Jason Woodworth Edward Woods

Jww7675 ebw3559

Evan Perry Troy Stockman

Erp1067 tas6329

**CMPS 455 - Stage 3: Main Course**

**Task 2: System Calls**

1. How do your system calls work? Provide a pseudocode algorithm for each and elaborate on any noteworthy steps or problematic areas.
   1. Exec:
      1. SC\_Exec:
         1. Get the file address from register(4) and read from the memory to get the filename
         2. Send filename to SExec to get the processId
         3. When done, write to register(2) the processId
      2. SExec(char\*):
         1. Create an OpenFile (executable) by calling fileSystem->Open on the filename
         2. If executable == NULL
            1. Display output and return -1 to processId.
         3. Create a new address space with the executable passed in.
         4. Create a new thread, set it’s address space to the new one, and fork it to processCreator
         5. Delete resources and return processId.
      3. ProcessCreator(int)
         1. Initialize the registers in the current address space
         2. Load page table registers in the current address space
         3. Call machine->Run()
   2. SC\_Join:
      1. List.cc
         1. List was modified to allow the creation of a dynamic list of processes.
         2. A get thread function was added that iterates through the linked list until it hits a thread that matches the passed ID, and then returns the thread that matches. This is used to find the specific child process to join when passed the process ID from arg1.
         3. A remove thread function was added to remove a thread from a list upon its exiting.
         4. A function was added to sever parent/child relationships from an exiting thread that may be a parent. The function iterates through the list, checking if the current thread’s parent’s ID is equal to that of the passed ID from the possible parent. If that relationship is found, the child’s parent is just set to NULL.
      2. SC\_Join:
         1. Turn off interrupts so that the process of going to sleep will not be interrupted.
         2. Assign a thread called child to the thread whose process ID is that of the variable in arg1. This means that the child will be whatever was executed on the Join call.
         3. Set the child’s parent to be the current thread, increase the current thread’s number of children, and set the child’s boolean “isAChild” variable to true.
         4. Save the user state and address space state of the thread and put it to sleep.
         5. Whenever it is returned from sleep by a child restoring its state and putting it back on the ready queue.
         6. Turn interrupts back on.
   3. SC\_Yield:
      1. Print letting the user know the process is yielding.
      2. CurrentThread->Yield()
   4. SC\_Exit:
      1. Call SExit with value in register(4), arg1
      2. SExit(int):
         1. Check if the thread has a parent, and if it does wake it up..
         2. Sever parent/child relations with the current thread with the List’s sever ties function.
         3. If status passed in = 0; Exited Normally & write to register(2) value = 0
         4. If status passed in = 1; Exited Abnormally & write to register(2) value = 1
         5. Otherwise, print whatever exit code was passed, & write to register(2) value = status
         6. After, delete current address space
         7. Finish current thread.

**Task 3: Address Space**

1. Explain how you manage the memory.
   1. Memory is managed through the main memory in machine and page table with an offset found by our memory allocation methods. The offset is measured in page sizes and represents where in the bitmap the program will start. The bitmap is marked for the number of pages the program takes up in full, and as such will not allow another program to overwrite its space.
   2. Once the offset is found and there is determined to be enough memory to run the process, the page table is assigned a physical address of (i + offset) in the for loop, offsetting it by the appropriate amount.
   3. The main memory is then set to zeroes starting at (offset \* PageSize) and going for the length of the process’s size.
   4. Finally, a page of data is read into memory at a time for the code and initialized data. it starts from the virtual address and adds the offset \* PageSize and the current page number \* PageSize.
   5. Note: We believe this to be the cause of at least some of our segfaults and problems, but could never properly figure out the correct math.
2. Describe your memory allocation and deallocation scheme.
   1. Allocation:
      1. There are three different allocation schemes: FirstFit(), BestFit(), and WorstFit()
      2. An offset is created, and set to the page the selected memory allocation scheme returns.
      3. Using a bitmap, we Mark pages in the bitmap for each page allocated if there are enough pages available to Mark.
   2. Deallocation:
      1. In the Address Space deconstructor, we deallocate the address space
      2. We loop from offset to (offset + numPages) and Clear the bits in the bitmap
      3. Finally, we delete the pageTable for that process.
3. How does NachOS start a user process? Walk through the algorithm step by step.
   1. First NachOS calls execute () to loads and executes the program by passing the file name.
   2. Next NachOS load the program into the address space.
   3. The thread is fork with the initialized processor's registers and address translation information
   4. Then NachOS calls Run () to start the user code.

**Task 4: Memory Allocation**

1. Did any particular memory allocation scheme prove more difficult to implement, debug, or test than the others? If so, why do you think this happened?
   1. For us, FirstFit() was the easiest one to create. Once we got that one down, WorstFit() was pretty easy to create as well. BestFit() probably was the one that gave us the hardest time to make, though it still wasn’t too bad. It took us a little while to wrap our heads around how to get the start and end for what the best size would be, but we figured it out fairly quickly, compared to everything else in this project.

**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. Our largest problem encountered during this project had to do with implementing Exec() with the address space. Implementing multi-programming and dealing with the memory was an incredible hassle. At first our code would infinitely loop on machine->OneInstruction(). We believe this to have been an error in zeroing out the memory, where we were starting from the very beginning of main memory for any process instead of starting at its spot in the memory. We since changed that, but it now seems to repeat exec’d processes ad infinitum. It also segfaults on some occasions when attempting to join.
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. In order to keep track of the pages being used by the memory, we used a bitmap object, where we would Mark and Clear bits based on memory being allocated and deallocated. We used the offset of the process to determine where to start Marking and Clearing bits.
   2. We used a LinkedList to create a Process Control Block (PCB). Our PCB is essentially a list of processes. The linked list was created with efficiency in mind as we believed it to the be the least memory intensive option for storing a list of threads.
   3. System Calls
      1. **(Exec)** In Exec we did not worry about speed or efficiency. The only data structure that was used was the PCB, into which we added the exec’d process.
      2. **(Join)** We used a LinkedList for our Join system call. We chose LinkedLists for Join and our PCB because LinkedLists are more memory efficient than Dynamically Allocated Arrays. They allow for easy insertion and deletion.
      3. **(Exit)** Exit did not require any advanced data structures or algorithms apart from thread creation. Exit outputs a message for the user describing the type of Exit which was achieved. Then Exit deallocates the memory occupied by the current thread and deletes the thread.
      4. **(Yield)** Yield did not require any advanced structures or algorithms. Yield only needs to yield the current process and write to the register that the yield was successful.
   4. **(Address Space)** In AddrSpace we used bitmap to help us keep track of how much memory is being allocated for each program. We check to see if there enough memory to run run the program. If there is not enough memory NachOS will print out a message and only stop that program from running. The closest thing to an algorithm used here was the process of repeatedly adding an offset in the correct places for main memory’s position.
   5. **(Memory Allocation)** In AddrSpace we implemented the Best, Worst, and First Fit. We used these fits to find the offset for the new program to run in NachOs. The best fit find the smallest amount of continuous memory. The worst fit find the largest continuous memory, And first will find the first space the program could fit.