CMPS 455

October 12, 2021

Project 3 - Multiprogramming Notes

From *A Roadmap Through nachos*

6.3 Multiprogramming

- `Exec()` - creates a new address space, reads a binary program into it, and then creates a new thread (via `Thread::Fork`) to run it. (Warning: when opening files, Nachos opens all files for both reading and writing. Thus, the binary being loaded off of disk must be writable.)

- `Fork()` - creates a new thread of control executing in an existing address space.

- When a nachos program issues `Exec()` system call, the parent process is executing the code associated with the `Exec()` call. At some point during the `Exec()` call, the parent will need to invoke `Thread::Fork` to create the new thread that executes the child process. Careful thought is required in deciding at what point the `Fork()` operation should be done. Specifically, after `Fork()`, the (new) child thread can no longer access variables associated with the parent process. For example, if the parent copies the name of the new file to be executed into a variable, then issues a `Fork()`, expecting the child to open that file, a race condition exists. If the child runs after the parent, the parent may already have deleted (or changed) the variable holding that file name. This race condition appears in a number of similar contexts where two processes exchange data. The problem is that the parent and child threads need to synchronize with each other, so that the parent doesn't reclaim or reuse memory before the child is finished accessing it.

- When the MIPS simulator "traps" to Nachos via the `RaiseException()` routine, the program counter (`PCReg`) points point to the instruction that caused the trap. That is, `PCReg` will not have been updated to point to the next instruction. This is the correct behavior. When the fault is due to (say) a page missing, for example, the faulting instruction will need to be re-executed after the missing page is brought into memory. If `PCReg` had already been updated, there would be no way of knowing which instruction to re-execute. On the other hand, when a user program invokes a system call via the "`syscall`" instruction, re-executing that instruction after returning from the system call leads to an infinite system call loop. Thus, as part of executing a system call, the exception handler code in Nachos needs to update `PCReg` to point to the instruction following the "`syscall`" instruction. Updating `PCReg` is not as trivial as first seems, however, due to the possibility of delayed branches. The following code properly updates the program counter. In particular, not properly updating `NextPCReg` can lead to improper execution.

```
pc = machine->ReadRegister(PCReg);
machine->WriteRegister(PrevPCReg, pc);
pc = machine->ReadRegister(NextPCReg);
machine->WriteRegister(PCReg, pc);
pc += 4;
machine->WriteRegister(NextPCReg, pc);
```

Use of the following test programs greatly helped convince me that my implementation was (finally) correct:

`ConsoleA` - Simple program that simple loops, writing the character ’A’ to the console. Likewise, `ConsoleB` loops, printing the character ’B’. shell - The provided shell starts a process and then issues a ”Join” to wait for it to terminate. This test multiprogramming in a very rudimentary fashion only, because the shell and the invoked program rarely execute concurrently in practice. Modify the shell so that in can run jobs in background. For example, if the first character of the file is an ””, start the process as normal, but don’t invoke a ”Join”. Instead, prompt the user for the next command. With the modified shell, run the `ConsoleA` and `ConsoleB` programs concurrently.

4 User-Level Processes

Nachos runs user programs in their own private address space. Nachos can run any MIPS binary, assuming that it restricts itself to only making system calls that Nachos understands. Nachos requires that executables be in ”Noff” format. To convert binaries of one format to the other, use the ”coff2noff” program. Consult the `Makefile` in the `test` directory for details.

`Noff-format files` consists of four parts: -`Noff header` - describes the contents of the rest of the file, giving information about the program’s instructions, initialized variables and uninitialized variables. -`Noff header` resides at the very start of the file and contains pointers to the remaining sections. Specifically, the `Noff header` contains: -`noffMagic` - a reserved ”magic” number that indicates that the file is in Noff format. The magic number is stored in the **first 4 bytes** of the file. Before attempting to execute a user-program, Nachos checks the magic number to be sure that the file about to be executed is actually a Nachos executable. -`virtualAddr` - what virtual address that segment begins at (normally 0) -`inFileAddr` - pointer within the Noff file where that section actually begins (so that Nachos can read it into memory before execution begins) -`size` - the size (in bytes) of that segment

When executing a program, Nachos creates an address space and copies the contents of the instruction and initialized variable segments into the address space. Note that the uninitialized variable section does not need to be read from the file. Since it is defined to contain all zeros. Nachos simple allocates memory for it within the address space of the Nachos process and zeros it out.

4.1 "Process Creation"

Nachos proccesses are formed by creating an address space, allocating physical memory for the address space, loading the contents of the executable into physical memory, initializing registers and address translation tables, and then invoking `machine::Run()` to start execution. `Run()` simply ”turns on” the MIPS machine, entering an infinite loop that executes instructions one at a time.

Stock Nachos assumes that only a single user program exists at a given time. Thus, when an address space is created, Nachos assumes that no one else is using physical memory and simple zeros out all of physical memory (e.g., the `mainMemory` character array). Nachos then reads the binary into physical memory starting at location `mainMemory` and initializes the translation tables to do a one-to-one mapping between `virtual` and `physical` addresses (e.g., so that any virtual address N maps directly into physical address N). Initialization of registers consists of zeroing them all out, setting `PCReg` and `NextPCReg` to **0 and 4 respectively**, and setting the `stackpointer` to the **largest virtual address of the process** (the stack grows downward towards the heap and text). Nachos assumes that execution of user-programs begins at the first instruction in the text segment (e.g., virtual address 0).

When support for multiple user processes has been added, two other Nachos routines are necessary for process switching. Whenever the current process is suspended (e.g., preempted or put to sleep), the scheduler invokes the routine `AddrSpace::SaveUserState()`, in order to properly save address-space related state that the low-level thread switching routines do not know about. This becomes necessary when using virtual memory; when switching from one process to another, a new set of address translation tables needs to be loaded. The Nachos scheduler calls `SaveUserState()` whenever it is about to preempt one thread and switch to another. Likewise, before switching to a new thread, the Nachos scheduler invokes `AddrSpace::RestoreUserState`. `RestoreUserState()` insures the proper address translation tables are loaded before execution resumes.