <num_procs> <quantum> <num_frames> <num_pages>

   1. num_procs - the number of processes to be run through the process scheduler
   2. quantum - the time quantum used for round robin process scheduling
   3. num_frames - the number of frames used for page replacement
   4. num_pages - the number of pages used in the page replacement algorithms

2. Memory Management simulation will prompt the user to provide information regarding the MMU. The user will then be directed to an interactive menu:

```
                User Defined Menu
                =================
                1.  Reset System
                2.  Create Single Process
                3.  Create Multiple Processes
                4.  Terminate Process
                5.  Terminate All Processes
                6.  Access Memory
                7.  Access Multiple Memory
                8.  View Process Information
                9.  View System Information
                10. Back to Main Menu
        Choice: 
```

The user now will be able to do all of the following operations. The intent behind this interactive menu is to allow the user to be able to create specific situations in order to see how the Memory Management Unit would respond.

3. The final option is to exit the program.

Note, for page replacement there is a random reference string generated that has between 10 and 29 accesses. The processes are also generated at random, the user can only choose how many processes and the time quantum. Each processes burst time is between 1 and 10, the arrival time is between 0 and 5, and the priority is between 0 and 5 as well.

Included Files

prog3 directory:

- **makefile** - to compile the simulation program
- **shared_memory directory** - contains the shared_memory code
- **simulation_code directory** - contains code used to implement the simulations

simulation_code directory:

- **main.cpp** - The entry point of the program. Handles the parsing of the command line parameters, the random generation of certain information, and calls the individual algorithms used for simulation.
- **page_replacement.h** - Contains all of the page replacement algorithm implementations. These include first-in first-out, optimal, least recently used, least frequently used, second chance, and clock.
- **process.h** - A class used for the process scheduling algorithms.
- **process.cpp** - contains a constructor for the Process calss.
- **process_scheduler.h** - A header file containing the implementations of the process scheduler simulation algorithms. These include round robin, shortest job first, and priority scheduling.
- **memory_management.h** - a header file containing the class structure and function prototypes.
- **memory_management.cpp** - A class used for the process scheduling algorithms with implementations of the memory management unit algorithms. These include TLB, page tables, and paging.

shared_memory directory:

- this directory's files will be documented in the aforementioned separate shared memory documentation.

System Requirements

A machine running Linux with g++ version 4.8.2 installed is needed to compile this program. The program also depends on C++ 11 components so *'-std=c++11'* must be

specified on the command line. Other than these few software requirements there are no requirements on the system's specifications as far as hardware.

Testing

To test this program initially hard-coded reference strings and processes were used to ensure all the functionality of the program worked. This included testing edge cases, empty cases, trivial cases, etc. After this testing was done the random generation was implemented. With this generation the program was executed repeatedly to ensure the program works for a large range of possible inputs.

The main function was tested to ensure most input scenarios were validated correctly and, if invalid parameters were provided, that the user was given the correct message to allow them to correct their input.

Teaching Tool

The program is useful as both simulation software and as a learning tool. The program covers many aspects of operating systems including the memory management unit, process scheduler, and page replacement. For a user with little to no knowledge of operating systems the output of this program can aid in teaching them the operation of these units when a few select algorithms are implemented.

The process scheduler simulates the round robin, priority, and shortest job first scheduling algorithms. The execution of this program will invoke each of these algorithms. Furthermore, for priority and shortest job first scheduling, both preemptive and non-preemptive versions of the algorithms will be invoked. This results in five different scheduling algorithms being executed.

The scheduling algorithms will output which process (process id) is being executed at each time step and the time steps value. Along with this the priority scheduling algorithms will display the priority of the process and the shortest job first algorithm will output the burst time remaining. This output allows a user to track the process of the processes due to the scheduling algorithms and helps understand exactly how each algorithms operates. At the end of each algorithm the average turnaround time and average response time are calculated and output so the user can compare these key statistics of the algorithms and determine which algorithms are best and in which scenarios.

The page replacement algorithms are all ran on program execution. The algorithms ran include first in first out, least recently used, optimal, least frequently used, second chance, and clock. These algorithms are given a reference string, a number of pages, and a number of frames and determine the number of page hits and misses.

After execution of each algorithm a table is generated that displays the page contained in each frame at each attempted page access in the reference string. This way a user can watch how the algorithm decides which page is to be replaced. After the table is displayed the number of page faults and page hits are displayed so the algorithms can be compared and it can be determined which algorithms perform best in each particular situation.

The Memory Management simulation is a great tool for learning how TLB, page tables, and paging work. The user is left with a lot of power at their fingertips, they get to choose what processes to make and what memory they would like process A to access. For each operation performed, the user will see diagnostic information about how the paging and page tables work. It is recommended to use small numbers as this might provide more insight into how these parts of an operating system function.