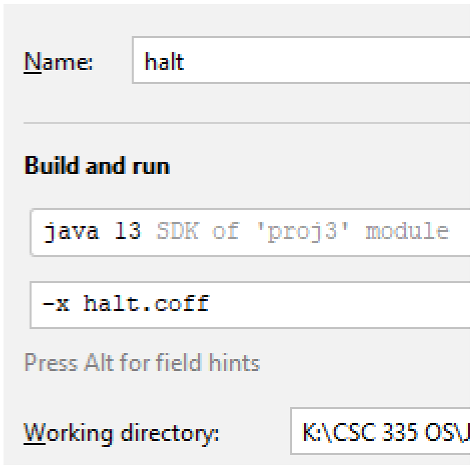
CSC 335: Project 3 – Run that Code
Due Friday, June 7, 2024 at 10:30am EDT
(Gradescope portion due Wed, May 29, at 10:30am EDT)

Objectives

- To understand the job of the **memory manager** in an OS
- To implement **system calls** like Read, Write, and Exit
- To understand and implement a **paging** system
- To learn a lot about how Nachos deals with user processes
- To finally allow user programs to run

Your Mission
Up until now, everything you have done has been inside the kernel -- running tests to make sure concurrency works. It is now time to allow user programs to run. Let's get you set up.

1. Make a copy of your previous project so that, once again, you have a working starting point that you can come back to if need be.
2. In the **ThreadedKernel** class, comment out the calls to all the concurrency tests from the last two projects. We don't need them to run anymore.
3. Change the working directory in your run configuration to use the "proj2" version of nachos.conf.
4. Some sample user programs are stored in the *test* directory in Nachos. Take a look at halt.c, which is a C program that simply executes a Halt() **system call** to shut down the OS. This one already works. In the run configuration, use the –x flag to execute a user program. Nachos executables are in the "coff" file format so make sure to include it in the file name. The images below show how to do steps 3 and 4 in IntelliJ and VSCode.



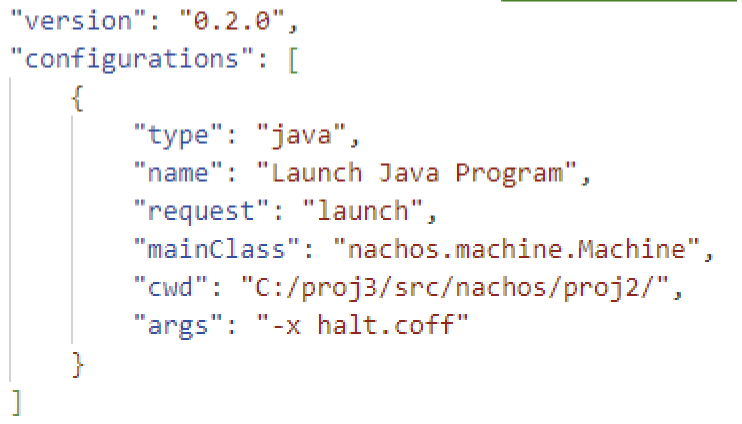
Name: halt

Build and run

```
java 13 SDK of 'proj3' module
-x halt.coff
```

Press Alt for field hints

Working directory: K:\CSC 335 OS\J



```
"version": "0.2.0",
"configurations": [
    {
        "type": "java",
        "name": "Launch Java Program",
        "request": "launch",
        "mainClass": "nachos.machine.Machine",
        "cwd": "C:/proj3/src/nachos/proj2/",
        "args": "-x halt.coff"
    }
]
```

How to set the working directory and the command-line argument –x for executing a user process in IntelliJ (left) and VSCode (right). Note that VSCode is now using "cwd" to indicate where the nachos.conf file is.

5. When you run the program, you'll first see a test of the console where you can type things and have them echoed back to you. Type q to quit the console test and watch as the machine then halts. Then try adding the "a" (address space) and "c" (coff file) debugging flags in the run configuration so it says

   ```
   -d ac –x halt.coff
   ```

   for the program arguments. Running it will now produce some extra information as it loads the process into memory. Use these debugging flags as you go through this project to help you see what's going on.
6. That console test in the previous step (where you have to type q to quit) is going to run with *every* user program, and that's going to get tiresome. Find where that's happening in the **UserKernel** class in the userprog directory and comment out the appropriate lines. Rerun halt.coff until the machine halts without that console test executing.
7. Finally, download and extract the zip file from Nexus that has a bunch of other user programs you can test your code with. Move all of these *.c and *.coff files into your *test* directory in Nachos. None of them work yet.

This project consists of three parts -- all non-trivial. There is a lot to digest about how NACHOS works and a lot of questions that will come up in the process. Start early. Ask lots of questions. You won't be able to proceed to a subsequent part unless you are pretty sure that each of the previous parts is working satisfactorily.

Part 1: Know your Nachos

You've already read the Nachos walkthrough and tutorial. Most of your time will be spent modifying files in the *userprog* directory in Nachos. **UserKernel** is the part of the OS that loads and runs user programs, and each user program is just an instance of **UserProcess**. Study these files. You should also read machine.**Processor** and machine.**CoffSection** to see what Nachos is already providing you. Then go to Gradescope and answer the questions there.

Part 2: Page it!

Implement a pure paging scheme (not demand paging) by creating the page table and then using it to translate addresses appropriately.

Page table creation

1. Add a free frames list to the UserKernel class. This will keep track of all the frames not currently in use by a process. Every UserProcess instance will need to access this list (possibly concurrently) so it needs to be thread-safe. Use your DLList monitor from the previous project to implement it. The free frame list will be initialized to `numPhysPages` nodes, where `numPhysPages` is the total number of frames.

2. Implement the following two UserKernel methods:

```
/**
 * return a list of <requested> free frame numbers that can be used
 * for a process
 *
 * @param requested  number of free frames requested
 * @return array of frame numbers that the process can use or null
 * if request cannot be fulfilled
 */
public static int[] allocatePages(int requested){

/**
 * put frameNumber back in the free frames list
 */
public static void releasePage(int frameNumber)
```

3. Alter the `loadSections` method in UserProcess to
   a. allocate the correct number of pages to the process
   b. create a page table with just that many pages
   c. load each page of each section of the coff file into its correct frame numbers, making a page table entry for each as you go along
   d. Make the appropriate number of page table entries for the stack and the arguments to the function

4. Write the `unloadSections` method in UserProcess. It should close the coff file and destroy the page table, freeing up all frames that it used.

Address Translation

Now that the page table is made, you need to make sure it's used whenever address translation happens, i.e. whenever memory is read from or written to. Thus, your next task is to alter `readVirtualMemory` and `writeVirtualMemory`. They both do the same thing: copying data either from the `memory` array to the `data` parameter array (read) or vice versa (write). Get the page and offset you need from the logical address (hint: you shouldn't need to write this yourself) and then get the appropriate page table entry to know the frame number. Then copy the appropriate number of bytes using System.arraycopy.

PITFALL ALERT: the `length` parameter tells you how much to transfer. This could be less than a page's amount of data, a single page's amount of data, or more than a page's amount of data. That means you need to copy a page at a time, remembering that the first and last pages you copy may not be whole pages. Make sure to ask me if you don't understand this!

Put some debugging print statements in `loadSections` that will print (1) the number of pages allocated to this process and (2) which page maps to which frame. Then run halt.coff again to make sure it still works. Finally, alter the allocation process to make sure page *i* is allocated to something else other than frame *i*. If it does and halt still works, then congrats! You've got a paging system that can produce relocatable code! Take a screenshot of halt working under this last condition to turn in with your code.

Part 3: System calls
Recall that system calls are low-level OS operations that applications can ask to have done on their behalf. (Review slides 18-20.) Nachos system calls are defined in test.syscall.h. On Nexus are two sample methods for making the Read and Write method calls partially work. Add them to the appropriate class and then integrate them into the method calling system. Finally, write the code necessary for the exit() system call to work. It should deallocate the process's page table and terminate the kernel.

Make sure all user programs I gave you on Nexus work with the new calls. Take screenshots of each user program producing correct output. (except the "sort_too_much" program. It ought to fail… in the correct way.)

Turning it in
Place a pdf version of your code along with all your screenshots into your "nachos" project folder and zip it up. Upload the zip file to Nexus.

Grading
This project is worth 50 points <u>but counts double</u> due to its complexity.

- 13 points for the Gradescope section
- 10 points for correct output of all user programs
- 20 points for correctly implementing the paging system
- 7 points for system call implementation

Gentle Reminder
Programming projects are *individual* projects. I encourage you to talk to others about the general nature of the project and ideas about how to pursue it. However, the technical work, the writing, and the inspiration behind these must be substantially your own. You must cite anyone else who contributes in any way to the project by adding appropriate comments to the code. Similarly, if you include information that you have gleaned from other sources, you must cite them as references. Looking at, and/or copying, other people's code is inappropriate, and will be considered an honor code violation.