Steven Babineaux smb0007
Jarret Delle Donne jad8149
Deandre Metoyer ddm1025
Ian Beatty imb4769

## **For Two-Level Hierarchical Page Table: -B**

Task 2: Demand Paging

1. **Compare and contrast the changes made to AddrSpace for this task with the changes made for Stage 3: Multiprogramming.**

In stage 3, we allocated memory for a process when the system call Exec was used. Now, in stage 4, we set the valid bit to every entry in the page table to false. but we don't allocate the memory until the thread that holds the process starts to run.

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

The virtual address that failed to translate into a loaded physical address is stored in register 39. We take that address and turn it into a virtual page number. First we find an available physical page by using bitmap->Find() if this returns -1(Full memory) nachos is crashed. If we do find an available page, the current thread runs the Assign Page function in AddrSpace.cc. We pass the VPN and the Physical Address we got from the bitmap to this function. Inside this function we zero out the memory, the size of a page, and prepare to load it into main memory using the ReadAt Function. We've stored the executable inside of a member of AddrSpace class. We take the Physical address and multiply it by PageSize and this is the position of the place we want to load memory in main memory. The size of the part we're loading in is PageSize. We start reading from the noff's file address and add the VPN * the PageSize. After reading into memory, we simply close the file.

Task 3: Swap Files

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

We take the file's name and store it within our own filename char array. We then create a few more char arrays and copy in the data and code from the executable. We then create a file called (threadID).swap We then use the WriteAt function to copy in the data from the executables that we stored in those previous char arrays into this new file. We then copy the name of the swap file into that character array that stored the file's name. Now we have access to it later for loading pages. After that we close the swap file and delete all the allocation created for copying in the executables data.

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

We only added a character array to store the name of the swap file.

**Task 4: Virtual Memory**

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

**An Inverted Page Table handles access to physical memory by taking a thread identifier, logical address, and offset, then searching the IPT for an entry matching the thread**

**identifier to get a virtual address, then using the virtual address plus the offset to get a physical page.**
**We wrote a new struct to implement our IPT. It stores a thread pointer and a virtual address. There is a global array (size 32) of these IPT structs. We use these when we get a page fault exception; the page fault exception occurs when a needed physical page is not currently loaded into main memory. To load a page into main memory, we need to know the virtual address to get it from and the physical address of where to put it. We find the corresponding IPT struct in the IPT array by iterating through it and comparing each IPT struct's thread pointer to the current thread. Once we find the IPT struct for the current thread, we now also know the virtual address.**
**To choose the physical address to load into, we need to choose a victim page (an already loaded page in main memory) to kick out. We have two ways of deciding the victim page - random and least recently used. Both methods result in a page number representing the index of the page to remove. Random choice is simple and just picks an index between 0 and 32 (number of physical pages). We have a list of ints, which acts a queue, to keep track of the least recently used main memory page number in a first-in, first-out fashion. To maintain this we basically add page numbers to the end of FIFO list when we use them, and choose the page number at the front of the list when we need to choose the least-recently-used victim frame.**

**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. Refer to the following table for an example of the data required:**

| Program | -rs seed | Replacement | Page Faults | Timer Ticks |
|---|---|---|---|---|
| tester | 159 | FIFO | 33 | 49453 |
| | 753 | FIFO | 33 | 48946 |
| | 159 | Random | 33 | 49453 |
| | 753 | Random | 33 | 48946 |
| tester2 | 147 | FIFO | 36 | 49099 |
| | 963 | FIFO | 36 | 49265 |
| | 147 | Random | 36 | 49099 |
| | 963 | Random | 36 | 49265 |

| | | | | |
|---|---|---|---|---|
| tester3 | 761 | FIFO | 57 | 77749 |
| | 167 | FIFO | 57 | 78185 |
| | 761 | Random | 57 | 77752 |
| | 167 | Random | 57 | 81288 |

The number of page faults as an effect of the replacement algorithm chosen did not vary significantly (at all, in fact). We expected the FIFO replacement method to have resulted in fewer page faults. This is because FIFO and other temporal-related replacement methods (like LRU) will keep frequently-accessed memory available in main memory, and should result in fewer page faults in programs with lots of loops and stuff. But since our numbers of page faults for random replacement and FIFO replacement were so similar, we think our test programs just happened to be structured in a way that didn't really benefit from FIFO.
The execution time, approximated by timer ticks, seems to have increased in proportion to the number of page faults. This is expected because larger programs will be managing more memory, and page faults/page replacement are load/store operations, which by nature take a lot of time compared to basic arithmetic operations..

Task 5: Two-Level Hierarchical Page Table

1. **Explain the structure of your two-level paging system and how it is used by your code.**

We used a 2-dimensional array as our hierarchical page table. The first dimension references the outer table and is initialized to four. The second dimension references the inner page tables. The size of the inner page table is determined by a ternary operator in which if the number of pages a process needs % 4 is equal to zero then the inner size is numPages / 4. If it isnt equal to zero then the innersize is numPages / 4 + 1. This allows enough space for pages to fit. We created a double nested for loop to initialize all the page tables and set all the valid bits to false that are needed within each inner page table. We created a 2 dimensional array of Translation entry inside of Machine.h as well to store our HPT(Hierarchical Page Table) within the kernel. Inside of translate cc we make a check to see if we are using HPT or Linear Paging (bool created in system.cc). After that we take the virtual address given and divide it by 4 to get the index of the outer table and use the address % 4 to get the offset within that. After that we set the entry equal to that index and continue as normal, throwing the page fault exception if the page is not valid. In our assign page function, we just set the physical page of the virtual page(found by using the same method in translate.cc) to the physical page we found if it turns out we are using HPT.

2. **Explain the changes that you needed to make to your Demand Paging scheme, and the IPT implemented in Task 4, if applicable, to implement your two-level paging system.**

There weren't any changes needed to be main within the IPT and Demand Paging scheme since the virtual page is still accessed from translate.cc The IPT still stores the physical page and the virtual address as well as the current process that controls that physical page.

Task 7: 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.**

Some of the biggest problems with this assignment was figuring out how to read from swap files during task 3 and how to update them as needed. We ended up realizing that we weren't passing in the right value for the position argument in the ReadAt() function. Another problem we had was during task 4 we were using a very inefficient way to tell if we had enough space to run a process or if we had to select eject frames from main memory. We were checking if the number of clear pages in memory was less than the number of pages needed. This is inefficient and we realized that it wouldn't work because we were trying to solve the exception that was being thrown instead of handling it. We solved this by handling the exception and swapping out pages as exceptions were thrown. We also ran into problems with how to implement two-level hierarchical page tables. It took us a while to realize the math we needed to get from the first page table to the second; this ended up being a division operator to get the first page table entry number, and a modulo operator to get the second page table entry number.

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.**

We wrote a struct for our inverted page table and an array of those structs in order to use them as needed for task 4. Also in task 4 we used a List as a way to hold page numbers in a First in/First out format. For task 5 we are using a two dimensional array of type TranslationEntry this functions as the two-level hierarchical page table.
For task 2 we wrote an algorithm to assign pages to main memory, it finds empty spaces in memory and fills them with processes that need to run. For task 4 we have an algorithm that uses a List to find the page in memory that has been there the longest. We thought about efficiency a little bit but generally we had an attitude that if it worked we should move instead of trying to improve speed or efficiency.