PROJECT 02
Multiprogramming

1. General rule:

The project is done in groups: each group has a maximum of **3** students, a minimum of **2** students

- The same exercises will all be scored 0 for the entire practice (even though there are scores for other exercises and practice projects).

- Environment: Linux virtual machine (recommend Ubuntu 18.04), Nachos 4.0

2. Submission Requirements:

Submit assignments directly on the course website, not accepting submissions via email or other forms.

Filename: **StudentID1_StudentID2_StudentID3.zip** (with StudentID1 < StudentID2 < StudentID3)

Ex: Your group has 3 students: 2012001, 2012002 and 2012003, the filename is: **2012001_2012002_2012003.zip**

Include:

1. **StudentID1_StudentID2_StudentID3_Report.pdf:** Writeups should be short and sweet. Do not spend too much effort or include your source code on your writeups. The purpose of the report is to give you an opportunity to clarify your solution, any problems with your work, and to add information that may be useful in grading. If you had specific problems or issues, approaches you tried that didn't work, or concepts that were not fully implemented, then an explanation in your report may help us to assign partial credit
2. **StudentID1_StudentID2_StudentID3_Source**: your team's source code (NachOS-4.0/code/)

Submit the program: when you're done, delete the object files (.o) and executables you created

Note: It is necessary to strictly follow the above requirements, otherwise the work will not be graded.

3. Demo Interviews

Your implementation is graded on completeness, correctness, programming style, thoroughness of testing, your solution, and code understanding.

When administering this course, we do our best to give a fair assessment to each individual based on each person's contribution to the project

4. Grading:

Part		**Describe**	**Point**
1	1	Change the code for other exceptions (not system call exceptions) so the process can complete	**0,5**
	2	Multiprogramming solutions: Manage memory allocation and release, data section management, and synchronization.	**2,5**
2	1	syscall Exec	**1**
	2	syscall Join, Exit	**2**
	3	syscall CreateSemaphore	**0,5**
	4	syscall Wait, Signal	**2**
	5	syscall Exec with argument	**0,5**
3		shell program	**1**
		Report (*your project will not be graded without the report*)	

Content

Part1. Multiprogramming

In this lab we will observe and practice how multiple processes share the single CPU and other resources of the computer system. After a program is compiled and linked, its binary executable is usually stored in a file in the secondary storage. How does the process which exe- cutes this program start? How does the system switch from one process to another when it cannot proceed? How can a suspended process resume its execution?

We will design and implement to support multiprogramming on Nachos. You have to write more system calls about process management and interprocess communication. Nachos is currently a uniprogramming environment only (one process at a time). We will program each process to be maintained in its system thread. We have to manage memory allocation and release, data section management, and synchronization.

Note that the solution should be designed before setting up the program. Details are as follows:

1. Change the code for other exceptions (not system call exceptions) so the process can complete, rather than halting the machine like before. A runtime exception will not cause the operating system to shut down. Process the synchronization job when the process is finished
2. Implement multi-process. The current NachOS restricts you to only one program, you have to make some changes in the **addrspace.h** and **addrspace.cc** files to convert the system from uni-program to multi-program. You will need to:
    a. Solves the problem of allocating frames of physical memory, so that multiple programs can load into memory at once. (**bitmap.h**)
    b. Must handle releasing memory when the user program terminates.
    c. The important part is changing the instruction that loads the user programs into memory. Currently, address space allocation assumes that a program is loaded into consecutive segments of memory. Once we support multiprogramming, memory will no longer be contiguous. If we program incorrectly, loading a new program can damage the OS.

    Add synchronization to the routines that create and initialize address spaces so that they can be accessed concurrently by multiple programs. Note that **scheduler.cc** now saves and restores user machine state on context switches

Part2. System calls:

1. Implement the syscall **SpaceID Exec(char* name).** Exec return -1 if was error and Successful returns the Process SpaceID of the newly created user user program. This is the information that needs to be managed in the Ptable class.
2. Implement the syscalls: **int Join(SpaceID id)** and **void Exit(int exitCode)**

- Join will wait and block on a “Process ProcessID ” as noted in its parameter. Join returns the exit code for the process it is blocking on, -1 if the join fails. The exit code parameter is set via the exitCode parameter.
- Exit returns an exit code to whoever is doing a join: 0 if a program successfully completes, another value if there is an error

A user program can only participate in processes that have been created with the Exec system call. You cannot join other processes or your main process. You must use a semaphore to distribute activity between Join and Exit user programs.

3. Implement the syscall **int CreateSemaphore(char* name, int semval).** You have created a data structure to store 10 semaphores. System call CreateSemaphore returns 0 if successful, otherwise returns -1.
4. Implement the syscall **int Wait(char* name)**, and **int Signal(char* name).** Both system calls return 0 on success and -1 on failure. Errors can occur if the user uses the wrong semaphore name or the semaphore has not been created
5. The current version of the “Exec” system call does not provide any way for the user program to pass parameters or arguments to the newly created address space. UNIX does allow this, for instance, to pass in command line arguments to the new address space. Implement the syscall **SpaceId ExecV(int argc, char* argv[]).** Run the executable, stored in the Nachos file "***argv[0]***", with parameters stored in ***argv[1..argc-1]*** and return the address space identifier (Like Exec without argument)

Part3. Test program

Install a simple shell program to test the system calls installed above, and at least two other utility programs such as UNIX cat and cp. The shell receives one command at a time from the user via the console and executes the program accordingly, then runs the command by invoking the kernel system call Exec. The UNIX program ***csh*** is an example of a shell.

The shell should "join" each program and wait until the program terminates. When the Join function returns, show the exit code if it is not 0

This is an example:

```
…
SpaceId newProc1;
SpaceId newProc2;

newProc1 = Exec(“cat”); // Project 01
newProc2 = Exec(“copy”); // Project 01

Join(newProc1);
Join(newProc2);
…
```