CSE 314 – January 2020

Assignment 3 on NACHOS

Introduction

NACHOS stands for Not Another Completely Heuristic Operating System. This was developed in 1992 originally in C++. Later, a Java version was written. Basic Nachos has an incomplete threading system. It also does not support multiple programming (multiple user processes in the same physical memory space).

In this assignment, we are going to complete the nachos threading system and make nachos compatible for multiprogramming.

Grouping

Before explaining the assignment tasks, let us first say that nachos assignments will be in group of three (3) students. You should form your group within your lab group. Cross-groups (one from A1 and others from B1) are NOT allowed. Enter your group info here. When submitting, only one submission is expected.

Assignment Tasks

This assignment is organized in two parts.

- Part 1: Threading
- Part 2: Multiprogramming

Part 1: Threading Tasks

Your working directory for this part will be the `proj1` directory. The files where you will code are in the package `nachos.threads`. Nachos does not *fully* support multiple kernel threads. In this part of the assignment, you will enable Nachos to support multiple kernel threads.

- **Task 1: Implement `KThread.join()`**. Note that another thread does not have to call `join()`, but if it is called, it must be called only once. The result of calling `join()` a second time on the same thread is undefined, even if the second caller is a different thread than the first caller. A thread must finish executing normally whether it is joined.

- **Task 2: Implement condition variables directly**, by using interrupt enable and disable to provide atomicity. We have provided a sample implementation that uses semaphores; your job is to provide an equivalent implementation without directly using semaphores (you may of course still use locks, even though they indirectly use semaphores). Once you are done, you will have two alternative implementations that provide the exact same functionality. Your second implementation of condition variables must reside in class `nachos.threads.Condition2`.

- **Task 3: Complete the implementation of the Alarm class**, by implementing the `waitUntil(long x)` method. A thread calls `waitUntil` to suspend its own execution until time has advanced to at least `now + x`. This is useful for threads that operate in real-time, for example, for blinking the cursor once per second. There is no requirement that threads start running immediately after waking up; just put them on the ready queue in the timer interrupt handler after they have waited for at least the right amount of time. Do not fork any additional threads to implement `waitUntil()`; you need only modify `waitUntil()` and the timer interrupt handler. `waitUntil` is not limited to one thread; any number of threads may call it and be suspended at any one time.

- **Task 4: Implement synchronous `send` and `receive` of one-word messages** using condition variables (don't use semaphores!). Implement the `Communicator` class with operations, `void speak (int word)` and `int listen()`.

  `speak()` atomically waits until `listen()` is called on the same `Communicator` object, and then transfers the word over to `listen()`. Once the transfer is made, both can return. Similarly, `listen()` waits until `speak()` is called, at which point the transfer is made, and both can return (`listen()` returns the word). Your solution should work even if there are multiple speakers and listeners for the same `Communicator` (note: this is equivalent to a zero-length bounded buffer; since the buffer has no room, the producer and consumer must interact directly, requiring that they wait for one another). Each communicator should only use exactly one lock. If you're using more than one lock, you're making things too complicated.

- **Task 5: Write test code for all the previous four tasks.** *Without the test code, you will not get any marks for that task. The purpose of the test code is to use the methods that you have implemented and demonstrate that they work.*

Part 2: Multiprogramming user processes

Your working directory for this part will be the `proj2` directory. That is, you will go to this directory and type `gmake clean`, `gmake` and `nachos` here. The files where you add your code for this part will be in `nachos.userprog` package.

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 them via *system calls*.

The first step is to read and understand the part of the system we have written for you. The kernel files are in the `nachos.userprog` package, and there are a few additional machine simulation classes that get used:

- `Processor` simulates a MIPS processor.
- `SerialConsole` simulates a serial console (for keyboard input and text output).
- `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.

The new kernel files for Part 2 include:

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

The provided code can run a single user-level MIPS program at a time, and supports just one system call: `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.

We have provided several other example MIPS programs in the test directory of the Nachos distribution. You can use these programs to test your implementation, or you can write new programs. Of course, you won't 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 run user .c programs on nachos, please see the other instruction file on Moodle.

Part 2 Tasks

Nachos can run only one user program as a Nachos process. It can not run more than one user program. The goal of Part 2 is to enhance Nachos so that it can support multiple user processes.

1. **Task 1: Implement the system calls `read` and `write` documented in `syscall.h`.** You will see the code for halt in `UserProcess.java`; it is best for you to place your new system calls here too.

    - We have provided you the assembly code necessary to invoke system calls from user programs (see `start.s`; the `SYSCALLSTUB` macro generates assembly code for each syscall).
    - 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 bogus arguments to the kernel which causes the kernel to corrupt its internal state or that of other processes.
    - You should make it so that the `halt()` system call can only be invoked by the "root" process – that is, the first process in the system. If another process attempts to invoke halt(), the system call should be ignored and return immediately.
    - When a system call wishes to indicate an error condition to the user, it should return -1 (not throw an exception within the kernel!). Otherwise, the system call should return the appropriate value as documented in `test/syscall.h`.
    - 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 make this easier. A user process is allowed to close these descriptors, just like descriptors returned by `open()`.

2. **Task 2: Implement support for multiprogramming.** The code we have given you is restricted to running one user process at a time; your job is to make it work for multiple user processes.

   - Come up with a way of allocating the machine's physical memory so that different processes do not overlap in their memory usage. Note that the user programs do not make use of `malloc()` or `free()`, meaning that user programs effectively have no dynamic memory allocation needs (and therefore, no heap). What this means is that you know the complete memory needs of a process when it is created. You can allocate a fixed number of pages for the processe's stack; 8 pages should be sufficient.

     We suggest maintaining a global linked list of free physical pages (perhaps as part of the `UserKernel` class). Be sure to use synchronization where necessary when accessing this list. Your solution must make efficient use of memory by allocating pages for the new process wherever possible. This means that it is not acceptable to only allocate pages in a contiguous block; your solution must be able to make use of "gaps" in the free memory pool.

     Also be sure that all of a process's memory is freed on exit (whether it exits normally, via the syscall `exit()`, or abnormally, due to an illegal operation).

   - 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. The physical memory of the MIPS machine is accessed through the method `Machine.processor().getMemory()`; the total number of physical pages is `Machine.processor().getNumPhysPages()`. You should maintain the `pageTable` for each user process, which maps the user'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 using the method `CoffSection.isReadOnly()`. Note that these methods should not throw exceptions when they fail; instead, they must always return the number of bytes transferred (even if that number is zero).
   - Modify `UserProcess.loadSections()` so that it allocates the number of pages that it needs (that is, based on the size of the user program), using the allocation policy that you decided upon above. This method should also set up the pageTable structure for the process so that the process is loaded into the correct physical memory pages. If the new user process cannot fit into physical memory, `exec()` should return an error.

     Note that 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. **Task 3: Implement the system calls (`exec`, `join`, and `exit`, also documented in `syscall.h`)**.

   - Again, all the addresses passed in registers to exec and join are virtual addresses. You should use `readVirtualMemory` and `readVirtualMemoryString` to transfer memory between the kernel and the user process.
   - Again, you must bullet-proof these syscalls.
   - Unlike `KThread.join()`, 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, 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. 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.
   - When a process calls `exit()`, its thread should be terminated immediately, 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*.
   - The exit status of the exiting process should be transferred to the parent, in case the parent calls the join system call. The `exit status` of a process that exits abnormally is up to you. For the purposes of join, a child process exits normally if it calls the exit syscall with any status, and abnormally if the kernel kills it (e.g. due to an unhandled exception).
   - 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.)

4. **Write a test c program which makes use of all the three features that you have implemented**, i.e. reading from console, writing to console, making valid and invalid system calls, forking other processes using the syscall `exec`, waiting for them using `join`, letting multiple processes work in parallel by forking multiple processes, trying to halt as non-root process, trying to halt as root process etc. Be innovative. Compile a `coff` file using a cross-compiler, then start nachos using this `coff` executable as the default shell program (see `nachos.conf` file).

Submission

In your submission, you will submit two things:

1. The Java project of Nachos
2. A report that explains step by step what you have done for both parts of the assignment. The report should be a PDF file. The report should have two chapters, one on Part 1 and the other on Part 2. Each chapter should have one section for each of the tasks. In that section, you should explain where in Nachos source code you made change, the data structures that you used, basic idea of implementation, and how you tested your implementation of that task. There should be no separate section for Task 5 of Part 1 and Task 4 of Part 2.

Put the report and the Java project in a single folder. Name that folder: 1605abc_1605prq_1605xyz. Zip that folder. Then, upload only that file. The file should be named 1605abc_1605prq_1605xyz.zip.

Evaluation

You will be evaluated in a group viva. **All members must be present**. During the viva, you will have to present and explain your report and show the code. **The questions will NOT be limited to the assignment tasks, but other things from the Nachos kernel code will be asked as well.**

Book

The PDF file named “nachos-java-walkthrough” contains some useful Nachos overview. You can study the entire thing to get a good idea about how Nachos works, which is recommended. Sections 3.1, 3.3, 3.4 and 3.5 will be very useful in Part 1 of the assignment. For Part 2, entire Section 5 will be very handy.