CMPS 455 - Project 3 Report

Samantha Luke - C00210755
Konnor Miller - C00235967
Ben Hearod - C00022761

Task 2 - Multiprogramming

1. Explain how you manage the memory.

Memory is managed via a bitmap. A process will be assigned the first free bits in the bitmap equal to the amount of pages needed to run the process. If there is a second process being run, it will be assigned the next few free bits, also equal to the amount of pages needed to run that process.

2. Describe your memory allocation and deallocation scheme.

The virtual and physical pages are set to a free space in the bitmap. The memory is allocated via the bitmap. Memory is deallocated by clearing the bitmap, or moreso clearing the bits used for the process that has ended.

3. How does NachOS start a user process? Walk through the algorithm step by step.

NachOS starts a process in progtest.cc with StartProcess. StartProcess then makes an AddrSpace for the current thread running, which is running the process.

Task 3 - Demand Paging

1. Compare and contrast the changes made to AddrSpace for this task with the changes made for Task 2.

The changes made from task one was to remove the all or nothing memory allocation and change the page table initialization from setting all valid bits to true, to be false. The first attempt at making the method Update Page used a while loop that would iterate a value up until the first page with the valid bit set to false. Then it would update the properties of the page such as turn the valid bit to be true and update the physical page variable. However this method did not work, it would only load one page despite multiple page faults. Konner was able to find a solution and fix the issue. He used a for loop which looked for a matching virtual page which is found during a page fault.

2. What steps do you take when a page fault occurs? Explain in detail.

To handle page faults there are three pieces of information needed. First the bad virtual address (badVArre), second the bad virtual page (badVPage), and lastly by use of the method bitmap->find() to find an available physical page. BadVArre is found by reading the machine register using a calculation that is made with the page table variable to find the BadVPage. With this information we can read from the buffer to a physical page in memory. We use the badVPage to find an initial location in the buffer and load 265 bytes into memory at a location determined by the bitmap times the badVPage with the use of ReadAt. After this is done UpdatePage is called and some of these values are passed to find the appropriate page and update the page info, such as set the valid bit to true and link the physical page with the virtual page.

Task 4 - Swap Files

1. How did you modify the AddrSpace constructor for this task?

The AddrSpace constructor was modified to include a thread id as a parameter. This is so that each program run will have a different swap file, indicated by the thread id of the program.

2. If you created any new classes or data structures, explain them. If you did not, say so.

The only data structure related thing I added was a public char array of the name of the swap file created. This will make it so that you are able to open and use the swap file in other files, like exception.cc.

Task 5 - Virtual Memory

1. Explain the structure of an IPT and how it is used by your code.

In our NachOS implementation of the Inverted Page Table, the IPT is an array of thread pointers initialized to the size of the amount of pages in memory, which here is 32. The IPT is instantiated in system.cc so that it is visible to all of the threads and other files as a global variable. When PageFaultException is called, if virtual memory is enabled, the code uses the IPT to find the thread using the physical page we're replacing so that we can check the dirty bit of that thread, access the thread's name to open the correct swap file, and set the valid bit to false. We then replace the pointer in the IPT at that physical page location to the swapped out thread with the pointer to the thread we're swapping in. Basically, the IPT acts as our virtual memory that allows us to keep track of the threads and associated data.

2. Select at least 2 numbers for use with -rs. Run a series of tests on a minimum of 3 different user programs with the -rs option enabled. For each program, use all permutations of virtual memory options and your chosen -rs seeds. Record the programs used, page replacement algorithms, -rs seeds, number of page faults, and number of timer ticks in a table. Using this data, comment on the performance of each algorithm, and explain your reasoning.

Program	-rs seed	Replacement	Page Faults	Timer Ticks
sort	54345	FIFO	20	21,589,671
	9231	FIFO	20	21,588,222
	54345	Random	20	21,589,695
	9231	Random	20	21,588,186
matmult	54345	FIFO	24	701,735
	9231	FIFO	24	702,144
	54345	Random	24	701,917
	9231	Random	24	702,187
halt	54345	FIFO	2	24
	9231	FIFO	2	34
	54345	Random	2	24
	9231	Random	2	34

For each of the -rs seeds and programs, the algorithms performed extremely similar to each other. With *halt*, the timer ticks were the exact same using FIFO and Random. With the more complicated programs such as *matmult*, we observed that the FIFO algorithm performed slightly better than the Random algorithm with both of the two seeds. Specifically, the FIFO algorithm timed 182 ticks faster than the Random algorithm using the first -rs seed, and 43 ticks faster using the second -rs seed. This could suggest that the FIFO algorithm worked slightly better in this case when compared to the Random algorithm, though not by much. Neither algorithm is particularly better than the other, as neither take into consideration things like predicted burst times and rather operate on fairly random placement and timing of processes.

Task 8 - Report

1. What problems did you encounter in the process of completing this assignment? How did you solve them? If you failed to complete any tasks, list them here and briefly explain why.

For multiple programs, there was some confusion as to how to get more than one to run, but then figured out what to do with the bitmap. In demand paging the first iteration of the page update method did not work, it only loaded a single page regardless of how many page faults there were. The first assumption was to look for a page with the valid bit, and with a while loop set a variable to where that page was and use it to update the page. This method did not work and so Konnor fixed it by using a for loop to find a page that matched the virtual page with badVPage. The first iteration led to some confusion on how readAt worked which stalled development until the pageUpdate was fixed.

2. What sort of data structures and algorithms did you use for each task? Did speed or efficiency impact your choice at all? If so, how? Be honest.

For demand paging mostly primitives like ints were used. The first iteration of the method page update used a while loop but was later changed to a for loop. Bad address variable was located by use of machine->ReadRegister code which then bad virtual page was derived from. The next unused physical page was found via the bitmap. For the demand page part in terms of efficiency, was not really considered too much. The first version of the page update method was an attempt to be efficient but the logic was wrong but in the end the fixed version was more efficient. For swap files, OpenFiles were used to make, store, and get stuff out of the swap files. The name of the swap file was then saved in the addrspace so that it can be opened again in the PageFaultException. Neither speed nor efficiency impacted my choice, this was the only way I could think to implement the swap files.