Project 2: Multiprogramming

Fall 2019

The second Nachos project is to implement system calls and support multiprogramming. As in the first project, we give you some of the code you need and your task is to complete the system and enhance it. 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 allows user-level programs to access some of its routines via *system calls*. The goal of this project is to enable user-level programs to invoke Nachos routines that you implement in the Nachos kernel.

~~Due: Wednesday, November 13 at 11:59pm~~
Due: Saturday, November 16, at 9:00pm

Background

The changes you make to Nachos will be in these two files in the `userprog` directory:

- `UserKernel.java` - a multiprogramming kernel.
- `UserProcess.java` - a user process; manages the address space, and loads a program into virtual memory.

You will also want to familiarize yourself with the other classes in `userprog`:

- `UThread.java` - a thread capable of executing user MIPS code.
- `SynchConsole.java` - a synchronized console; makes it possible to share the machine's serial console among multiple threads.

as well as a couple of classes in the `machine` directory:

- `Processor` simulates a MIPS processor.
- `FileSystem` is a file system interface. To access files, use the `FileSystem` returned by `Machine.stubFileSystem()`. This file system accesses files in the `test` directory.

Nachos emulates user programs executing on a real CPU (a MIPS R3000 chip). By emulating execution, Nachos has complete control over how many instructions are executed at a time, how the address translation works, and how interrupts and exceptions (including system calls) are handled. The emulator can run normal programs compiled from C to the MIPS instruction set. The only caveat is that floating point operations are not supported.

Nachos initially is only able to run a single user-level MIPS program at a time, and supports one system call fully: `halt`. All `halt` does is ask the operating system to shut the machine down. This test program is found in `test/halt.c` and represents the simplest supported MIPS program.

Nachos provides several other example MIPS programs in the `test` directory. You can use these programs to test your implementation, and you will be writing new test programs of your own. Of course, you will not be able to run the programs which make use of features such as I/O until you implement the appropriate kernel support. That will be your task in this project.

To compile the test programs, you need a MIPS cross-compiler. This cross-compiler is already installed on the instructional machines as `mips-gcc` (see the `Makefile` in the `test` directory for details). When logged into an instructional machine, `prep` will initialize two environment variables (ARCHDIR and PATH) to enable you to use the cross-compiler. If you would like to use the cross-compiler on your own Linux machine (or Linux virtual machine), you can download the distribution to your system. Alternatively, you can compile your test programs on `ieng6` and copy them over to your own machine.

The `test` directory includes C source files (`.c` files) and Nachos user program binaries (`.coff` files). The binaries can be created while in the `test` directory by running `make`, or from the `proj2` directory by running `make test`.

You can run test programs by running `nachos -x program.coff` where `program.coff` is the name of the MIPS program binary in the `test` directory. You will be creating many test programs both for this project and project 3, and you will place them all in the `test` directory. To compile your own test programs, add them to the `Makefile` in the `test` directory in the line:

```
TARGETS = halt sh matmalt sort echo cat cp mv rm
```

and then run `make`.

Tasks

0. (0%) Run your first Nachos user-level program. Make sure you have run prep if necessary:

   ```
   % prep cs120fa19
   ```

   Go to the `proj2` directory, run `make test` to compile the test programs, run `make` to compile Nachos, and then run `nachos -x halt.coff`.

   ```
   % cd nachos/proj2
   % make test
   % make
   % nachos -x halt.coff
   ```

   After typing 'q' at the prompt, Nachos will shutdown ("Machine halting!"). To give you more insight into what happens as the user program gets loaded, runs, and invokes a system call, next run `nachos -d ma -x halt.coff`. The `'m'` debug flag enables MIPS disassembly, showing the instructions of the user-level program being emulated on the CPU, and the `'a'` debug flag prints process loading information.

```
% nachos -d ma -x halt.coff
```

Note that the baseline Nachos implementation has code in UserKernel.selfTest() that tests the console device. Nachos executes this code every time it runs. When you start testing your implementation, you can comment out this code so that you do not have to type 'q' every time you run Nachos.

1. (35%) Implement the file system calls `creat`, `open`, `read`, `write`, `close`, and `unlink`. Their semantics and specifications are documented in `test/syscall.h`, and the calling conventions are documented in the comments to `UserProcess.handleSyscall`. You will see the code for `halt` and skeleton code for `exit` in `UserProcess.java`. Implement the other system calls following the same pattern. Note that you are *not* implementing a file system. Rather, you are simply giving user processes the ability to access a file system that Nachos already implements.

   For further suggestions and tips, see the Tips section below. For examples and strategies for testing, see the Testing section below.

   - Nachos already provides the assembly code necessary for user-level programs to invoke system calls (see `test/start.s`; the `SYSCALLSTUB` macro generates assembly code for each syscall).
   - When implementing the system calls, you will need to "bullet-proof" the Nachos kernel from user program errors. There should be nothing a user program can do to crash the operating system (with the exception of explicitly invoking the `halt` syscall). In other words, you must be sure that user programs do not pass bad arguments to the kernel (e.g., a NULL pointer value of 0x0, or an invalid address) that cause the kernel to crash or corrupt its internal state or that of other processes.
   - To handle large `read` / `write` calls, you should use a page-sized buffer to pass data between the file and user memory.
   - Since the memory addresses passed as arguments to the system calls are virtual addresses, you need to use `UserProcess.readVirtualMemory` and `UserProcess.writeVirtualMemory` to transfer data between the user process and the kernel.
   - User processes store filenames and other string arguments as null-terminated strings in their virtual address space. The maximum length for strings passed as arguments to system calls is 256 bytes (not including the terminating null).
   - System calls should return the appropriate value as documented in `test/syscall.h`. When a system call needs to indicate an error condition to the user, it should return -1. In particular, it should not assert or otherwise thrown an exception.
   - When any process is started, its file descriptors 0 and 1 must refer to standard input and standard output. Use `UserKernel.console.openForReading()` and `UserKernel.console.openForWriting()` to implement these

semantics. A user process *is* allowed to close these descriptors, just like descriptors returned by `open`.

- A stub file system interface to the UNIX file system is already provided for you, and the interface is implemented by the class `machine/FileSystem.java`. You can access the stub filesystem through the static field `ThreadedKernel.fileSystem`. (Note that since `UserKernel` extends `ThreadedKernel`, you can still access this field.) This filesystem is capable of accessing the `test` directory in Nachos, which is going to be useful when you implement the `exec` system call described below. You do not need to implement any file system functionality, but you should examine carefully the specifications for `FileSystem` and `StubFileSystem` to determine what functionality you need to implement, and what is handled by the file system.

- Do not implement any kind of file locking, the file system is responsible for it. If `ThreadedKernel.fileSystem.open()` returns a non-null `OpenFile`, then the user process is allowed to access the given file; otherwise, you should return an error. Likewise, you do not need to worry about the details of what happens if multiple processes attempt to access the same file at once; the stub filesystem handles these details for you.

- Each file that a process opens should have a unique *file descriptor* associated with it (see `syscall.h` for details). The file descriptor should be a non-negative integer that is simply used to index into a table of currently-open files by that process. Your implementation should have a file table size of 16, supporting up to 16 concurrently open files per process. Note that a given file descriptor can be reused if the file associated with it is closed, and that different processes can use the same file descriptor value to refer to different files (e.g., in every process file descriptor 0 refers to stdin).

2. (30%) Implement support for multiprogramming. The initial Nachos code is restricted to running only one user process, and your task is to make it work for multiple user processes. For further suggestions, see the Tips section below. For examples and strategies for testing, see the Testing section below. To help understand how Nachos translates from virtual to physical addresses, we strongly recommend doing the VM Worksheet in Homework #3 at this point.

   - You will need to manage the allocation of pages of physical memory so that different processes do not overlap in their memory usage. You can use whatever data structure you like to manage physical pages, but we suggest maintaining a static linked list of free physical pages (perhaps as part of the `UserKernel` class). Be sure to use synchronization where necessary when accessing this list to prevent race conditions.

     Your solution must make *efficient* use of memory by allocating pages for a new process wherever possible. This means that it is not acceptable to only allocate pages in a contiguous block; your implementation must be able to make use of "gaps" in the free memory pool.

   - You will create and initialize the `pageTable` data structure for each user process, which maps the process's virtual addresses to physical addresses. The `TranslationEntry` class represents a single virtual-to-physical page

translation. The field `TranslationEntry.readOnly` should be set to `true` if the page is coming from a COFF section which is marked as read-only. You can determine this status using the method `CoffSection.isReadOnly()`.

Modify `UserProcess.loadSections()` so that it allocates the `pageTable` and the number of physical pages based on the size of the address space required to load and run the user program (and no larger). This method is the one that should set up the `pageTable` structure for the process so that the program is loaded into the physical memory pages it has allocated for the address space. Note that user programs do not make use of `malloc` or `free`, meaning that user programs effectively have no dynamic memory allocation (and therefore, no heap). The stack is fixed size as well. As a result, Nachos knows how many virtual pages a new process needs for its address space when it is created. If the new user process cannot allocate sufficient physical pages for its address space, `exec` should return an error.

All of a process's memory should be freed on exit (whether it exits normally, via the syscall `exit`, or abnormally, due to an illegal operation). As a result, its physical pages can be subsequently reused by future processes.

- Modify `UserProcess.readVirtualMemory` and `UserProcess.writeVirtualMemory`, which copy data between the kernel and the user's virtual address space, so that they work with multiple user processes. Note that these methods should not throw exceptions if they encounter an error when copying data; instead, they must always return the number of bytes transferred (even if that number is zero).

  The physical memory of the MIPS machine is accessed through the method `Machine.processor().getMemory()`, and the total number of physical pages is `Machine.processor().getNumPhysPages()`.

- The user threads (see the `UThread` class) already save and restore user machine state, as well as process state, on context switches. So you are not responsible for these details.

3. (35%) Implement the system calls `exec`, `join`, and `exit`, also documented in `syscall.h`. For further suggestions, see the Tips section below. For examples and strategies for testing, see the Testing section below.

   - Note that, although Nachos chose the name `exec` for its system call, it is not the same as the Unix exec system call. As described in `syscall.h`, the Nachos `exec` system call both creates a new process and loads a new program into that process. (As a result, it essentially combines `fork/exec` on Unix and is similar to `CreateProcess` on Windows.)

   - As with the other system calls, the addresses passed in registers as arguments to `exec` and `join` are virtual addresses. Use the methods `readVirtualMemory` and `readVirtualMemoryString` to transfer data between kernel memory and the memory of the user process.

- Also bullet-proof these syscalls (e.g., handle cases such as the program passing a NULL pointer value of 0x0, or an invalid address, as the file name to `exec`).

- Note that the memory of the child process is entirely private to this process. This means that the parent and child do not directly share memory or file descriptors. Note that two processes can of course open the same file; for example, all processes should have file descriptors 0 and 1 mapped to the system console, as described above.

- Use `KThread.join` to implement the `join` system call (you have implemented the functionality once, no need to implement it again). Unlike threads using `KThread.join` in project 1, though, for this project enforce the rule that only a process's parent can join to it. For instance, if A executes B and B executes C, A is not allowed to join to C, but B *is* allowed to join to C.

- `join` takes a *process ID* as an argument, which is used to uniquely identify the child process which the parent wishes to join with. The process ID should be a *globally unique positive integer*, assigned to each process when it is created, and set the process ID of the first process to 0. (Although for this project the only use of the process ID is in `join`, for project 3 it is important that the process ID is unique across all running processes in the system.) The easiest way of accomplishing this is to maintain a static counter which indicates the next process ID to assign. Since the process ID is an `int`, then it may be possible for this value to overflow if there are many processes in the system. For this project you are not expected to deal with this case; that is, assume that the process ID counter will not overflow.

- Extend the implementation of the `halt` system call so that it can only be invoked by the "root" process — that is, the initial process in the system. If another process attempts to invoke `halt`, the system should not halt and the handler should return immediately with -1 to indicate an error.

- When a process calls `exit`, its thread should be terminated and the process should clean up any state associated with it (i.e., free up memory, close open files, etc.). Perform the same cleanup if a process exits abnormally (e.g., executes an illegal instruction).

- If a parent process has called `join` on a child process, and the child process exits normally, then `join` needs to transfer the child's exit status value to the parent (see methods in the `Lib` class for converting between bytes and integers). A child process exits normally when it calls the `exit` system call and provides a status value as an argument. If the `status` parameter is NULL, then `join` behaves normally and simply does not return the status from the child. If the `status` parameter is invalid (e.g., beyond the end of the address space), then `join` immediately returns with -1 to indicate an error.

  If a child process terminates abnormally (e.g., due to an unhandled exception), it will not have an exit status. In this case, `join` will return 0 to the parent and the value of the `status` parameter does not need to be set (see `test/syscall.h` for the complete specification).

- The last process to call `exit` should cause the machine to halt by calling `Kernel.kernel.terminate()`. (Note that only the root process should be allowed to invoke the `halt` system call, but the last exiting process should call `Kernel.kernel.terminate()` directly.)

EC1. (5%) The extra credit problem provides an additional challenge for students interested in implementing more functionality in Nachos. We recommend only working on the extra credit once you are completely finished with the other parts of the project.

Implement named pipes as an interprocess communication mechanism between processes. A pipe has an internal kernel buffer one page in size that follows the producer-consumer pattern (note that, although it has a name, it is not a file). A process can write into a pipe until the pipe fills up, at which point it blocks until the pipe is no longer full. Write will not return until all bytes from the write buffer have been written into the pipe. A process can read from a pipe as long as the pipe is not empty, at which point it blocks until the buffer is no longer empty. Specifically, if the pipe is not empty when read is called, read will return the bytes that are available in the pipe (even if the read buffer is much larger), up to the size of the read buffer. If the pipe is empty when read is called, read will block until the writer writes more bytes in the pipe (it will not return 0 in this case).

You need only support one writer and one reader process, where one process creates the pipe and writes to it while another process opens it and reads from it. Use synchronization primitives (not interrupts) to synchronize the writer and reader. A process creates a pipe using the `creat` system call with a special pipe name argument of the form "/pipe/*name*", where *name* is a valid filename string. If a pipe of the same name already exists, `creat` returns -1. Otherwise, `creat` creates a new pipe and returns a new file descriptor for it. A system-wide maximum of 16 named pipes can be created at once, and `creat` returns -1 when 16 pipes are in use.

A process opens a pipe using the `open` system call with a pipe name as the name argument. If the named pipe has not been created, `open` returns -1. Otherwise, `open` returns a new file descriptor for the named pipe. Creating and opening pipes count towards the per-process file descriptor limit. To write or read from a pipe, a process simply calls the `write` and `read` system calls on the pipe file descriptor. A process closes a pipe using the `close` system call on the pipe file descriptor. After being created, a pipe exists as long as at least one process has a file descriptor referring to it.

Tips

Here are some guidelines and tips for project 2 from previous CSE 120 TAs:

- Ryan Huang's tips
- Matus Telgarsky's tips

Testing

As with all of the projects, it is your responsibility to implement your own tests to thoroughly exercise your code to ensure that it meets the requirements specified for each part of the project. Testing is an important skill to develop, and the Nachos projects will help you to continue to develop that skill. In this project, you will implement tests as user-level programs written in C. See the discussion at the top of this page on creating test programs in the `test` directory, compiling them, and running them with Nachos.

The following pages provide testing strategies and example test programs for the project:

- File system calls
- Multiprogramming support
- Multiple processes

As with project 1, during the project period you can also use Gradescope to run a snapshot of your code on the sample tests that we have given. For reference, see the following Piazza post describing how to use Gradescope to compile and run the sample tests. **Important:** Before the deadline you must submit your code to Gradescope at least once to initialize the grading system for your project.

Code Submission

As a final step, create a file named README in the `proj2` directory. The README file should list the members of your group and provide a short description of what code you wrote, how well it worked, how you tested your code, and how each group member contributed to the project. The goal is to make it easier for us to understand what you did as we grade your project in case there is a problem with your code, not to burden you with a lot more work. Do not agonize over wording. It does not have to be poetic, but it should be informative.

For grading, as with project 1 we will use a snapshot of your Nachos implementation in your github repository as it exists at the deadline, and grade that version. (Even if you have made changes to your repo after the deadline, that's ok, we will use a snapshot of your code at the deadline.) **Important:** Before the deadline, you must submit your code to Gradescope at least once. See the instructions in the Testing section above.

Cheating

You can discuss concepts with students in other groups, but do not cheat when implementing your project. Cheating includes copying code from someone else's implementation, or copying code from an implementation found on the Internet. See the main project page for more information.

We will manually check and also run code plagiarism tools on submissions and multiple Internet distributions (if you can find it, so can we).

---

voelker@cs.ucsd.edu