Operating Systems
Final Program Documentation

Daniel Nix Chris Smith Matthew Rames

May 1, 2015

Contents

Chapter 1

Shared Memory: Mailboxes

Operating Systems, Program 3
D-shell Shared Memory Daniel Nix May 1, 2015

Program Description The D-shell is a simple shell to provide the user with information about processes running on the system. The purpose of the program 3 expansion is to implement shared memory between D-shells, specifically:

- Create a block of shared memory mailboxes with `mboxinit`
- Write to a specific mailbox with `mboxwrite`
- Copy one mailbox to another with `mboxcopy`
- Free the block of shared memory with `mboxdel`

1.1 Algorithm

1.1.1 Event Loop

The event loop is handled in main and has not changed from program 2, aside from accepting new shared memory commands. As long as "exit" has not been executed, the program continues to accept commands from the user.

1.1.2 Parsing

To implement the new functions, the "parse_arg" function was added to split a string of commands into a 2 dimensional character array.

1.1.3 Shared Memory Implementation

To implement shared memory the c library sys/shm.h was used. It allowed easy creation of a shareable block of memory using a key managed by the OS.

Mailbox Creation

Creating shared mailboxes requires two features to be addressed:

1. Allocate memory to be shared
2. Split shared memory block into individual mailboxes

To allocate memory a call to shmget was made with a request for the appropriate amount of memory and flags IPC_CREAT and IPC_EXCL set. This indicates if shared memory with the given key is already in existence. If shared memory exists we know that reinitializing shared mailboxes unnecessary so the D-shell notifies the user that shared memory has already been set up before returning to the D-shell command line.

If memory has not been set up then the newly allocated memory must be split into individual mailboxes. The shared memory block begins with a small mailbox to hold two integers, the first integer to hold the number of subsequent mailboxes and the second integer to indicate the size (in KB) of each mailbox.

To ensure the read, write, and copy functions behave correctly the first character of all mailboxes is initialized to null.

Synchronization while using mailboxes is discussed in section 1.1.4.

Read from Mailbox

Mailbox reading requires the memory location of the mailbox. The helper function `box_num_to_addr` is used to convert the box number to base address. Using this address the mailbox is read until a null terminator is found. The write function guarantees that

the final character in a mailbox string is a null terminator so the size of a mailbox is not required.

Reading also requires a guarantee that no other D-shells attempt to write to that mailbox at the same time. This means that a semaphore on the mailbox must be set before the read and released upon read completion. Semaphore use is further discussed in section 1.1.4.

Write to Mailbox

Mailbox writing requires the memory location of the mailbox determined by the function `box_num_to_addr`, and the size of a mailbox held in the first mailbox in the shared memory block.

Before writing, the message provided by the user is checked for length. If it is larger than the size of a mailbox minus 1 (to allow for the null terminator) the message length is shortened to fit in a mailbox without overflow. The message is then written to the desired mailbox with a null terminator appended to indicate the end of used memory in the mailbox.

The D-shell must also guarantee that no other D-shells are accessing the mailbox while a write is performed. This is further discussed in section 1.1.4.

Mailbox Deletion

Before deleting a mailbox block the D-shell must confirm that this instance of the shell created the memory. This is done with a flag set when memory is created set to true if this shell created the memory and false otherwise.

If this shell created the memory then a lock on all memory locations must be made to ensure no other D-shells are accessing the shared memory when it is deleted. This is further discussed in section 1.1.4.

Once it is confirmed this shell created the shared memory and no other shells are accessing it, mailbox deletion is done with a call to `shmctl` with the IPC_RMID flag set.

1.1.4 Synchronization

Any time shared memory is implemented the standard synchronization problems including data races, starvation, and deadlock arise. To address these problems the c sys/sem.h library was used.

Data Races

When using shared memory the D-shell must guarantee that memory stays consistent between all shells. For ease of implementation the D-shell allows one shell to read or write to a memory block at a time. This approach was selected for ease of implementation and rate of memory access. If the shared mailboxes were to be used in a high performance computing application then this solution would likely lead to starvation but for a simple message passing system between shells it is effective.

Starvation

As with any shared memory implementation, starvation of processes must be addressed. However, it was not a major concern in D-shell shared memory implementation because of the low rate of memory accesses. Shared memory is only accessed when a user manually requests a read, write, or copy with the respective mbox command. Each access takes only a short amount of time, orders of magnitude less than the fastest rate a user could manually send them, so there is essentially no risk of starvation.

Deadlock

To avoid deadlock the D-shell is only allowed to read or write to one mailbox (except in the case of copying in which case a read followed by a write is done). This guarantees no deadlocks may occur because a read or write may only last a finite amount of time and all processes will finish.

The only concern of deadlock arises when two shells attempt to copy at the same time. For example, shell 1 attempts a copy from mailbox 1 to mailbox 2 while shell 2 attempts a copy from mailbox 2 to mailbox 1. This does not threaten deadlock because, even if both shells began their reads at the same time, the reads would finish, the semaphores would be released, and the writes would be carried out.

1.2 D-shell Testing

Unit testing for the D-shell shared memory implementation was done in an iterative process. Each time code was added to the shell it was tested to confirm the new code performed the desired function and did not break existing shared memory implementation.

After completing shared memory implementation with unit testing, D-shell regression testing and system was done. All commands previously allowed were tested to confirm that previously written functions were not affected by the newly implemented shared memory code. New functions were also tested to confirm the system worked as a whole.

1.3 Submission Description

1.3.1 Compiling Instructions

To compile the source code, unzip the prog3.tar file. From the prog3 directory, type “make" and the source code will be compiled. The dash executable will be placed in the root prog3 directory. Run the D-Shell with the command "./dash". All output from the dash shell is directed to the terminal.

1.3.2 External Functions

For portability, functions were separated into dash1_funcs.cpp, dash2_funcs.cpp, and dash3_funcs.cpp. The header file dash.h was also created to hold constants.

Chapter 2

Process Scheduling

Program Description The purpose of the process simulation is to demonstrate how different process scheduling methods work to a new user. A user may select from a list of page replacement algorithms consisting of:

- Round Robin
- Shortest Job First
- First Come First Serve

The output of each algorithm is when a process is running or if the system is in an idle state. The application will give a time for when each process starts and finishes.

2.1 Algorithm

2.1.1 Setup

Each simulation begins by prompting the user to enter in the number of processes they would like to simulate between 1 and 10. In the Round Robin simulation the user is also prompted to enter a quantum. The simulation then will prompt the user to enter arrival and burst times for each process. Once the user has all the information inputted a process table will be displayed. A breif description of the simulation is outputted and then the simulation starts. The output is when a process starts and finishes and if the system is idle it will output how long the system is idle.

2.1.2 First Come First Serve

In the First Come First Serve process scheduling alogrithm, process are ran as soon as they arrive if they can. If not, they are added to a queue and are executed once the other process finishes. If nothing is in the queue the system is idle.

2.1.3 Shortest Job First

In the Shortest Job First process scheduling algorithm, the processes that have the smallest burst and have arrived are executed first. If a process is already running they are put in a queue and will be executed once the other process has finished.

2.1.4 Round Robin

In the Round Robin process scheduling algorithm, a quantum is added which is a block of time that a process will run before starting the next process in the queue. If there are no processes that have arrived yet the system is idle for that quantum. Round Robin trys to make sure every process gets a reasonable response time. Please note each block of text is a quantum.

2.2 Simulation Testing

Testing for each algorthm was done by testing all the different edge cases that could happen with a sort. Even testing bad input to make sure proper error messages are displayed. When teach algorithm was confirmed to be correct it was pushed to the GitHub repository. All simulations ran for tests resulted in correct output.

2.3 Submission Description

2.3.1 Compiling Instructions

To compile the source code, unzip the prog3.tar file. From the prog3 directory, type “make" and the source code will be compiled. The mmu_sim executable will be placed in the root mmu_sim directory. Run the mmu_simlation with the command "./mmu_sim".

All output from the simulator is directed to the terminal.

2.3.2 External Functions

For portability, functions were separated into reloc.cpp, tlb.cpp, and paging.cpp. As well as corresponding header files in an include directory.

Chapter 3

Memory Management Unit

Program Description The MMU-Simulation program is a simple implementation of the memory management unit (mmu) that simulates the relocation register, paging, and paging with a Transition Look-Aside Buffer (TLB). The purpose of this programe is to simulate the different methods of mmu.

- Creates a section of memory to run in or pages for paging and the tlb
- Creates a offset to access the memory by and a page for paging and the tlb
- Accesses physical memory with the given offset or a page with given offset
- Successful if offset doesn't overstep bounds of max or frame size

3.1 Algorithm

3.1.1 Event Loop

The event loop is handled in main and uses an array of function pointers to call the relocation register, paging, and tlb functions. The event loop gets the user input with 1 being relocation register, input of 2 is paging, input of 3 is tlb, and 4 exiting the program. Any other number input causes an error message to be displayed and redisplays the prompt.

3.1.2 Relocation Register

To implement the relocation register a logical max is created. Then the offset for physical memory is created which will be used as the relocation register. Last a offset to access memory is created and used to see if it stays within the bounds of the physical min and max of memory.

3.1.3 Paging

To implement paging the user is asked to enter how many frames that are to be implemented, how many pages to implent with pages being less than or equal to the number of frames. Next page and frame size is asked to be a power of two. The page table is then displayed to the user so they know what page is referencing what frame. Once everything is set up the algorithm starts creating random pages and random offsets that are used to access physical memory. If the offset is less then the page size the frame was accessed successfully, if it wasn't then it is checked if it goes into another frame or out of physical memory. This occurs a number of iterations that the user specifies.

3.1.4 Transition Look-Aside Buffer

Creating the TLB requires:

1. Pages Table and Frames
2. TLB Table
3. TLB Replacement Algorithm

Pages and Frames are created using the same scheme in the Paging section except that the algorithm is not used just the creation of the pages and frames.

To create the TLB table the pages and frames are selected randomly to fill the TLB table. The Size of the TLB table should be less then the number of entries in the page table. The associated frames of the pages stored in the TLB are also stored in the TLB frame portion.

The user is then prompted with what replacement algorithm they would like used on the TLB if there is a miss when accessing it. The choices are random, round-robin, and least recently used.

TLB Replacement: Random

The Random replacement algorithm for the TLB is a straightforward approach. The program will randomly select an index to replace in the TLB table with the page that was missed and then replace that entry with the page and associated frame.

TLB Replacement: Round-Robin

The round robin replacement algorithm is a little more involved than the random replacement with its book keeping of what pages are in the que. A que is created that will store all the pages and use the 0 index as the exit point of the que and the last index as the entry point after it is removed from the TLB. When The TLB is first created the pages that are currently in the que are put at the end of the que since every page should get equal priority of going in TLB and que to be in the TLB. If a TLB miss occurs the 0 index in the TLB will be removed and added the end of the que with everything in the TLB shifting up one and the page in the 0 index in the que is put in the last index of the TLB. The que is then shifted down and the program continues.

TLB Replacement: Least Recently Used

The least recently used replacement algorithm keeps track of how often a process is used. The more often it is used the more chances it has to stay in the TLB and not get removed and replaced when a miss happens. If all pages have been used the same number of times the random replacement algorithm is selected to do the TLB replacement. If a few are equally least used the first found will be selected to be replaced.

3.2 MMU-Simulation Testing

Unit testing for the MMU-Simulation program was done in an iterative process. Each time code was added to the shell it was tested to confirm the new code performed the desired function and did not break existing memory magement unit functions.

3.3 Submission Description

3.3.1 Compiling Instructions

To compile the source code, unzip the prog3.tar file. From the prog3 directory, type “make" and the source code will be compiled. The mmu_sim executable will be placed in the root mmu_sim directory. Run the mmu_simlation with the command "./mmu_sim". All output from the simulator is directed to the terminal.

3.3.2 External Functions

For portability, functions were separated into reloc.cpp, tlb.cpp, and paging.cpp. As well as corresponding header files in an include directory.

Chapter 4

Page Replacement

Program Description The purpose of the page replacement simulation is to demonstrate different page replacement algorithms to a new user. A user may select from a list of page replacement algorithms consisting of:

- First In First Out
- Optimal
- Least Recently Used
- Least Frequently Used
- Second Chance/Clock

The output of each algorithm is a table representing frames filled with their respective pages in memory along with a note of what happens with each page request.

4.1 Algorithm

4.1.1 Setup

Each simulation begins by prompting the user for a number of physical memory frames (F), number of different frames the simulated program may request(P), and the number of page requests the simulated program will make(R). A table simulating empty physical memory with F frames is created first. Then a list of randomly generated pages from numbers 0 to P is created.

The program then simulates a page request for each request in the list. If there is an empty entry in physical memory it is used. If the page is already in memory it is not swapped out, but may be updated according to the appropriate algorithm. If there is a page miss the selected algorithm is invoked to determine the victim frame which is swapped for the requested frame.

4.1.2 First In First Out

The FIFO algorithm tracks the order the pages were put into physical memory and selects the first one to arrive as the victim frame. A deque is used to track frames as they are brought into and taken out of memory. When a page miss occurs the first item in the queue is dequed and its frame number is used as the index to store the requested page in physical memory.

4.1.3 Optimal

Optimal simulates a perfect situation when we know the minimum time each frame in physical memory has before being used again. Using a greedy algorithm to select the frame with the "longest-to-next-use" time as the victim frame guarantees the minimum number of page misses.

While a real machine cannot see into the future to implement this algorithm this simulation has the full list of page requests to simulate what would happen if this were possible. This provides a goal for the minimum number of page misses for a given physical memory size and page request list.

4.1.4 Least Recently Used

The least recently used (LRU) algorithm selects the frame that has sat idle for the longest amount of time as the victim. This comes from the assumption that if a frame has not been used in a while it is not likely to be required soon.

The LRU algorithm is implemented by setting a counter in each page to the current clock time each time a page request is made. This makes it easy to see which frame in physical memroy has not been used for the greatest amount of time.

4.1.5 Least Frequently Used

The Least Frequently Used (LFU) algorithm selects the frame in physical memory with the smallest number of previous page requests as the victim. This comes from the assumption that if a page has not been used many times in the past it is not as likely to be used again in the future.

The LFU algorithm is implemented by creating a counter for each possible page that might be requested. Each time a page is requested its counter is incremented. When a page miss occurs the simulation looks at the counters for each page in memory and selects the page with the smallest counter as the victim.

4.1.6 Second Chance/Clock

The second chance algorithm attempts to avoid page misses by giving a page a second chance at being a victim. That is, if a page is used twice then there is a better chance that it will be used again than if the page has only been used once. A second chance counter is used to track the pages that have been requested twice.

The second chance algorithm is implemented in essentialy the same manner as the LRU algorithm. It searches for the page in memory that is least recently used but as opposed to selecting it as a victim frame immediately, a second chance counter is checked. If the second chance counter is 1, it is decremented to zero and the page is not replaced. If the second chance counter is 0 the page is selected as a victim.

When a page request is made for a page already in the frame table, that frame's second chance counter is set to 1 indicating that it has been used and should be given a second chance.

4.2 Simulation Testing

Testing for each algorithm was done by printing a verbose output for each step in the page replacement algorithm and following it by hand. When each algorithm was confirmed to be correct it was pushed to the GitHub repository. This method for testing ensures correctness for most cases but, in some extraneous circumstance, may have missed an unknown bug that could occur in a larger simulation. However, all simulations run in testing produced correct output.

4.3 Submission Description

4.3.1 Compiling Instructions

To compile the source code, unzip the prog3.tar file. From the prog3 directory, type “make" and the source code will be compiled. The page replacement executable will be placed in the root prog3 directory. Run the simulation with the command "./page_repl_sim". All output from the dash shell is directed to the terminal.

4.3.2 External Functions

For portability, functions were separated into paging.cpp and paging_algorithms.cpp. Header files for each cpp file were also created was also created to hold constants, structures, and function prototypes.