Design Decisions

Jeremiah Thomas:4145047
Kyle Michaels:3914066

General Overview

The purpose of this project was to make nachos execute several user programs concurrently and implement a memory allocator that would safely map physical pages of memory to virtual pages for programs. This memory allocator also needed to correctly free pages when a program finished. To do this we created our own MemoryAllocator class that took care of all these requirments.

As described in the write up for pa3 we did not have to implement page swapping. This meant we had to limit when we loaded physical pages and had to make programs wait when they couldn't load. This process is described below in the section explaining the modification to the UserProcess class.

MemoryAllocator

This class keeps track of the available physical pages and handels allocating them and causing programs to wait when there is not enough pages. It also keeps track of the following information:
private int numReservedPages; number of current reserved pages
private int numPhysPages; number of total physical pages
private int numMapped; number of current physical pages mapped
private int maxNumReserved; max number of concurrently reseved pages
private int maxNumMapped; max number of concurrently mapped pages

We used a LinkedList of Integers that stores the available physical page numbers. Upon creation all physical page numbers are loaded into the list. Along with the list of physical page numbers we used a Semaphore for the number of free physical pages so if there were no more free physical pages when a program tries to allocate that program would wait in the Semaphore waiting queue.

When a page is allocated the freeMemory Semaphore is decremented and the first item in the freePages list is removed and returned as the allocated physical page number. When a program frees a page then the freeMemory Semaphore is incremented and the physical page number given by the argument ppn is added back into the List of free physical pages.

The number of reserved pages is the total number of pages (not only mapped) for all loaded programs. This is used to determine whether or not to load another program. If the number of unreserved pages is less than the number of pages for a program that program enters the waiting queue of the waiting Semaphore. Pages are unreserved when all the physical pages for a program have been freed.

Anytime the freePages Semaphore and the List of freePages are accessed we use a Mutex Lock to protect shared data. All UserProcess use a static instance of

the MemoryAllocator.

Data Structure:
LinkedList - We used a Linked List to keep track of the free physical page numbers because allocation does not depend on the order of the pages therefore a LinkedList is the most time efficient (O(1)) to allocate and free a single page.

UserProcess

In UserProcess we modified how virtual and physical pages numbers are mapped. Orginally pages virtual page numbers were mapped to the same physical page number. Our implementation gets the physical page number from a static MemoryAllocator which is the class we created and described above.

We also modified the UserProcess to keep track of the following statistics:
protected static int totalProcesses;
protected static int maxProcesses;

totalProcesses indicates the current number of concurrently running programs
maxProcesses indicates the max number of concurrently running programs

Physical page numbers are mapped either when the program is loading or when a page fault occurs. A page fault will occur anytime the program writes to a new physical page. There will never be a page fault when reading because we are not swapping physical pages.

We also added a signal handler for syscallExit which frees all pages mapped by the program and decrements the total number of running programs. If that number reaches 0 and the kernel has loaded all user programs then the machine is told it can halt. At this point all system statistics are printed to stdout. Also when a program exits it signals the waiting queue in MemoryAllocator that a program may be able to load because it has freed its pages.

Because we did not implement swapping, which was expressed in the assignment writeup, we added safety checks to avoid deadlocks. When a program is being loaded if the number of unreserved pages, maintained by the static MemoryAllocator, is less than the total number of pages for that program, the program is rejected and a message is printed to stdout.

UserKernel

In UserKernel we modified the run method to run all the programs given by Machine.getShellProgramNames() which gives the programs listed in the config file under the Kernel.shellPrograms field. We also added a process.readtToExit() call that tells the machine it can halt when there are no more programs running.

Processor

We modified Processor so that it set the pageSize to the specificed value in the config file used when run. The field in that sets the page size in the config file is:
Processor.pageSize

We modified this value and show how the pageSize affects the number of pages allocated and how many programs can run at a time in the Performance Analysis at the end of this document.

Machine

We added the method getShellProgramNames. This method parses the config file for the field Kernel.shellPrograms and creates a list of strings that represent the names of all the programs that will attempt to execute.

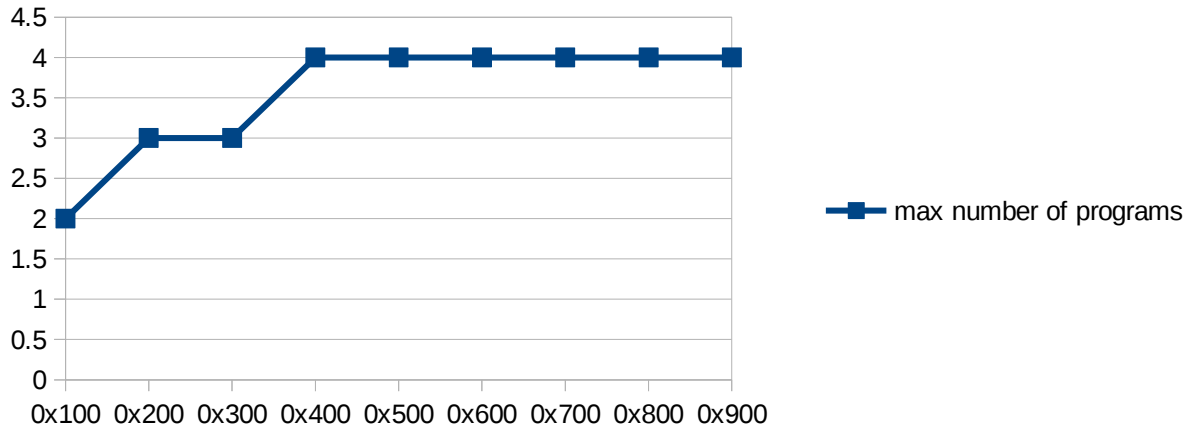
Modifications to Nachos

UserProcess, UserKernel, Processor and Machine were all modified as described above.

Performance Analysis:

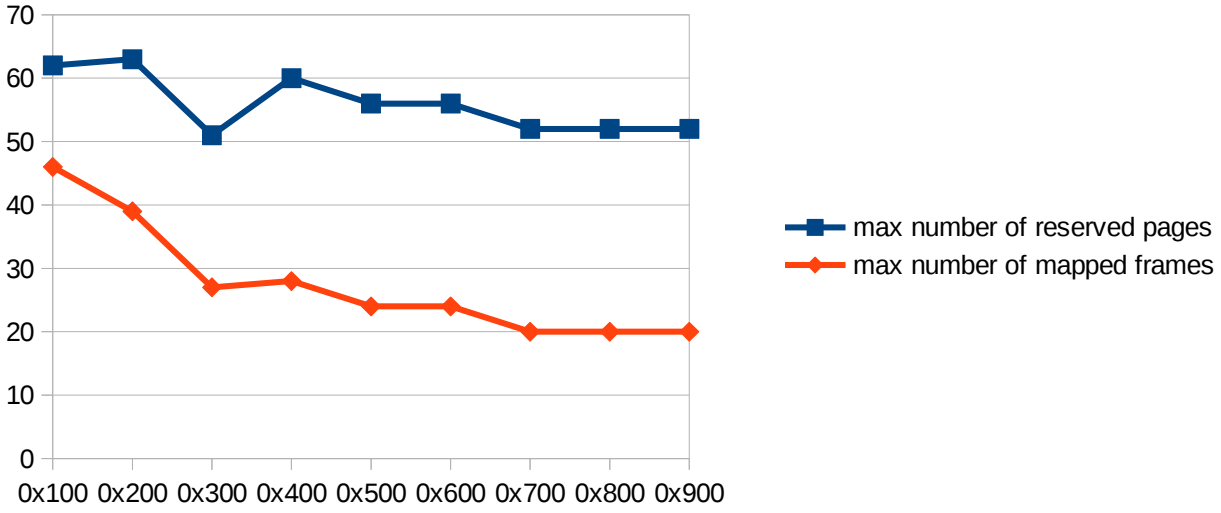
To test we ran 5 copies of safematmult.coff at 9 different page size values to determine the affects of page size on various statistics. All tests with varying pageSize have 64 physical pages.

PageSize vs number of concurrently programs



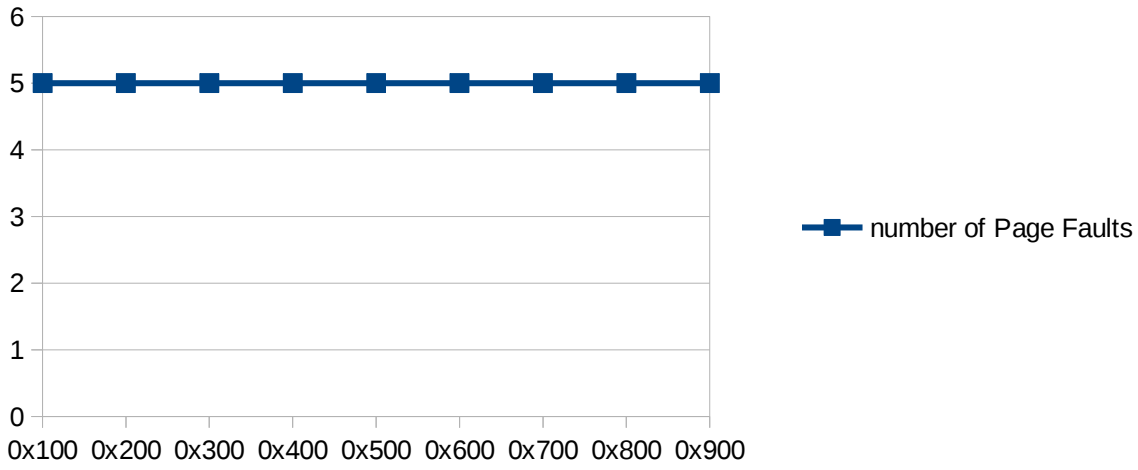
as the pageSize increased we noticed an increase in number of concurrently running programs. We believe this is because it now takes longer to load pages so there are more context switches per pages loaded.

pageSize vs number of reserved pages and pageSize vs number of mapped pages



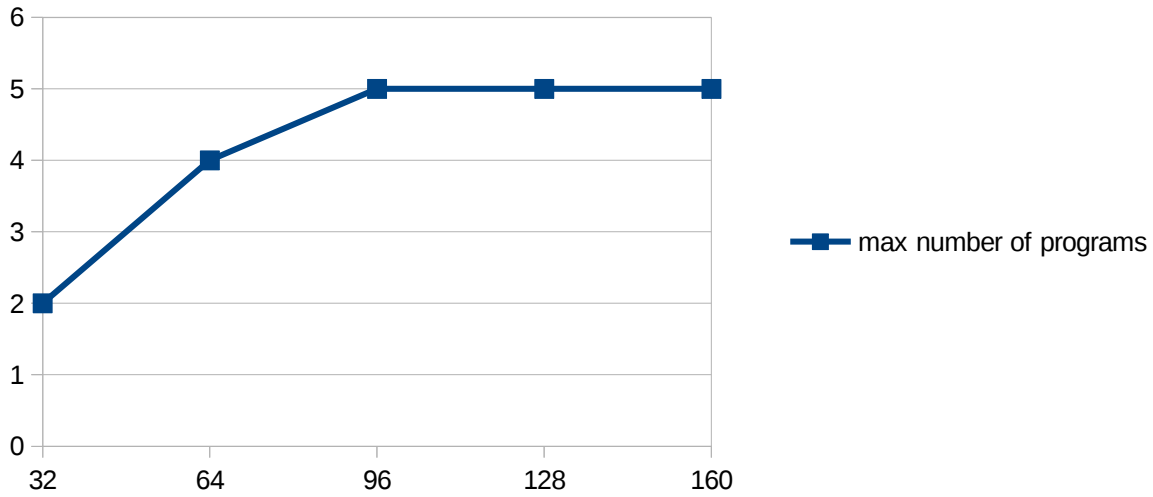
As the page size increased the number of reserved and mapped pages decreases because programs need less pages because they can store more per page. This affects mapped pages more in the beginning because percent increase is smaller at each increase because pageSize increases linearly. The amount of internal fragmentation appears to increase as pageSize increases because the number of reserved pages stays fairly constant but as pageSize increases the number of mapped pages decreases. We believe the program has to reserve a set number of pages per section regardless of page size but programs need less mapped pages because they can store more data into each page.

pageSize vs number of page faults



The number of page faults was constant during our tests. We believe this is because safematmult is a very memory efficient program. We anticipated that the number of page faults would decrease as pageSize increased because programs would need less mapped pages and pageSize increased.

Number of physical pages vs number of concurrent programs



for this test we changed the number of physical pages and kept the pageSize at 1024 bytes. All previous tests ran at 64 physical pages. From this data we see that as the number of pages increases the max number of concurrent programs also increases. However we are only running 5 so the max is 5.