Project 4: System Calls and Memory Management

Readings: Nachos Tutorial (Chapter 5, Chapter 6)
TA: Qiang Xu (xuq22)
Due date: March 31st, 2015

March 9, 2015

In this assignment we are giving you a simulated CPU that models a real CPU (a MIPS R3000 chip). By simulating the execution, we have 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. Our simulator can run normal programs compiled from C to the MIPS instruction set. The only caveat is that floating point operations are not supported.

The code we provide can run a single user-level MIPS program at a time, and supports just one system call: `halt`. All `halt` does is to 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.

1 Getting ready

Warm-up questions for Task I

1. `UserProcess` needs to keep tracks of the open files associated with the user process. Upon termination of the process, the open files should be closed. What kind of data structures are needed to track the open files and file descriptors?

2. Explain how `readVirtualMemoryString` works. Provide the source code for `UserProcess.readVirtualMemoryInt` method that takes one input argument `int vaddr` as the starting virtual address and returns a 4-byte Integer read. You may use this code in later part of the assignment.

3. Write a C program using the system calls in `syscall.h` in the test directory that does the following: 1) it creates a file named `1`; 2) it writes a string "Hello word"; 3) it creates another file named `2`; 4) it read the bytes from file `1` and write them to file `2`; 5) it deletes file `1`; 6) it closes file `2`. Make sure that you include the header files

```
#include "syscall.h"
#include "stdio.h"
#include "stdlib.h"
```

Compile and submit both the source code and the .coff program.

Warm-up questions for Task II

1. Hand trace the code in the `UserProcess.load` method and explain how the value of `numPages` is determined.

2 Tasks

(1) (30%) Implement the file system calls (`creat`, `open`, `read`, `write`, `close`, and `unlink`, 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. Note that you are not implementing a file system; rather, you are simply giving user processes the ability to access a file system that we have implemented for you.

- 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. Also, you must take steps to ensure that if a user process does anything illegal – such as attempting to access unmapped memory or jumping to a bad address – that the process will be killed cleanly and its resources freed.
- 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 memory 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 of for strings passed as arguments to system calls is 256 bytes.

- 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()`.
- A stub file system interface to the UNIX file system is already provided for you; the interface is given 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 your Nachos distribution, which is going to be useful when you want to support the exec system call (see below). You do not need to implement any file system functionality. You should examine carefully the specifications for `FileSystem` and `StubFileSystem` in order to determine what functionality you should provide in your syscalls, and what is handled by the file system.
- Do not implement any kind of file locking; this is the file system's responsibility. If `ThreadedKernel.fileSystem.open()` returns a non-null `OpenFile`, then the user process is allowed to access the given file; otherwise, you should signal 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.
- Your implementation should support at least **16 concurrently open files** per process. Each file that a process has opened 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. 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 (i.e. integer) to refer to different files.

Tips on debugging and testing You can use the binary executables (using "-x filename.coff") in the `test` directory to test your code. In particular, you can use the .coff file you create in Warm-up question #3 for Task I to test creating a file, writing to/reading from a file, removing a file and closing a file. However, since we only implement a subset of system calls, not all programs are supported (e.g., using sh.coff to load another coff binary from the shell).

(2) (40%) 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.

Step 1: 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 process's stack. By default, 8 pages are used for the stack.

Step 2: We suggest maintaining a global linked list of free physical pages (perhaps as part of the `UserKernel` class). Be sure to use **synchronization** (e.g., use `Lock`) 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.

Step 3: 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. In other word, each entry in the pageTable should be an instance of `TranslationEntry`. 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()`. The page table should be allocated once the total number of pages that a process occupies has been decided.

Step 4: 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.

Step 5: Modify `UserProcess.readVirtualMemory((int vaddr, byte[] data, int offset, int length)` and `UserProcess.writeVirtualMemory(int vaddr, byte[] data, int offset, int length)`, which copy data between the kernel and the user's virtual address space, so that they work with multiple user processes. Note that methods should not throw exceptions when they fail; instead, they must always return the number of bytes transferred (even if that number is zero).

Step 6: Modify `unloadSections` to free all of a process's memory is freed. This method will be called in handling the system call `exit()`.

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

Tips on debugging and testing Include `Lib.debug(dbgProcess, ...)` statements to help debugging your code. Turn on the debug option "-d a". You can test your implementation using `matmult.coff`. Modify `matmult.c` by changing the value in the macro `#define Dim 20`. Observe the changes in the output during Nachos execution (particularly those from `loadSections()`).

(3) (30%) Implement the system calls (`exec`, `join`, and `exit`, also documented in syscall.h).

Step 1. Handle `exec` system call. As indicated in `syscall.h`, it executes the program stored in the specified file, with the specified arguments, in a new child process. It takes three parameters (`exec(char *name, int argc, char **argv)`. The first argument `name` is the starting address in the virtual memory that stores the string containing the name of the .coff executable. The second argument `argc` points to the number of arguments passed (from the parent process to the current child process). The third argument is the starting array of an array of pointers `argv[]` to null-terminated strings that represent the arguments to pass to the child process, where `argv[0]` points to the first argument, and `argv[argc-1]` points to the last argument. Each pointer in the array is stored as a *4-byte integer*. Note to retrieve the arguments, two levels of indirection are needed, namely, to get the value of the ith pointer `argv[i]` first and then to get the *string argument* that `argv[i]` is pointing to.

`exec` returns the child process ID upon success. 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.

Step 2. Handle `join` system call. `join` takes a process ID as an argument, used to uniquely identify the child process which the parent wishes to join with. `join` suspends execution of the current process until the child process specified by the process ID argument has exited. If the child has already exited by the time of the call, returns immediately. When the current process resumes, it disowns the child process, so that join() cannot be used on that process again.

You should avoid using busy wait on the parent process, instead, use condition variable to wakeup the suspended parent process when the chlid process finishes. This part should be implemented in conjunction with the `exit` system call.

Step 3. Handle `exit` system call. When a process terminates, it needs to do a number of things. First, it should close all open files and free up all the file descriptors it uses. Second, it should free all the physical memory. Third, it should notify its parent if the later was suspended in `join()`. Finally, 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.)

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).

Tips on debugging and testing You should now be able to test your implementation using `sh.coff`. `sh.coff` starts a shell that can interpret and run commands. If the process is executed and exits properly, you should be able to run one command after another. If at some point, your program reports the lack of physical memory, you need to check if all the physical memory has been freed by the respective process upon exiting.

Congratulations! You now have a functioning kernel that supports multi-programming and system calls. Paging is used to manage physical memory. The main limitation of this kernel is that it does not support virtual memory, which will be the topic of the bonus project.

3 Submission

You should **commit** your codes through the SVN repository. *Make sure the directory structure is intact.* Under the current nachos project directory, include a file called "report.pdf" (e.g, nachos/proj3/report.pdf for this project). In the report, you should include the following two parts:

1. Answers to the questions in the project description.
2. Key data structures used and a block diagram of your implementation (limited to 2 pages).

Grade of each project is divided as 20% for Q&A, 40% for key data structures/block diagram on the implementation and 40% for the correctness of your implemention with the respective percentage for each task.