CSE-314

<u>Nachos Project 2: Multiprogramming</u>

The second phase of Nachos is to support multiprogramming. You need the code of your first assignment and now your job is to complete enhance the system.

Up to now, all the code you have written for Nachos has been part of the operating system kernel. In a real operating system the kernel not only uses its procedures internally, but also allows user-level programs to access some of its routines via system calls. An executing user program is a process. **In this project you will modify Nachos to support multiple processes, using system calls to have processes request services from the kernel**.

Since your kernel does not trust user programs to execute safely, the kernel and the (simulated) hardware will work together to protect the system from damage by malicious or buggy user programs. To this end, you will implement simple versions of key mechanisms found in real operating system kernels: virtual-addressing, protected system calls and kernel exception handling. Virtual addressing prevents user processes from accessing kernel data structures or the memory of other programs; **your kernel will use process page tables to safely allow multiple processes to reside in memory at the same time**. With protected system calls and exceptions, all entries into the kernel funnel through a single kernel routine, "**ExceptionHandler**"; you will "**bullet-proof**" this routine so that buggy or malicious user programs cannot cause the kernel to crash or behave inappropriately. All of these protection mechanisms require cooperation between the hardware and the operating system kernel software. Your implementation will be based on "hardware" support in the Nachos MIPS simulator, which resembles a real MIPS processor.

If all processes are created by other processes, then who creates the first user process? The operating system kernel creates this process itself as part of its initialization sequence. This is bootstrapping. You can "boot" the Nachos kernel by running nachos with the -x option (x for "execute"), giving the name of an initial program (MIPS format) to run as the initial process. The Nachos release implements the -x option by calling **StartProcess** in **userprog/progtest.cc** to handcraft the initial process and execute the initial program within it. The initial process may then create other processes, which may create other processes...and so on.

The first step is to read and understand the part of the system written for you. The code that you will have to look at and modify is spread over several directories. There are some kernel files in "userprog", a few additional machine simulation files in "machine", and a stub file system in "filesys". The user programs are in "test", and utilities to generate a Nachos loadable executable are in "bin". Since Nachos executes MIPS instructions (and there aren't very many MIPS workstations left!), you will need a cross-compiler. The cross-compiler runs on Linux and compiles user programs into MIPS format.

The code provided can run only a single user-level "C" program at a time. As a test case, there is a trivial user program, "halt". All "halt" does is to turn around and ask the operating system to shut the machine down. To run the "halt" program, make and then run Nachos in the "userprog" directory:

```
% cd userprog
% ./nachos -x ../test/halt
```

Trace what happens as the user program gets loaded, runs, and invokes a system call.

As with the previous assignment, you may find Narten's Road Map to Nachos and Nachos System Call Interface helpful.

In this project, you will be adding your code to addrspace.cc, progtest.cc, and exception.cc in the userprog directory. You will also be creating test programs in the test directory. Overall, the files for this assignment that you might look at include:

- **progtest.cc** -- test routines for running user programs.
- **addrspace.h, addrspace.cc** -- create an address space in which to run a user program, and load the program from disk.
- **syscall.h** -- the system call interface: kernel procedures that user programs can invoke.
- **exception.cc** -- the handler for system calls and other user-level exceptions, such as page faults. In the initial Nachos code, only the "halt" system call is supported.
- **bitmap.h, bitmap.cc** -- routines for manipulating bitmaps (this might be useful for keeping track of physical page frames)
- **filesys.h, openfile.h** (found in the filesys directory) -- a stub defining the Nachos file system routines. For this project, the Nachos file system makes direct corresponding calls to the UNIX file system.
- **translate.h, translate.cc** -- translation table routines. In the initial code, we assume that every virtual address is the same as its physical address — this restricts us to running one user program at a time. You will generalize this to allow multiple user programs to be run concurrently. We will not ask you to implement demand-paged virtual memory support until project 3; for now, every page will be in physical memory.
- **machine.h, machine.cc** -- emulates the part of the machine that executes user programs: main memory, processor registers, etc.
- **mipssim.cc**-- emulates the integer instruction set of a MIPS R2/3000 processor.
- **console.h, console.cc** -- emulates a terminal device using UNIX files. A terminal is (i) byte oriented, (ii) incoming bytes can be read and written at the same time, and (iii) bytes arrive asynchronously (as a result of user keystrokes), without being explicitly requested.

In this assignment we are giving you a simulated CPU that models a real CPU. In fact, the simulated CPU is the same as the real CPU (a MIPS chip), but we cannot just run user programs as regular UNIX processes because we want complete control over how many instructions are executed at a time, how the address spaces work, and how interrupts and exceptions (including system calls) are handled.

Our simulator can run normal programs compiled from C -- see the Makefile in the "test" subdirectory for an example. The compiled programs must be linked with some special flags, and then converted into Nachos format using the program "coff2noff" (which we supply). The only caveat is that floating-point operations are not supported.

TASK-01 (Basic Multiprogramming Support)

Implement system call handling and multiprogramming. To support multiprogramming, you must support some of the system calls defined in syscall.h: Exec, and Exit. Nachos has an assembly-language routine, "syscall", to provide a way of invoking a system call from a C routine (UNIX has something similar -- try "man syscall"). Use the routine "StartProcess" in progtest.cc as a reference for implementing the "exec" system call.

First, you will need basic facilities to load processes into the memory of the simulated machine. **Spend a few minutes studying the AddrSpace class**, and look at how the StartProcess procedure uses the AddrSpace class methods to create a new process, initialize its memory from an executable file, and start the calling thread running user code in the new process context. The current code for AddrSpace and StartProcess works OK, but it assumes that there is only one program/process running at a time (started with StartProcess from main via the nachos -x option), and that all of the machine's memory is allocated to that process. **Your first job is to generalize this code to implement the Exec system call for the general case in which multiple processes are active simultaneously.**

Start by implementing a memory manager class so that your kernel can conveniently keep track of which physical page frames are free and which have been allocated. Note that the memory manager does not return a pointer to a page, it just manages physical page numbers. To implement the memory manager, you can use the BitMap class (bitmap.h) to track allocated and free physical page numbers. It was created with this goal in mind. You are free to implement the memory manager as you like, but here is a suggested interface:

```
/* Create a manager to track the allocation of numPages of physical memory.
   You will create one by calling the constructor with NumPhysPages as
   the parameter.  All physical pages start as free, unallocated pages. */
MemoryManager(int numPages)

/* Allocate a free page, returning its physical page number or -1
   if there are no free pages available. */
int AllocPage()

/* Free the physical page and make it available for future allocation. */
void FreePage(int physPageNum)

/* True if the physical page is allocated, false otherwise. */
bool PageIsAllocated(int physPageNum)
```

One last thing about MemoryManager is that it needs to be thread safe. As with other thread-safe classes, create an internal lock and have each method of MemoryManager acquire/release that lock.

Now modify AddrSpace to allow multiple processes to be resident in the machine memory at the same time. The default AddrSpace constructor code assumes that all of the machine memory is free, and it loads the new process contiguously starting at page frame 0. You must modify this scheme to use your memory manager to allocate page frames for the new process, and load the process code and data into those allocated page frames (which are not typically contiguous). For now it is acceptable to fail and return an error (0) from Exec if there is not enough free total machine memory to load the executable file.

Also modify AddrSpace to call the memory manager to release the pages allocated to a process when the process is destroyed. Make sure your AddrSpace code also releases any frames allocated to the process in the case where it discovers that it does not have enough memory to load the entire program into the address space of the process.

You can use some help from [here].

array.c -- sample program testing loading code and data.

TASK-02 (Implementing System Calls : Exec, Exit)

Now, use these new facilities to implement the Exec and Exit system calls as defined in the Nachos System Call Interface and in userprog/syscall.h. To add to the general confusion of terms, the Naches Exec() system call both creates a new process and loads and runs a new program in that process. (It is similar to CreateProcess on Windows, or combining fork+exec on Unix.) If an executing user process requests a system call (by executing a trap instruction) the machine will transfer control to your kernel by calling ExceptionHandler in exception.cc. Your kernel code must extract the system call identifier and the arguments from the machine registers, decode them, and call internal procedures that implement the system call. You may impose a reasonable limit on the maximum size of a file name. Also, use of Machine::ReadMem and Machine::WriteMem is not forbidden as the comment in machine.h implies.

When an Exec call returns, your kernel should have created a new process and started a new thread executing within it to run the specified program. At this point, do not concern yourself with setting up OpenFileIds. For now, you will be able to run user programs, but they will not be able to read any input or write any output. **For Exec, you must copy the filename argument from user memory into kernel memory safely, so that a malicious or buggy user process cannot crash your kernel or violate security.**

Your kernel must handle the case where the filename string crosses user page boundaries and resides in noncontiguous physical memory. You must also detect an illegal string address or a string that runs off the end of the user's address space without a terminating null character. Your kernel should handle these cases by returning an error (SpaceId 0) from the Exec system call.

Exec must return a unique process identifier (SpaceId) for each process created, and your kernel will need to keep track of active processes. Implement a thread-safe table class to use as the process table. The Table class will store an array of untyped object pointers indexed by integers in the range [0...size-1], and support the following methods:

```
/* Create a table to hold at most "size" entries. */
Table(int size)

/* Allocate a table slot for "object", returning the "index" of the
   allocated entry; otherwise, return -1 if no free slots are available. */
int Alloc(void *object)

/* Retrieve the object from table slot at "index", or NULL if that
   slot has not been allocated. */
void *Get(int index)

/* Free the table slot at index. */
void Release(int index)
```

To make it thread-safe, simply allocate a lock in the constructor and acquire/release the lock in each of the methods. As with MemoryManager, you can declare a global pointer to the process table and initialize it in StartProcess (since StartProcess only runs once, and runs before any processes start).

In your Exit system call hander, print out the status value passed as the parameter. This will aid in debugging and testing. Here are some sample test programs you can try.

- exittest.c -- sample program testing Exit
- exectest.c -- sample program testing Exec and Exit

Anyway, you should write your own test code.

TASK-03 (Implementing System Calls: read, write)

Implement the "read" and "write" system calls as documented in **userprog\syscall.h**

If the arguments to Read and Write are invalid (e.g., invalid buffer address), have the system calls return -1 as an error.

TASK-04 (Bonus: Exception Handling)

Implement the Nachos kernel code to **handle user program exceptions** that are not system calls. The machine (or simulator) raises an exception whenever it is unable to execute the next user instruction, e.g., because of an attempt to reference an illegal address, a privileged or illegal instruction or operand, or an arithmetic underflow or overflow condition. The kernel's role is to handle these exceptions in a reasonable way, i.e., by printing an error message and killing the process rather than crashing the whole system. (**Note: an ASSERT that crashes Nachos is a reasonable response to a bug within your Nachos kernel as in project 1, but you do not want to use ASSERT when a user program causes an exception**.)

This part is relatively straightforward to do by adding code to ExceptionHandler in exception.cc to check for various exception types other than SyscallException. All exception types are defined in the machine/machine.h header file.

Submission

- Create a folder 1305xxx (ex: 1305001). Place the code folder of your extended nachos inside the folder.
- Zip the folder. Your zip file should be named 1305xxx.zip
- Submit the zip file

Marks Distribution

Proper submission	10%
Task 1	25%
Task 2	35%
Task 3	30%
Task 4 (Bonus)	20%

Deadline

12/05/2017 11:55 PM

(Friday of 10th academic week)

Acknowledgement

The content of this assignment was made with the help of Nachos assignment declaration of **Dept. of CSE, UC San Diego** (**http://cseweb.ucsd.edu/classes/fa14/cse120-b/projects/multi.html**)