Project 2: Pager – A Virtual Memory Manager

CST-315-99 (Operating Systems)

Cameron Verini

11/13/2022

**Project Description:**

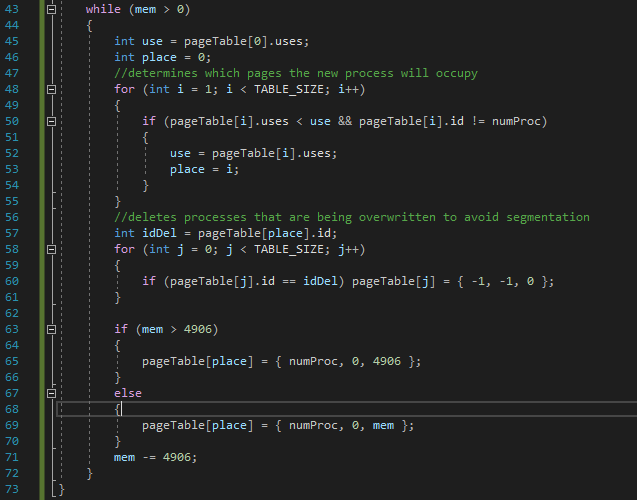
Within this project, we were tasked with updating our program from project 1 by adding in the building blocks for a virtual memory manager. This would demand the ability to manage memory resources, page tables, and page frames.

**Methodology/Approach:**

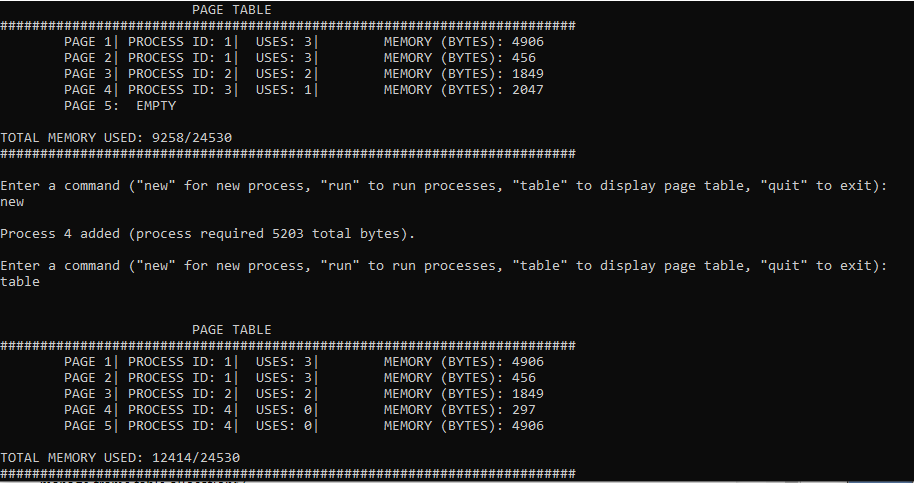
Within the program, an array was used to represent the page table. The array would specifically contain structs called “process”, which contain a variable for identification, a variable to keep track of how many times it has been run, and a variable that represents the total memory in bytes required for the process. The processes and page table can be interacted with via the newly implemented commands “new”, “table”, and “run”. The new command would create a new process and split it into pages within the page table, the table command would display the page table, and the run command would allow the user to run processes that are in the page table.

**11 Points:**

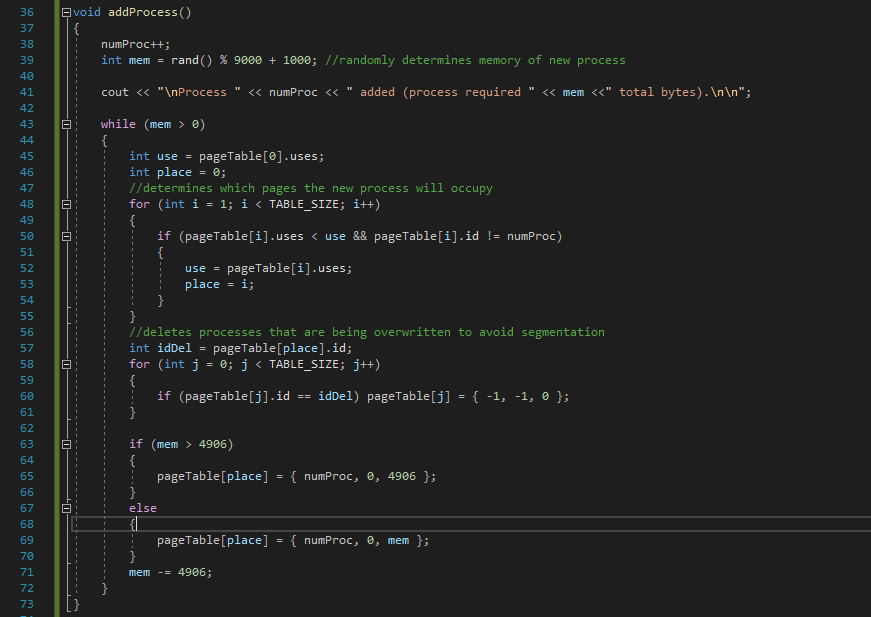
1. The size of the physical memory is 24530 bytes.
2. The size of the virtual memory in the project is also 24530 bytes.
3. The amount of memory that each process needs is randomly generated in the program so that the manager can really be tested. It has a range of bytes from 1000 to 9999.
4. Each page can store 4,906 bytes.
5. The amount of pages each process needs is based on the amount of memory randomly generated for each process. The program assigns each page 4,906 bytes of memory until the process has less than 4,906 bytes left to assign. It then assigns the rest to a page.



1. Within this program only one page table was used, though more could be managed by an address-translation scheme that utilizes nested tables.
2. Page frames are managed within the page tables. If there are empty pages, then they will be prioritized for taking in new pages. If there are no empty pages, then new pages will take the place of lower priority pages. Within this program, the processes that have been run the least (least “USES”) are considered to have lower priority. If one page that contains part of a program is overwritten, then all other pages that contain parts of the same program are emptied to avoid segmentation. If more than one page has shares the lowest priority, then the page that appears higher in the page table is replaced.



1. This program does not have any real memory allocation, because it simulates virtual memory management rather than actually doing it. Despite this, a general starting point is set in the addProcess() function:



1. Pages will be swapped between physical and virtual memory depending on their priority. As stated before, priority is determined by how commonly a process is run, with more runs leading to higher priority leading to the process being closer to physical memory to guarantee the quickest calls.
2. Segmentation faults are avoided by emptying pages that contain processes that have been partially replaced. A virtual lock can be used to avoid page faults.
3. This is currently handled by replacing pages with “empty” pages, but in an actual implementation could be done with the “free()” function that is included in C++.