# NachOS Project Stage 3
# Main Course: Multiprogramming
# Total Points: 390

**Assigned: February 23, 2016**
**Design Summary: February 25, 2016**
**Due: March 15, 2016**

## Objectives

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

1. Implement basic system calls necessary to run concurrent user programs.
2. Change the memory allocation scheme to allow concurrent user programs to exist.

## Overview

This third stage of the NachOS project will help you understand multiprogramming and system calls through implementation of the following tasks:

1. Design Summary
2. System Calls
3. Address Space
4. Memory Allocation
5. Testing
6. 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 the implementation of system calls and concurrent user processes.

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*.

---

## 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. You should describe your intended solutions to Tasks 2, 3, 4, and 5. The intent of the design summary is to ensure that students are on the right track with their intended solutions, given the complexity of the assignment.

**Procedure**

The design summary should be typed, with the names and CLIDs 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 (System Calls), describe what you believe each system call to do and how you intend to code it to support the specified behavior. If you can think of any, also include potential pitfalls or additional support you may need to add.

For Task 3 (Address Space), describe how you intend to manage the memory with multiple user processes. Again, if you can think of any, include any difficulties you foresee.

For Task 4 (Memory Allocation), describe the specified behavior of the various algorithms you must implement and how you plan to choose them based on a command line option.

For Task 5 (Testing), describe the sort of tests you intend to subject your solution to, and the purpose each test will serve.

The total length of the design summary should not exceed 2 to 3 pages. Spacing, margins, font, etc. are 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: System Calls

**Summary**

For this task, you must implement 4 system calls. A system call is a function that a user level process calls in order to access kernel level functions. The system calls you will implement for this task are *Exec()*, *Join()*, *Exit()*, and *Yield()*.

*Exec()* will create and run a new user process. *Join()* will make a child process, run it, and put the parent to sleep. When the child process finishes, it wakes the parent up. *Exit()* terminates the user process and notes if it was done normally or abnormally. *Yield()* makes the user process yield the CPU.

**Procedure**

Your work for this task will be done in *exception.cc*. Note that the *exception.cc* that comes with NachOS is almost completely barren. We have provided a more solid framework for you to work with. Locate and download the *exception.cc* file on Moodle and copy it into the *code/userprog* directory, overwriting the old version.

You will find that there exist other system calls beyond the 4 you must implement. Some, such as *Read()*, *Write()*, and *Halt()*, have been implemented for you. Others, such as *Create()*, *Open()*, *Close()*, and *Fork()*, are blank. (Do not confuse the *Fork()* system call with *Thread::Fork()*.) You do not need to concern yourself with any of the already implemented or blank system calls, only the 4 specified above in the task summary.

All system calls must produce output. The output should consist of the name of the system call (Exec, Join, Exit, or Yield), the ID of the thread that called it, and any other relevant identifiers, such as the exit code for *Exit()* or the second thread ID for *Join()*.

If you are having difficulty understanding the code flow, study the relationship between *exception.cc*, *syscall.h*, *start.s*, and the various user level programs located in the *code/test* directory. Further information about the intended behavior of the system calls can be found in the Salsa document on Moodle, under “System Calls and Exception Handling.”

**Report**

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

1. How do your system calls work? Provide a pseudocode algorithm for each and elaborate on any noteworthy steps or problematic areas.

## Task 3: Address Space

**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 assignment.

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 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.

If a user process tries to access a pointer that's outside its memory, NachOS normally enters an infinite loop of *AddressExceptionError* messages. Change this so that instead it says "Pointer out of bounds" and terminates the offending thread. Other threads and NachOS itself should continue to run normally.

**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 4: Memory Allocation

**Summary**

For this task, you must implement three different methods of allocating memory for a user process: first fit, best fit, and worst fit. These methods will be selected via the *-M* command line option.

**Procedure**

Implement a new command line option, *-M*. It should take a single integer parameter, 1, 2, or 3, that will correspond to first fit, best fit, and worst fit, respectively. The *-M* option itself should be optional. That is, the following commands should both work:

```
./nachos -x ../test/PROGRAM_NAME
./nachos -x ../test/PROGRAM_NAME -M 2
```

In the event that the *-M* option is not used, or if the parameter is somehow invalid, NachOS should default to the first fit algorithm.

All of the algorithms involve searching through memory to find all contiguous blocks of memory that are big enough to hold the user process. First fit then selects the first such block it found, best fit selects the smallest, and worst fit selects the biggest.

To search through the memory, you may wish to add a search function to the *Bitmap* class, or you can do it from within *AddrSpace*. Either option is acceptable.

Your code should produce output when NachOS first starts running that tells the user which memory allocation scheme is being used. If it defaulted due to a wrong or nonexistent value, say so.

**Report**

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

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?

---

## Task 5: 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 can handle proper system calls, it should also catch and deal with the following:

- Invalid file names for *Exec()*.
- Non-executable files for *Exec()*.
- Bad SpaceID for *Join()*.
- Unprogrammed system calls.
- Infinite recursion.

These examples are not an exhaustive list, but are meant to illustrate the variety of situations that can crop up when attempting to run arbitrary user programs.

You will also need to modify *makefile* to accommodate any custom test files you make.

---

## Task 6: 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 CLID must be clearly visible. For group assignments, include the name and CLID 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

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*. You will also probably end up having to modify *thread.h/.cc*, located in the *threads* directory.

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.

Unlike previous assignments, the two main tasks are highly intertwined. In other words, you cannot truly finish Task 2 or 3 without working on the other somewhat. In order to test the system calls, you need support for concurrent processes, and in order to test concurrent processes, you need to run some system calls. Coordination is key to avoid unnecessary mistakes and delays.

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

*./nachos -x ../test/PROGRAM_NAME -M X*

Where *X* is 1, 2, or 3 to select between the first fit, best fit, and worst fit memory allocation schemes. The *-M* option in general should be optional, defaulting to first fit 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 implementing *Join()*, remember that the child thread must be able to wake up its parent once it's done running. To this end, you must somehow record this parent/child relationship and make sure the child can access its parent. You will likely need to extend the *Thread* class and assign each thread a unique identifier in order to accomplish this.

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 -M 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.

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++.

Due to the complexity of NachOS in general, you may wish to set up a version control system so that you can readily revert to a previous version if you accidentally break your code. However, this is not a requirement at any stage.

---

## 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 Devin Rooney.
...
//End code changes by Devin Rooney.
```

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_CLID_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.

For group submissions, make sure to include the CLID 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.