Project 3: Multiprogramming and Virtual Memory

(Group Project; collaboration between groups is forbidden)

Assigned: October 11, 2021
Design Summary Due: October 16, 2021 (via Moodle)
Project Due: November 04, 2021 (via Moodle)

This project is worth 15% of the overall weight for this course.

Learning Outcomes and Objectives

After successful completion of this assignment, students will be able to:

1. Understand usage of address space, manage memory, and implement memory allocation schemes to allow concurrent user programs to exist.
2. Implement swap files and allow physical pages to be swapped to and from the disk.
3. Implement two page replacement techniques.
4. Write and test complex code with custom test programs.
5. Communicate with the instructor and teaching assistants to seek a greater understanding and clarification on topics they need support with.

Choice of an OS for Implementation

This project can be implemented in an OS of a student's choice including UC Berkeley's NachOS (C/C++), UC Berkeley's NachOS (Java), Stanford University's PintOS (C), simulated Linux or Unix or any other operating system.

Only UC Berkeley's NachOS (C/C++) version is supported, and its documentation is provided in Moodle which includes Salsa, commented source code, and supportive videos. The project specification is written with references to NachOS but can be used with any OS to implement equivalent functionality.

Note: Students who can successfully implement their solution correctly in an OS other than NachOS (C/C++) will be given bonus credits of up to 20% of this project's weight. However, no credit will be given if their solutions are incomplete or incorrect. An OS is a very involved codebase and students are cautioned to choose wisely.

Student Outcomes Assessed for ABET Accreditation

Student Outcomes (SOs) Assessed
SO 1. Analyze a complex computing problem and to apply principles of computing and other relevant disciplines to identify solutions.
SO 5. Function effectively as a member or leader of a team engaged in activities appropriate to the program's discipline.
SO 6. Be proficient in more than one programming language on more than one computing platform

Detailed Instructions on This Project

This third stage of the NachOS project will help you understand multiprogramming, virtual memory, and dynamic page allocation through implementation of the following tasks:

1. Design Summary
2. Address Space
3. Demand Paging
4. Swap Files
5. Virtual Memory
6. Output
7. Report

This document will give a detailed specification of the above tasks. In addition, it will provide design and implementation hints as preparation for further work. The focus of this assignment will be enabling concurrent user programs and creating a virtual memory system.

Before you begin, it is strongly advised that you read through the NachOS material posted on Moodle. Also read and understand the files pertaining to user programs and memory allocation, especially *exception.cc*, *progtest.cc*, and *addrspace.h/.cc*.

Note: This is a group project and the only allowed collaboration is among the members of a group. Inter-group collaboration is prohibited. Any other form of collaboration is prohibited as well.

Task 1: Design Summary

Summary

Before doing any work on this project, your group must write a design summary that outlines in broad terms the steps you intend to take in order to complete this assignment. Cover all tasks briefly, except this one.

Procedure

The design summary should be typed, with the names and ULIDs of all group members at the top. Break the summary into sections, one for each task, and describe your plan to solve each task as described below. Sample code snippets may be used but are not required; plain English and pseudocode algorithms are acceptable.

For **Task 2** (Multiprogramming), the changes you intend to make to the *AddrSpace* constructor to manage memory for multiple user processes. Discuss any problems you anticipate.

For **Task 3** (Demand Paging), describe the changes you intend to make to the *AddrSpace* constructor in order to accommodate demand paging. Also show the flow of execution within NachOS, in terms of files or objects affected, when a page fault occurs. Discuss any problems you anticipate.

For **Task 4** (Swap Files), describe the planned when, where, and how of swap file creation, how you plan to keep track of both the stack and data, and how you plan to clean up swap files after the associated user process is finished. Discuss any problems you anticipate.

For **Task 5** (Virtual Memory), describe the specified behavior of the various algorithms you must implement and how you plan to choose them based on a command line option. Also list any data structures you expect to use in order to implement these algorithms. Discuss any problems you anticipate.

In addition, if your group plans to use git for version control, include confirmation that you have been able to install git and set up a project repository. If you do not plan to use git, include justification and a plan for how you will work together to merge code.

The total length of the design summary should not exceed 2 to 3 pages. Spacing, margins, font, etc. is irrelevant within reason. Files should be submitted in *.txt*, *.pdf*, *.odt*, or *.doc* format.

Completed design summaries should be submitted on Moodle. Design summaries will receive a simple satisfactory/unsatisfactory grade meant to reflect whether or not your group is on the right track, possibly with additional feedback; your group should only proceed with the rest of the assignment once the design summary has been judged satisfactory.

Task 2: Multiprogramming

Summary

Currently, NachOS only supports one user process at a time. You must modify the *AddrSpace* constructor so that multiple user processes can run concurrently.

Procedure

The current implementation of the address space works by assigning a user process a contiguous block at the start of physical memory. It also zeroes out the entire physical memory when a new user process wants to run. Therefore, every *Exec()*call will cause the old user process to be destroyed before the new one takes its place. To accommodate concurrent user processes, you must make several changes.

The *AddrSpace* class uses a *pageTable*, which is simply an array where each element represents a page of the user process's memory. Each element of this array has a *virtualAddr* and a *physicalAddr* field, in addition to other fields such as *valid* that you do not need to worry about for this task.

Modify the *AddrSpace* constructor so that *pageTable* is not simply allocated physical pages starting from 0. You need to use a global map of some sort to keep track of which pages are available. By checking this global map first, you can assign physical pages to *pageTable* without worrying if another process is using them. The *Bitmap* class is excellent for this purpose and already exists within NachOS.

Once you know which pages the process will be using, you should zero out those pages and only those pages. You also need to change the method by which NachOS copies the code and data into memory, as it currently assumes the virtual page is the same as the physical page. When a process is finished, it must properly deallocate the memory it was using so that another process can use it. You may add any additional functions and data members you wish to help facilitate your changes.

In addition to *addrspace.cc*, you may need to modify the *StartProcess()* function, located in *progtest.cc*, so that it can properly start a new process.

If there is insufficient memory for a user program to run, display an error message and terminate that process. Do not terminate any other process, especially those that are already running, or NachOS itself.

Report

As part of your report, answer the following questions about Task 3:

1. Explain how you manage the memory.
2. Describe your memory allocation and deallocation scheme.
3. How does NachOS start a user process? Walk through the algorithm step by step.

Task 3: Demand Paging

Summary

For this task, you must implement demand paging in NachOS by forcing page faults. Demand paging is a memory scheme that loads pages into memory only when necessary, i.e. when a user process attempts to access a page that is not yet loaded into memory. If space does not exist (no free page is available), simply terminate the process.

Procedure

Unlike in the current implementation, you should not immediately load anything into memory when running a new user process. That is, comment out the original "executable->ReadAt()" code inside of *addrspace.cc* at the bottom of the constructor. Instead, in the *AddrSpace* constructor, set the *valid* bit of each entry in *pageTable[]* to *false*. From this point on, when a user process tries to run code from a page where the *valid* bit is *false*, it will generate a *PageFaultException* from *exception.cc*, and the NachOS kernel will react accordingly. You must write code to handle this exception and load a single page as it's needed.

More specifically, you must read register *BadVAddrReg*. *BadVAddrReg* is a NachOS system variable that stores the virtual location (in bytes) inside of the user program that threw the *PageFaultException*. From here, you can translate this location in bytes to a virtual page number using *PageSize*. Then, find the first available free page (no need for special memory allocation) and update the currentThread's *pageTable[].physicalPage* **and** valid bit to true. If no free pages exist, print some error and terminate NachOS (this will not be the case in later tasks!). Finally, if space was found, load the **single** page into mainMemory using ReadAt().

For testing purposes, you may experiment with *NumPhysPages*, which represents the number of physical pages. This is a constant variable located in *machine/machine.h*. If you choose to modify this variable for testing, then remember to revert it back to the default value of 32 before you submit.

Once you have finished this task, try loading multiple user processes. Use a custom test program that calls *Exec()* on itself several times in order to get multiple user processes running at once, or run shells inside of shells. Remember to use the *-rs* option as well. Once everything seems to be in order, proceed to Task 3.

Report

As part of your report, answer the following questions about Task 2:

1. Compare and contrast the changes made to *AddrSpace* for this task with the changes made for Task 2.

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

Task 4: Swap Files

Summary

Implement a physical swap file system for virtual memory in NachOS. Instead of loading directly from the executable, you must use this swap file instead. Then, delete the swap file after a process exits or terminates.

Procedure

When memory is full, pages will need to be swapped out of memory in order to make room for incoming pages. However, outgoing pages cannot simply be swapped back to the original executable. Not only will the size of the executable have to be extended, but it may inadvertently wipe out some of the code as well. There must also be enough swap space for all currently loaded threads. You may create any new classes or data structures you wish in order to facilitate your implementation.

Each user process must have its own swap file. A swap file is a physical file with the name *ID.swap* where *ID* is a unique thread identifier of some sort, such as a thread ID number (you should see them filling up the */userprog/* folder). Create this swap file when you initialize a program's address space.

The swap file must be large enough for all memory required by the user process. This includes code, initialized data, uninitialized data, and the stack. (**HINT:** Refer to how our executable loads into memory previously. Mimic this behavior and load both code and initdata into the swap file instead of main memory.)

When calling *Exec()*, the *valid* bits in *pageTable[]* should still be set to false (*demand paging*). After setting up the *pageTable*, copy the code and data from the executable to the swap file instead of into memory. The uninit data and stack space in the swap file should be filled with zeroes.

Delete the swap file when the associated user process terminates as it will fill up the *userprog* directory.

Do not use the executable itself as a swap file under any circumstances.

Follow the testing procedure outlined in Task 3 once you are finished implementing swap files. After running a shell, you should see a '0.swap' in the file browser. Run another shell and you should see '1.swap'. Etc.

Report

As part of your report, answer the following questions about Task 3:

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

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

Task 5: Virtual Memory

Summary

Implement the Random and First-In First-Out (FIFO) algorithms for dynamic page replacement once memory becomes full. These algorithms should be selectable via a new *-V* command line option. In order for virtual memory to work properly, you must also implement an Inverted Page Table (IPT) to find running (or sleeping) threads to pull out of main memory.

Procedure

Implement a new command line option, *-V*. It should take a single integer parameter, 0, 1, or 2. The *-V* option itself should be optional. That is, the following commands should both work:

./nachos -x ../test/PROGRAM_NAME

./nachos -x ../test/PROGRAM_NAME -V 1

The parameter values correspond to the algorithms as follows:

Command Line	***Algorithm***
./nachos -x ../test/PROGRAM_NAME -V 0	Disable Virtual Memory (Demand Paging Only)
./nachos -x ../test/PROGRAM_NAME -V 1	FIFO Page Replacement
./nachos -x ../test/PROGRAM_NAME -V 2	Random Page Replacement

In the event that the *-V* option is not used, or if the parameter is somehow invalid, NachOS should default to demand paging. If memory becomes full using demand paging, simply terminate the process trying to execute. In all cases, display output indicating what memory scheme is being used.

FIFO selects the first page brought in as the page to be swapped out. In other words, the page that has been in memory the longest is the one chosen for removal. (**HINT:** The *List* class is a FIFO object.)

Random selects a random page to swap out. Pick a random number from 0 to *NumPhysPages - 1* and remove that page.

An IPT is an array of size *NumPhysPages* where each entry is a thread pointer, the threads each being a user process. Once you determine which page needs to be replaced, consult the IPT to determine which thread is using that page. (NachOS does not allow threads to share pages; only one thread is using any given page.) From here, you can handle the process of swapping out the page and doing the necessary housekeeping on the thread in question. Note that the nature of the IPT and the data it represents means that there is only one IPT, which is visible to all threads.

Report

As part of your report, answer the following questions about Task 4:

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

2. 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
sort	R1	FIFO	XXX	YYY
	R2	FIFO	XXX	YYY
	R1	Random	XXX	YYY
	R2	Random	XXX	YYY
matmult	...	...	...	...
halt	...	...	...	...

Task 6: Output

Summary

You must include additional output reflecting the number of pages, page size, and page replacement algorithm being used. In addition, a new *-E* command line option should enable extra output similar to the debug option -d.

Procedure

By default, your code should produce output when NachOS first starts running that provides the following information (along with previously defined outputs):

1. Number of physical pages.

2. Page size.

3. Page replacement algorithm, including if chosen via *-V* or by default.

In addition to the above, the user should have the option of selecting a new command line option, *-E*. It should take no parameters and enable the production of extra output. This extra output should take place whenever a page fault occurs. If there is a free page available and no swapping is required, then notify the user of a page fault and note the process ID, requested virtual page, and assigned physical page. If there is no free page and a page swap is required, then notify the user of a page fault and note both

process IDs, requested virtual page, assigned physical page, and the virtual page number of the page being removed. Refer to the following for an example of how the output might look. Bolded output should only be visible when *-E* is used.

> *./nachos -x ../test/PROGRAM_NAME -V 1 -E -rs 1234*
>
> ...
>
> Number of physical pages: 32
>
> Page size: <NUM>
>
> Page replacement algorithm: FIFO
>
> ...
>
> **Page fault: Process <PNUM> requests virtual page X.**
>
> **Assigning physical page Y.**
>
> ...
>
> **Page fault: Process <PNUM> requests virtual page X.**
>
> **Swap out physical page Y from process <ONUM>.**
>
> **Virtual page Z removed.**

Task 7: Testing

Summary

Once you are finished implementing the rest of this assignment, you should run a wide variety of tests to make sure that everything works properly. You do not need to submit anything in particular to prove that you completed this task, but your submission will be subjected to a large suite of custom built test programs designed to expose any flaws in your implementation.

Procedure

NachOS provides a few user programs in the *code/test*directory that you can use to get started with your own testing. The ones that are best suited for this assignment are as follows:

- *matmult*: Uses 25 to 26 pages of memory, but does not take long to run.
- *sort*: Uses less memory than *matmult*, but takes longer to run.
- *halt*: Calls the *Halt()*system call.

These programs, while a good place to start, are insufficient for a thorough testing. You should use them as a starting point to create your own test files that push your code to its limits. In addition to making sure that your code handles the given tests, it should also catch and deal with the following:

- Programs that call Exec() and Join() on other programs
- Programs that take up more physical pages than available

- Programs with infinite recursion
- Exec() calls with bad arguments

Task 8: Report

Summary

You must turn in a report on this assignment along with your code. In addition to the questions listed under each task, the report should answer the following:

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.

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.

Procedure

Your report should be in *.pdf*, *.txt*, *.doc*, or *.odt* format. Other formats are acceptable, but you run the risk of the TA being unable to open or read it. Such reports will receive 0 points with no opportunity for resubmission.

Your name and ULID must be clearly visible. For group assignments, include the name and ULID of all group members. The questions in the report should be arranged by their associated task, then numbered. There is no minimum length, although insufficient detail in your answers will result in a penalty.

Hints

This assignment builds on another code base that has already implemented Syscalls for you. You can download it from Moodle, and should review the *exception.cc* file for any changes from your base Nachos. You may choose to implement the Syscalls yourself if it will help you understand Nachos more thoroughly, but should keep the project timeline in mind if choosing to do so.

Your code for this project will mostly be located in the *userprog* directory. You will need to modify *exception.cc*, *addrspace.cc*, and *progtest.cc*. For command line arguments, continue to modify */threads/system.cc/.h*. You will also probably end up having to modify *thread.h/.cc*, located in the *threads* directory in order to add thread IDs.

This assignment is historically the one that students struggle the most with. It is also worth the most points out of any assignment. Start your work early and make sure to ask questions if you are uncertain about what is going on. Completion will take a full understanding of the Nachos system, so please be sure to read through to see how user programs work before coding.

Unlike previous assignments, these tasks are highly intertwined and will require group effort. Coordination is key to avoid unnecessary mistakes and delays. Use of version control software such as git may help when merging collaborations on the same file.

This project asks you to use built in parts of Nachos you may be unfamiliar with (e.g. the Bitmap or List classes, the Machine->mainmemory array, the Nachos file system). It may benefit you to spend time in the beginning experimenting with these structures to understand how they work and can be used to your

advantage. Questions like “How do I create a file in Nachos” or “What is contained in main memory” may be answered by clever experimentation and reading the docs.

To run a user program for testing, use the following command:

./nachos -x ../test/PROGRAM_NAME -V X

Where *X* is 0, 1, or 2 to select between demand paging, FIFO, and Random virtual memory algorithms. The *-V* option in general should be optional, defaulting to demand paging if it is not used or if *X* is an invalid value. Note that the user program should be the executable, not the source code. For example, you would run *matmult*, not *matmult.c*.

When testing your code, make sure it works with the *-rs* option. *-rs* is a command line option that takes a positive integer as a seed and uses it to force the current thread to yield the CPU at random times. You can use *-rs* like so:

./nachos -x ../test/PROGRAM_NAME -V X -rs 1234

Use a wide variety of *-rs* seeds to ensure your code works properly. A badly coded assignment can break if *-rs* forces a thread to yield at an inopportune time.

When a user process terminates, you must delete the associated swap file. However, there are circumstances where it is impossible to delete all swap files, such as when NachOS is forcibly terminated with *CTRL + C*. If these junk swap files prove to be a problem during testing, you can delete them from the Linux terminal using:

*rm -rf *.swap*.

Your code should gracefully handle any errors that crop up. Graceful handling means not letting NachOS crash or segfault, but instead catching and managing the error so that, at the very least, NachOS exits normally. Any crashes, segfaults, etc. will result in a penalty, no matter the cause, so it is in your best interest to avoid them.

Although NachOS is written in C++, it doesn’t play very well with the C Standard Library, most notably data containers like *vector*. Although you may attempt to integrate it into NachOS, it’s generally not worth the hassle and we will be unable to assist you if you run into any difficulty. It may be helpful to think of NachOS as C code with some C++ sugar on top, instead of straight C++.

Submission

Throughout your code, use comments to indicate which sections of code have been worked on by which student. For example:

//Begin code changes by Name 1.
...
//End code changes by Name 1.

Inside the *nachos-3.4* folder, create a subfolder called *submission_documents*. Place your report inside this folder. In addition, put a copy of all files you created or modified for this project into this folder. Tar and gzip together the *nachos-3.4* and *gnu-decstation-ultrix* folders with the following command:

tar -czvf project03_ULID_nachos.tar.gz ./nachos-3.4/ ./gnu-decstation-ultrix

Submit the resulting *tar.gz* archive on Moodle. When unpacked, it should be ready to run with no setup required. Ensure that you run 'make' before uploading, to commit any final changes to your code.

For group submissions, make sure to include the ULID of all students involved. Only one student per group should submit. Do not submit a paper copy. Any paper copies will be shredded into confetti and dumped over the student at a random point in the semester.

Late and improper submissions will receive a maximum of 50% credit.

Good luck!