Jason Woodworth Edward Woods

Jww7675 ebw3559

Evan Perry Troy Stockman

Erp1067 tas6329

**CMPS 455 - Stage 4: Side Dish**

**Task 2: Demand Paging**

1. **Compare and contrast the changes made to *AddrSpace*** **for this task with the changes made for Stage 3: Multiprogramming**
   1. In Stage 3, changes were made only to the *AddrSpace* constructor, so that when the process was initially created it would load the entire size of the process (the executable) into main memory so that it could be run. This would load in every page for the process regardless of whether or not it was actually needed in order to run, and also made it necessary to have contiguous space available in order to load a process into memory.  
      In contrast, this implementation will only load a single page at a time into memory, and only when NachOS deems it necessary. As opposed to the previous implementation, an address space’s page table will initially set every page in its page table to false. Now whenever a page is needed, it will trigger a Page Fault Exception, which makes NachOS load whichever page is necessary into whatever free spot of memory it has, or swap out from another frame. This way, it eliminates the problem of external fragmentation and allows a process to run when not all of its pages are in memory at the same time.
2. **What steps do you take when a page fault occurs? Explain in detail.**
   1. When a page fault occurs, we get the virtual page address that caused the PageFaultException to occur from register 39. Then, we go to the address space, and call the Paging function. We compute the Virtual Page Number (vpn) by dividing the virtual page address by the PageSize. When doing Demand Paging, we first check to see if there is enough space to hold the page needed. If not, we just halt Nachos. If we are not doing Demand Paging, we will swap out a file based on the page replacement algorithm selected. If there is enough space, we find the next space available in the bitmap, and set that as the offset to the main memory. We mark this page in the bitmap, get the physical address of the page by multiplying the offset by Page Size, and then clear out a page of main memory. Since we have the Swap Files implemented, we will read into the main memory from the Swap File. The position in main memory to be read into is the physical address. We read a size equal to PageSize, with a position in the Swap File of noffH.code.inFileAddr + (virtual page number (vpn) \* PageSize). finally, we will set the valid bit in the page table with an offset of vpn to true, and update the physical page and virtual page with the offset and vpn, respectively.

**Task 3: Swap Files**

1. **How did you modify the ​ AddrSpace​ constructor for this task?**
   1. For this task, instead of creating the Swap File in the constructor, we created another function to handle this. The function is called CreateSwapFile, which is supposed to be called after the thread is given its AddrSpace. The threads ID is passed into the function, which is used to create the filename, in the form “../SwapFiles/threadID.swap”. Once the filename is created the filesystem creates the file, and then opens it. Once this is done, a character array is created to read everything from the executable, and then copy this to the Swap File.
2. **If you created any new classes or data structures, explain them. If you did not, say ‘so’.**
   1. None were created.

**Task 4: Virtual Memory**

1. **Explain the structure of an IPT and how it is used by your code.**
   1. The Inverted Page Table (or Core Map, as we call it in our code) is essentially an array of size equivalent to the number of physical pages in the system with each slot in the array holding access to the thread that is occupying it’s frame with information about which virtual page of it’s address space’s page table is in that physical page.  
      We used this in our code when the user has selected a virtual memory option. Whenever a page is set into a frame, the core map detects which physical frame it was put into and updates that index in its array to hold the current thread that is paging, as well as the virtual page number that just got stored into main memory. This way, NachOS will always know specifically which page is in which frame. When NachOS goes to swap a page out of memory, it will consult the IPT to determine which page was in the frame it is swapping out from/to and set it’s valid bit to false. This way, when that page is needed again NachOS will know that it is no longer in memory and will know to swap something out for it.
   2. **For these tests, we lowered NumPhysPages to 15**
   3. **A sample command is:**
      1. **./nachos -x “../test/sort” -V 1 -rs 515132**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Program** | **-rs seed** | **Replacement** | **Page Faults** | **Timer Ticks** |
| **sort** | **515132** | **FIFO** | **20** | **39273** |
| **sort** | **323251** | **FIFO** | **21** | **39133** |
| **sort** | **515132** | **Random** | **23** | **39450** |
| **sort** | **323251** | **Random** | **21** | **39234** |
| **matmult** | **515132** | **FIFO** | **N/A** | **N/A** |
| **matmult** | **323251** | **FIFO** | **N/A** | **N/A** |
| **matmult** | **515132** | **Random** | **183** | **2616821** |
| **matmult** | **323251** | **Random** | **105** | **772523** |
| **halt** | **515132** | **FIFO** | **3** | **176** |
| **halt** | **323251** | **FIFO** | **3** | **176** |
| **halt** | **515132** | **Random** | **3** | **176** |
| **halt** | **323251** | **Random** | **3** | **178** |

**Task 6: 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.**
   1. A major error we ran into was that NachOS would hang on a yield sys call, meaning that it would not update pcreg at all once a user program called yield. We eventually determined that this was caused by not doing correct bookkeeping in the paging function. We were setting the entry in the address space’s page table for the virtual page number that caused the fault to valid, but not updating that entry’s physical and virtual page number. Fixing that fixed the problem.
   2. When running a program like sort with Random page replacement, our project would run it just fine, performing everything the way it should. However, when using FIFO, odd things would happen, like the second join not working because the filename was read incorrectly (getting +./test/halt instead of ../test/halt).
   3. Our project would also sometimes skip a system call whenever it would run a program. For example, in ../test/testexec, our project would skip the first exec and move onto the next syscall. This still occurs.
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.**
   1. The only data structure we used in this project was for the inverted page table (core map) in task 4. We created a struct to represent a single core map entry, each entry having a pointer to the thread which occupies that entry in the core map, the virtual page number for the thread’s page table, the physical frame number that it’s occupying in the core map, and a counter that tells how old that entry is. The core map itself is just an array of these entries of size equivalent to the number of physical pages.  
      Speed played a big role in the decision to use this. A manual search through every page table would be incredibly computationally expensive, as opposed to this solution which is memory intensive.