Lab Multiprogramming in NachOS

28th of September, 2021

Understanding threads

The ability to run multiple programs simultaneously is called "multiprogramming". In the previous lab, you worked with test cases in which you created multiple threads by hand as part of the operating system. In this assignment, you will work on code that allows `NachOS` to load and run programs that have been compiled separately from the operating system.

So far, 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".

In this assignment we are giving you a simulated CPU that models a real CPU. 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). Instead we use a simulator.

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, then converted into `NachOS` format, using the program "coff2noff" (which we supply). The only caveat is that floating point operations are not supported.

As in the first assignment, we give you some of the code you need; your job is to complete the system and enhance it.

The first step is to read and understand the part of the system we have written for you. Our code can run a single user-level 'C' program at a time. As a test case, we've provided you with a trivial user program, `halt`; all halt does is to turn around and ask the operating system to shut the machine down. This program is located in the `test/` folder. The code for the program is in `halt.c`. To compile it, `cd` to that folder and type `make`. If it fails to compile, you may need to first run the makefile in the `bin/` folder.

Run the program `nachos -x ../test/halt`. As before, trace what happens as the user program gets loaded, runs, and invokes a system call.

It is OK to change the definition of the "machine" parameters. For example, the amount of physical memory – if that helps you design better test cases.

Most of your work on this lab will be done in the `userprog/` folder. The files for this assignment are:

1. `progtest.cc` – test routines for running user programs.
2. `addrspace.h`, `addrspace.cc` – create an address space in which to run a user program, and load the program from the disk.
3. `syscall.h` – the system call interface: kernel procedures that user programs can invoke.
4. `exception.cc` – the handler for system calls and other user-level exceptions, such as page faults. In the code we supply, only the 'halt' system call is supported.
5. `bitmap.h`, `bitmap.cc` – routines for manipulating bitmaps (this might be useful for keeping track of physical page frames)
6. `filesys/filesys.h`, `filesys/openfile.h` – a stub defining the `NachOS` file system routines. *For this assignment, we have implemented the NachOS file system by directly making the corresponding calls to the UNIX file system: this is so that you need to debug only one thing at a time.*

7. `machine/translate.h`, `machine/translate.cc` – translation table routines. In the code we supply, we currently 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 real virtual memory support (with swapping) until the next lab. For now, every page must be in physical memory.
8. `machine/machine.h`, `machine/machine.cc` – emulates the part of the machine that executes user programs: main memory, processor registers, etc.
9. `machine/mipssim.cc` – emulates the integer instruction set of a MIPS R2/3000 processor.
10. `machine/console.h`, `machine/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, (iii) bytes arrive asynchronously (as a result of user keystrokes), without being explicitly requested.
11. `bin/noff.h` – defines the `NachOS` executable file format.

You will also need to modify some of the files in the `test/` folder.

1 System Calls

You must support ALL of the system calls defined in `syscall.h` except for thread **Fork** and **Yield** (which will be implemented as an individual task, below).

To do this, you will need to expand the `if` statement in `userprog/exceptions.cc` which currently only handles the **Halt** system call. You will need to add an `else if` clause for each of the system calls listed in `userprog/syscall.h`. You need to write helper functions in a separate file named `syslib.cc` and call them from your `if` statement (this will require adding the new file to `Makefile.common`). I recommend prepending "sys" or "lib" (ie `sysCreate`) to your helper functions to logicially seperate MIPS system calls from the UNIX helper functions.

Do not make any changes to `syscall.h` since it's used both by `NachOS` kernel code and by the test programs that will run in the operating system.

If you read the comments in these files carefully, you will learn two useful things:

1. Arguments to the system calls are passed in registers 4 through 7. Any return value should be written to register 2. The `Machine` class has `ReadRegister` and a `WriteRegister` functions.

   Note: After processing a system call, your code in `exception.cc` must also advance the program counter, which is stored in the `PCReg` register. The assembly language `NachOS` simulates is MIPS-based, which means every instruction is exactly 4 bytes long.
2. Arguments to the system calls are addresses in "user space". That is, they are logical (virtual memory) addresses, not "physical addresses". This means `NachOS` can't access them directly – they must be translated. To do this, you will need to use the `ReadMem` and `WriteMem` functions in `machine/translate.h`.

To aid in implementing your system calls. You need to write four memory helper functions.

Kernel Memory Functions

1. `int libReadString(char* from, char* to, int max)` Read at most `max` characters from MIPS memory into an UNIX buffer, stops reading at the `'\0'` character. Returns the number of characters read or -1 if there was an error.
2. `int libReadBytes(char* from, char* to, int max)` Read at most `max` bytes from MIPS memory into an UNIX buffer. Returns the number of bytes read or -1 if there was an error.

3. `int libWriteBytes(char* from, char* to, int max)` Write at most `max` bytes from an UNIX buffer to MIPS memory. Returns the number of bytes written or -1 if there was an error.

4. `int libWriteString(char* from, char* to, int max)` Write at most `max` characters from an UNIX buffer to MIPS memory, stops writing at the `'\0'` character. Returns the number of bytes written or -1 if there was an error.

Note that you 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 (other than calling **Halt** directly).

2 Files

Basic file I/O is handled by system calls.

Create

Create is one of the easiest system calls to implement and test. To implement **Create**, you should use the `Create` function of the `FileSystem` class in `filesys/filesys.h`. You can test it by creating a program in the `test/` folder that calls the create system call (use the code in `halt.c` as a model – don't forget to update the Makefile in `test/` to build your test case). You can then use `ls` at the Linux command prompt to see if the file has been created in your `userprog/` directory.

Until you implement **Read** and **Write** and the `SynchConsole` class, you won't have an easy way to print debugging statements. If you get **Create** working (and tested), you can kind of hack around this by using the **Create** system call to create files at different points in your MIPS test code.

Open

To implement **Open**, you should use the `Open` function of the `FileSystem` class which returns an `OpenFile` pointer. You will need to store that pointer in some kind of **in each address space** data structure. A vector or an array of `OpenFile` pointers works well. You should return a unique integer id that can be used later to retrieve the pointer from your data structure. The **Read** and **Write** functions will use that id number to access the pointer.

Close

There is no "Close" function in the `OpenFile` class. Instead, the destructor closes the file. Your **Close** system call should delete the corresponding `OpenFile` object and remove it from the file table data structure. You should implement this in a way that makes it possible to open at least 254 simultaneous files, close them, and open 254 more files. That is, we should be able to re-use closed file descriptors.

Read and Write

The read and write system calls are used for two separate tasks. If passed a file descriptor (`OpenFileId`) of `ConsoleInput (0)` or `ConsoleOutput (1)`, they are used for Console I/O. If passed a file descriptor of two or more, they read and write to a file. To do file read and write, you should look up the file descriptor in your file table and then use the `Read` and `Write` functions in `filesys/openfile.h` to get input or produce output to a file.

3 Console

In order to really test your code, you need to be able to print to the console. Add if statements to your **Read** and **Write** functions so that if they are passed the `ConsoleInput` or `ConsoleOutput` ids (respectively) they will read from or write to the console instead of a file.

`NachOS` provides a `Console` class which will allow you to read or write a single character at a time from the console, but access to this console device is currently "Asynchronous", which means that if two threads print a line, that line can be interrupted, causing a race condition.

Also, access to the `Console` is a producer/consumer system – we don't want to read from the console buffer until data is actually there (otherwise, we'll get garbage) and we don't want to write to the console while the previous character is still being output (otherwise, we might clobber it and not print everything).

To support Console Read and Write, create a `SynchConsole` class that provides the abstraction of synchronous access to the console. Then create a global `SynchConsole` object in `threads/system.cc` which your system calls can use. The file `progtest.cc` has some example code for using the `Console` class. You can use this as the basis for the functions in your `SynchConsole`. I highly recommend using Locks and Conditions.

UPDATE 10/06

Implement the following functions in your `SynchConsole`:

1. `int readLine(char* into, int maxLen)` – reads up to a new line, carriage return, end of file, null character, or maxLen. (think getLine)
2. `void writeString(char* from, int maxLen)` – writes up to maxLen characters or the null character

4 Processes

You also need to implement the remaining system calls (**Exit**, **Join**, and **Exec**). These system calls mostly involve process control. You will need to implement a `Process` structure/class, as well as a "process table". The process table will map "Process IDs" to `Process` instances. Each `Process` should keep track of its child processes and its parent process. You may also need to give `AddrSpace` objects a way to keep track of a PID. Process IDs should be able to be reused. A `Process` will need to know the PIDs of its children and its parent.

Exit

The **Exit** system call causes a thread to exit (calls `Thread::Finish`) and sets the process' exit status.The exit status is an integer value which is set by the **Exit** system call and retrieved (in another thread) by the **Join** system call.

The **Exit** system call should clean up any unneeded process data structures. Note that a `Process` cannot be removed from the process table until there is no chance of a **Join** being called on it.

Join

This is called by a process (the parent) to wait for a child process to exit. If the parent is still active, then **Join** blocks until the child exits. When the child process has exited, the **Join** call returns the child's exit status. You may assume that **Join** will be called on a process at most once.

This is different from `Thread::join`.

Exec

Exec in `NachOS` is more like the exec syscall in Windows than in Linux. It both creates a new thread (with a new address space) and loads a program from disk into the address space for that thread. This is very similar to the "StartProcess" function in `userprog/progtest.cc`. The parent thread continues running as before.

Exec returns the process id of the new thread (so that the parent process can call **Join** on it later).

5 Multiprogramming

Every thread in `NachOS` is given its own `AddrSpace` (address space) object which keeps track of which pages of physical memory have been allocated to it. Right now, we assume that only one process is running at a time.

Implement multiprogramming with time-slicing. Most of the code will probably go either in your `StartProcess` function or in the `Addrspace` class.

UPDATE 10/06

You will need to:

1. create a new `Page` structure containing a `TranslationEntry` and a byte buffer of size `PageSize`
2. write a function `Page AddrSpace::Fetch(int vpn)` which will read the `Page` at *virtual page number VPN* from the executable. This will involve reading segments. A page may contain bytes from multiple segments. The `Page` may be in the `uninit` segment or in the `stack`. (Both should be zeroed out)
3. move loading the executable from the `AddrSpace::AddrSpace()` into a new function `bool AddrSpace::LoadExecutable(char* filename)`
4. write a function `bool AddrSpace::Place(Page* p)` which places the `Page` into main memory, updates the `pageTable`, and `p`. If the `Page` cannot be placed return false.(The data structure in `bitmap.h` is useful)
5. use timer interrupts to force threads to yield after a certain number of ticks. This can be done in the `TimerInterruptHandler` function you used for your alarm clock. Note that `scheduler.cc` saves and restores user machine state on context switches automatically.

6 Individual Tasks

Exec with arguments

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 this feature! (You will need to pass the arguments on the stack to `main`)

Shell and strings

`NachOS` provides a "shell" program that provides you with a simple command prompt, but very little other functionality. Rewrite the program for readability and future expansion.

Create a library of functions, `nachosLib.h`, to be used in your `test` directory. Your test programs can `#include` your library to have access to the functions. **(You can use a different prefix)**

1. `int nac_strlen(char* str)` – return the length of the string str
2. `void nac_strcpy(char* destination, char* source)` – copies source into destination
3. `void nac_strcat(char* destination, char* source)` – concatenates source onto destination
4. `char* nac_strstr (char* search,char* token)` – returns a pointer to the first occurence in search of the characters in token, NULL if none
5. `int nac_atoi(char* str)` – converts str into an int, if not a valid integer behavior is undefined
6. `void nac_itoa(int i, char* str)` – converts i to string representation

Threads

Implement multi-threaded user programs. Implement the thread fork and yield system calls, to allow a user program to fork a thread to call a routine in the same address space, and then ping-pong between threads.

Yield The **Yield** system call simply causes the current thread to yield the CPU. (Woot, the easiest system call!)

Fork When a process calls **Fork**, you must allocate a new `AddrSpace` for it and then call `Thread::Fork`. The new address space should have its own set of stack pages, distinct from those in the parent process. It should share all other pages (including those for the code segment and the data segment) with its parent process. This will require you to write a new address space constructor.

`NachOS` systems with thread support should define the semantics of **Exit** in the following way. **Exit** indicates that the calling thread is finished; if the calling thread is the last thread in the process then it causes the entire process to exit (e.g., wake up parent if any). The status value reported by the last thread in the process to call **Exit** shall be deemed the exit status of the process, to be returned as the result from **Join**.

7 Handing in

Create a README.txt file which describes what each group member has done. You should include which files have been changed and why. If you were unable to get something to work, explain the problem. The documentation is an important part of the project.

This lab will be due on 10/15/2021 at 11:59pm.

This assignment is adapted from one by Dr. Robert Marmorstein and this lab, and all the other `NachOS` labs were derived from the original `NachOS` projects by Tom Anderson at UC Berkeley. They have been modified to fit our lab environment and changes in the compilation software since `NachOS` was originally published.