CS162: Operating Systems **Fall 2005**

Section: Intro to Processes and Threads

TA: Rajesh Nishtala 09/07/05

1 Announcements

- Email: cs162-tc@imail.eecs.berkeley.edu
- Office hours are Monday from 4-5 in 511 Soda and Thursday 10-11 511 Soda
- Make sure that you log into your account
- Group registration deadline is tonight 11:59pm
- Section assignments will be posted soon after
- The web site will be the primary source of information for the class Any changes will be put up there.
- Make sure you can access the newsgroup. A lot of conceptual and project questions will be answered there.
- See the syllabus for grading and such.
- Make sure you get the textbook (6th or 7th edition) and the reader available at the northside copy central.
- Section is designed to be interactive, so bring questions!
- Any admin questions?

Disclaimer: These notes aren't substitutes for attending lecture or reading the book. Those have been created with careful revision and review. These notes are mainly intended to be a more thorough index of the lectures and reading so that you can go off and read the full treatment on the topic and be able to ask the right questions while reading those two sources. If you have comments or suggestions please send me email and I'll fix them up.

2 Recap of Last week

any questions? Last week had a lot of fun historical fact about operating system design and evolution. I think the main concept to take out of that is that there are multiple devices (such as the iPod, cell phone, and laptop example in class) that have different design requirements and different design considerations. Keep in mind how you'd change different parts of the OS to suit these other types of systems.

Here are a set of some interesting review questions from last week.

Question: What is an operating system?

Question: What is the difference between a kernel mode and user mode?

Question: What are the different parts of an operating system?

Question: Where is the operating system in relation to the hardware/software boundary?

The prof gave a very nice succinct definition of a what an OS does.

- An operating system is an illusionist and provides the illusion of virtual machines
- An operating system acts as government to make sure two processes don't harm each other and interact in a well ordered manner
- An operating system provides a vehicle to teach complex system design and understand various tradeoffs (i.e. what threading model and what scheduling algorithms to use)
- An operating system can provide a centralized location where a machine can self adapt and optimize its operation based on history and what the user driven workload is..

There are other lists in the notes that i think provide good insight into what an OS does and what the components are. It'll be a worthy exercise to familiarize yourself with those . . . they might be good and easy midterm questions.

2.1 Virtual Machines

First off, I thought the concept of a Virtual Machine was very important. Remember that is one of the fundamental things that the operating system provides. Every process thinks that it is running on dedicated hardware devoted to it and its closest friends. Little does it know that other things are running in the background.

2.2 Dual Mode operation

Another key idea in OS is Dual-mode operation. Remember that the hardware is straightforward and doesn't know what is running on top of it. The processor just gets a stream of instructions and executes them as fast as possible and modifies the memory according to what the instructions say. And also remember that the physical resources are fixed. Therefore the operating system, in order to play the role of the illusionist and government, must be able to affect the system's physical , and often, limited resources. In order to cast the illusion some level of the software stack must have full access to the machine. We don't want to expose it at user level because programs could be malicious, so we write a trusted piece of code (the OS) and have that affect the machine's sensitive parts (i.e. Kernel mode). For a normal operation we want to the users to be able to write their own code to provide extensibility but make sure that these programs don't interfere with each other. So as these user programs run the OS oversees their operation and makes sure that they aren't doing any illegal (i.e. user mode).

3 Processes

3.1 What is a Process

A **process** is a program in execution. With it has associated the following information

- Code section: compiled object code which has the instructions that will get run (for example compiled MIPS code from CS61c)
- PC: the current program counter and where in the code section the program is in
- Stack: space for local function calls
- Data: space for global data
- Heap: space for dynamically allocated data

Having all that information is enough to run a program and return the result of that program.

Question: What if we are guaranteed that there is exactly one program that will be running at a time, do we still need an OS?

In the first week of class we mentioned that the operating system provides a clean abstraction about infinite memory and sole access to the processor. With this abstraction we don't need to constantly adjust addresses in code segments (i.e. jump targets) depending on what part of the memory we get. We'll go into much more detail about address translation and how all this works much later in the semester. For now realize that explicitly specifying actual addresses in code segments is a very bad idea from both a security and portability standpoint. We'll see why later.

3.2 Process Execution

In this section we'll look at how a process is executed and what information is needed to switch back and forth between different running processes. Again most of this stuff will be covered in much more detail as the semester goes on. This is just section is just intended to give you a brief overview to give you context.

One of the other overarching goals to keep in mind is that we want to be able to execute multiple processes simultaneously.

Question: How is this important when there is only one physical CPU?

Question: Why not let processes handle their own scheduling?

Another important role of the Operating System is fairness. We often run multiple processes. (What if your MP3s keep stuttering because another processor doesn't fairly relinquish its grasp of the CPU, that'll be really annoying!). Thus we need to construct mechanisms so that multiple processes can share one single CPU. In order to effectively switch between different processes we need to save the state.

Question: If you are running multiple processes what other meta-data do you need to keep to be able to switch between processes?

Related Question: How much state is enough state to save a program executing so that it can be executed later?

(Hint:)Think back to CS61C and how programs got run and what the simulator kept track of.

According to the book you'll need to save the following meta-data:

- Process state in the process life-cycle (discussed below)
- Program counter to make sure that we pick up the code where we left off
- CPU registers so that when the process restores it appears as if the world has not been changed.
- CPU scheduling information (we'll go into much more detail later)
- Memory-management information to preserve translations (again discussed later)
- Accounting information which keeps track of how much this process has gotten to run. This can be used to implement fairness
- I/O status just to see what I/O devices the process is using
- **Question: A**nything else?

3.2.1 Process Life-cycle

Since multiple programs can be running at one shot we need a way for the different processes to be scheduled and a mechanism to tell the system what processes are ready to go and what processes it doesn't make sense to reschedule because they are waiting on external events. Again the concept of waiting on external events will be discussed later in the term.

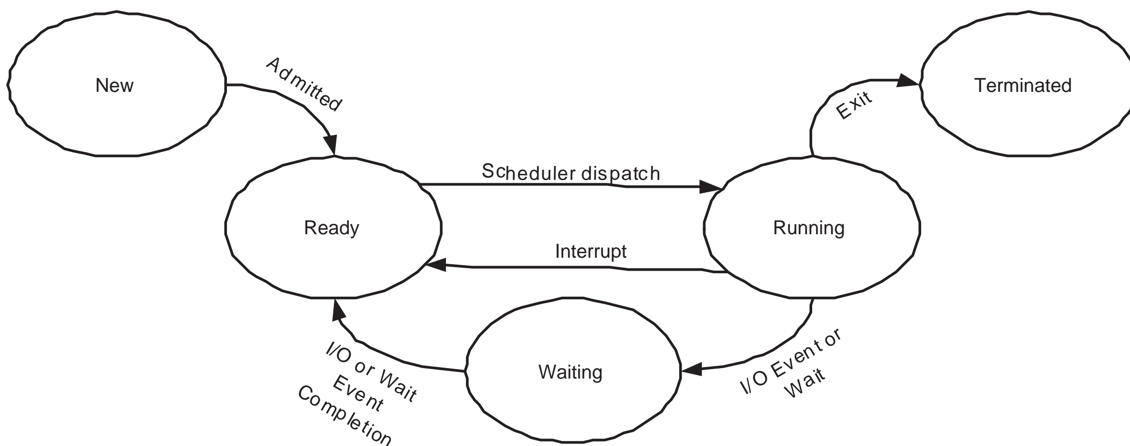

Figure 1: Lifecycle of a process from S&G chapter 3

Figure 1 shows the process state transition diagram for any process. Notice that there is a loop between ready, run, and wait. This implies that a process currently in "execution" can be in one of these three states and the interplay about when a process moves between these states is one of the key functions that an operating system delivers.

Question: Why does the Operating System need to provide this service?

The goal of most operating systems and scheduler subsystems is to maximize CPU utilization. There are many different scheduler types that are applicable for different application mixes. This an interesting research and design question, and one that you guys will be very familiar with by the end of project 2.

Question: How would the scheduler differ in a laptop versus an iPod?

Remember that only one process can be in the *running* state per physical processor at any given time.

3.2.2 Context Switch and Time slicing

In most modern computers the number of running processes will be far greater than the number of physical CPUs. Therefore we need to constantly switch between different processes so it appears that all the processes are making forward progress. A **Context Switch** is the mechanism that allows this switch between two processes (or threads). To perform the context-switch we stop the running process, save its state, load the new process' state and restart where the new process's PC told it to start.

Question: Should the context switch operation be in the user or kernel space? What are the pros and cons of each?

Question: How often do you want to switch?

Many processors and OSes rely on the concept of time slicing to give a fair amount of work to each process. Each process gets to run for k milliseconds before the processor and the operating system stop it and switch the process to another one. Thus in this k millisecond window you have full control (to the extent that the OS allows) to the underlying system before the OS stops you and gives the machine resources to someone else. If you block for I/O or another event that makes you wait (so you go into the wait state) or voluntarily give up control (so you go back on to the ready queue), then the scheduler gets to pick something else that is ready to run and the cycle continues.

3.3 Process Creation

In any operating system there is always a root process that gets created by the operating system itself. From then on, other processes are created by doing performing a fork operation. A fork spawns a new process and sets it as the child of the process that forked it. In this manner the processes are arranged in a tree structure. There are many interesting questions on what gets inherited and what doesn't by the child. Once a child forks there are a couple of options about how the parent runs. The first is to let the parent continue running to completion or until a join. The second is for the parent to wait until all the children have terminated.

Question: What are the advantages and disadvantages of each scheme?

The next question that needs to be answered about the fork is what state of the parent's process do we want to inherit.

Question: Again there are pros and cons to inheriting all, some, or none of the state. What are they?

These questions might seem very high-level right now, but by the end of the nachos project, you'll be intimately familiar with the nuances of the process creation.

3.4 Interprocess communication

So far we have talked about processes as being independent and unique entities that act on their own, however that always isn't the case. What if you want to communicate information from one process to another (such as a database to a web-server). In this section we'll briefly touch on the different techniques to be able to do this. This is one section in which I think the book makes much ado about two simple methods of communication. You can read the book for a much more thorough treatment of the idea.

3.4.1 Shared Memory

The first big idea to communicate between two processes is to use shared memory. Think back to that translation diagram that showed different processes mapping to different parts of the physical memory. What if the operating system and the processes that want to communicate agreed that they should set up an area of shared memory (i.e. through address translations etc, that we'll discuss later)? Then the operating system will allocate the space properly and make give all the processes that asked for shared data a pointer into this newly created shared space.

Question: What constructs do we need to make sure that these two processes don't stomp on each other's toes. Do we need anything?

3.5 Message Passing

Another way to have two processes talk to each other is a lot more intuitive. Imagine that every process has a set of mailboxes that it can receive data into. In order to send a message from one process to another you simply specify the mailbox of the remote process and send that message into that mailbox. It is the remote processes' responsibility to check that mailbox to see if it has anything to do. Very similar to people conducting work (i.e.. editing code) over email.

Question: What are the advantages and disadvantages of this method?

Related Question: When would shared memory work better than message passing and when would message passing work better than shared memory?

3.5.1 Remote Procedure Calls and Remote Method invocations

Remote Procedure Calls (RPC) and its java counterpart, remote method invocations(RMI), are ways in which two processes can specialize in their tasks and can publish what tasks

they are good at. The book has an exact definition of this so I'm just going to try to let it sink in with an example.

Lets say we want to set up a cool stereo in our apartment. On the desktop, we are running iTunes connected to a top-of-the line sound card and speakers. But lets say you're in bed and you really don't want to go all the way over to your computer to control the music. Lets say you also have access to a laptop in your bed that has the same iTunes interface as your desktop but doesn't have access to any of the songs nor the hardware. In that case what you can do is set up a system in which the laptop can control the desktop's music. When you send a request for a certain song to play from your laptop you in essence are doing an RPC. The laptop itself is very dumb about playing music. All it knows is some process somewhere (i.e. the desktop) that has the capability to understand the function *playSong()*. So you tell the desktop "hey play this song for me." The desktop then sees that request and understands what to do with that procedure request and actually performs the request and blasts the music through your top of the line audio equipment. This might seem a bit contrived but there are many research projects that are exploring how to do this cleanly.

4 Threads

From the previous section we see that we can get multiprogramming through just using processes, for and interprocess communication. However is this really the best way? Lets revisit the context switch operation. This is a very heavy weight operation, not only because its a complex set of operations to perform, but because of its side effects on the other parts of the machine. So constantly switching back and forth will be too slow. Interprocess communication also seems like a big sledge hammer when you want to send a few bytes to a neighboring process. In addition, we are at the mercy of the operating system scheduler which might be optimized for global fairness (across machine) rather than local fairness (one thread getting optimum time slicing).

All these reasons warrant a new construct that allows us to enable multiprogramming but without the heavyweight constructs that the operating system provides. In the previous section we assumed that there was one code sequence that was executing per process. What happens if we relax that assumption. What if we have multiple "threads" of execution. For example in Microsoft word, do you really want one process responding to keystrokes, and another to control the spell checker, and another to control the paper clip. Imagine how slow your machine would get if you have to constantly switch between the spell checker and the typing. And also imagine for a second that the operating system has to step in and create shared regions or set up mailboxes so that the two systems can interact every time you type a word. UGH!

Now imagine a world where all three of these actions are encompassed by one single memory space and one operating system construct. The operating system just schedules "Microsoft Word" to run and then all the subparts of Microsoft Word take care of itself. And also imagine that they share by default one common memory space so that they can exchange a few bytes at a time (i.e. pass a word from the keyboard detector to the spell checker) to enable fine-grained interaction. This is what the threads abstraction gives us.

According to the book: "A thread is a basic unit of CPU utilization; it comprises of

a thread id, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other OS resources such as open files and signals." I think that this definition is correct, but it is important to add that switching between different threads is a lot easier and less of a task than switching processes because of the amount of shared state across threads within the same process.
The book lists the following as the benefits of threads:

- Responsiveness: a different part of the process can run if one part is blocked. I.e. if the paper clip is figuring out how to scratch its head, you can still keep typing rather than waiting for that graphic to finish its thing.
- Sharing: Since every "Thread" is in the same space, they by default share everything. This is very useful and provides great easy of programmability ... or is it? What constructs are needed to properly enable sharing? Think back to 61A!
- Economy: don't need to over allocate memory. A process has a lot more state than a thread, therefore multiple threads take up a lot less room.
- Utilization of architecture: many modern machines have more than one physical processor that are connected to a common memory. This is exactly the model that thread programming was designed for!

Question: Lets say we are in the same process and want to switch between different threads of control within that process, what subset of the information for the process context switch do we need? Why is it faster to switch between threads? Hard: What are the other memory side effects that thread switching avoids?

4.1 Kernel Threads Versus User Threads

Many modern operating systems also provide for threads in the kernel space that are managed by the OS and thus the scheduler is controlled by the operating system. In addition there are many libraries that provide threads at a user level that rest on top of the kernel thread constructs. Kernel threads can be swapped in and out without user intervention and all the context swaps are handled by the OS. However, since user threads are completely in user space they can rest on one or many kernel threads. Note that a kernel thread is **not** the same as a process. A process can have many kernel threads. On top of each of these kernel threads we can implement many user threads.

4.1.1 One-to-one mapping

The concept that user threads rest on top of kernel threads is a bit vague. There are different approaches to this. First there can be a one-to-one mapping between user and kernel threads. This means that when a kernel thread gets swapped out the user thread associated with it goes too.

4.1.2 Many-to-one mapping

In this case we have many user threads mapping to one kernel thread. When the kernel thread gets scheduled one or all of the user threads on top of the kernel thread get to run in the time slice allocated to the kernel thread.

4.2 Many-to-Many (dynamic) mapping

Lets say we have k user threads and n kernel threads. In this case the k user threads can get mapped to an arbitrary kernel thread in order to run. Think of it as a hybrid between the previous two schemes.

Question: So far I've just listed what the mappings are . . . what are the advantages and disadvantages of each mapping?

(Hint:)Think about how the OS scheduler and the user-defined scheduler would interplay in each case

4.3 Thread Creation/Communication

I think that that section 4.4 in the book does a good job of explaining this idea so you should refer to that.

The one part that I think requires a bit of clarification is the idea of signals and how it relates to process and thread communication. So far we have talked about shared memory and message passing for processes and shared memory for threads as the model of communication. It is important to remember that signals are not done through either of these mechanisms. They have their own separate channel of communication. The shared memory and message passing are for process-process communication while signals are mainly process-OS communication events. They are built-in operating system constructs (for example the segfault signal) that send a signal to a specific process. The process then gets to decide to use the default handler or specify a user defined handler. For example lets say you are writing a calculator, you don't want the calculator to die every time someone divides by zero. You want to signal an error but not exit. Thats where a user defined signal handler would be useful. The signal interface changes from OS to OS and are key to signalling important events, such as I/O completion, network activity, and other such interesting activity that is not under direct control of the process. In some ways its similar to message passing but there are subtle but important differences. Multiplexing the correct signal to the correct process is another mechanism which the OS uses to play the illusionist and government roles.

5 Conclusion

I think the main take away from all this should be that there are two different mechanisms that the operating system uses to run user programs: processes and threads. Multiple threads are within a process and all share the same memory space. In general switching between multiple threads between a process is a lot easier and lighter of a chore than switching between multiple processes. The state information for a process is much higher.

In order for processes or threads to talk to each other we can setup shared memory or message passing. Threads usually get shared memory for free, however its much more difficult to set it up for processes. The interplay between processes, kernel threads, user threads and how you switch back and forth is one of the main concepts of this class and by come the midterms and the final you should know it inside out. Nachos will also help a great deal with this.

Any Questions?