

Section: Threads and Processes Continued

1 Announcements

1. Welcome to your new groups
2. Who am I ... for all those that might have missed it
3. Office hours M 4-5 and Th 10-11 in 511 Soda (not sure if I wrote up the right ones last time)
4. Need to login to your accounts if you haven't done so already
5. Instructions on group creation and such are on the newsgroup and the website.
6. Register your groups and get familiar with CVS
7. Groups of 3 ... please come see me
8. nachos source distribution ... download it ... read it ... live it
9. any admin questions?
10. Section notes online. The questions are meant to be conceptual more than problem based or coding based to start to get you thinking about the questions that you should be asking yourself while you read. By the end you should have a rough understanding of the answers to all these 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 Design Documents

The main intent of a design document is to make sure that every one in your group understands the overall design of the implementation. It is also to ensure that you guys think about the solutions before you rush off and start typing code. We are going to be dealing with some very tricky coding in this class. Parallel coding can be a pain in the neck and if you don't spend time on design you sink weeks on end trying to get bad parallel code to give the correct answer. This is the "engineering and science" part of EECS. That's why I will be reading your design docs to make sure that you guys have carefully thought about

all aspects of the design. There is a sample design doc on the website that you guys can refer to get an idea of how to write one. With an ideal design doc, I should just be able to take the pseudo-code and algorithm descriptions and hack away without thinking and get it to work right the first time around. The design documents creation are the hardest (and possibly the most fun) part of creating a large system since most of the interesting thought happens here.

Here are the main things that i'll be looking for while grading these documents:

1. has everyone contributed (include at the top of each function who the primary people that worked on pieces of code were)
2. Algorithms have gone through rigorous analysis
 - detailed descriptions and/or proofs of correctness
 - Big O notation for run-time and space used (where applicable)
 - Test cases that show how you'll test the corner cases (for parallel code you can not possibly test all possible inter-leavings of the code, have you captured a minimal set of test cases that prove correctness?)
3. presentation (I'm not too picky on this, but if its digital snapshots of something written in crayon i'll be displeased)
4. identify the hardest parts of the code that will require extra time to debug

3 Nachos

Before we talk more about the actual subject matter of the class I want to say a few words about Nachos.

any admin questions on nachos? Office hours are a great place to go through the code with me.

3.1 KThread.java

Kthread.java is the class that contains the information about the life cycle diagram from last week. (see Figure 1). At home you guys should have that thread diagram next to you and walk through that diagram and see what methods correspond to the different arcs of that diagram. Once you understand that it'll really help you see what the threads are doing under the covers. There are a couple of methods that I want to highlight. Also take a look at which variables are declared static and understand why they are! The first is *join* since you'll be implementing it.

3.1.1 Join

Lets say you have two threads (A and B) running. Lets say that they are affecting the memory in a non-trivial way and lets say that A needs the result of whatever B is doing so A will call B.join. Thread A then *waits* until B is done and then continues. If I say more i'll give away the solution to this problem. Keep it simple, the solution is very small.

3.1.2 runNextThread and run

These two methods are interesting in that they are the ones that get the next thread in the ready queue and actually relinquish the control of the thread and then resume control of the CPU. They implement the context switch and the stack switch that we'll talk about later today.

3.1.3 Begin and Finish routines

These are the routines that get run before a thread starts and after a thread finishes respectively. There isn't much more to say here but don't rely on the constructor or the destroyer to do this type of stuff.

Question: Why not use the constructor and destroyer to do initialization and cleanup?

3.2 Scheduler and ThreadQueue classes

There are a bunch of different scheduler classes that inherit from the same abstract class Scheduler. Understand the role of each of the different functions. An interesting thing to note is that this returns a thread queue which can be implemented anyways and functions that act on that thread queue. Remember queues don't have to always be FIFO.

3.3 Semaphore Condition and Condition2

These classes implement the thread synchronization primitives. One of the important things to note is that these classes enable and disable interrupts freely in order to achieve the atomic operations. Remember that interrupts don't get lost when interrupts are disabled or enabled, they merely get put on a queue and processed in order in which they arrive (based on some priority scheme). Also noticed that that we disable interrupts and follow it by "restore Status" **It is very important that you restoreStatus instead of re-enabling them.** A very fair midterm question on this would be:

Question: Why must we restore interrupts rather than re-enable them? What are the subtle differences between the two?

(Hint:) think about the interrupt queue and how we handle interrupts

Interrupts and synchronization will be covered next week. Understanding the nuances is very important to this stage of the project. I won't go into more detail on these topics until next week.

3.4 TCB.java

You don't have to directly modify this file but I think understanding how it works gives you a handle on how threading is actually done with java. Notice that TCB is the wrapper around the underlying java thread that this system is built on. **DO NOT CALL synchronize** in your code anywhere. That is considered cheating. Besides it'll make you loose your mind when stuff doesn't work. Trust me. TCB provides an interesting backdrop on how the threads will interact with each other.

3.5 The other parts

I hope that once you understand how these pieces of the code work, it'll be easier to see how the rest of it fits together

4 Recap of Last Week

Any Questions?

Last week in section and lecture the main topic was threads and processes. We visited that thread state diagram last week and come back to it again today in terms of processes and such. Here is a list of what I think the main takeaways from last weeks lecture were:

1. Individual Processes do not share the same address space
2. In order for two different processes to talk to each other OS has to step in and set up the communication (which is often very expensive)
3. Think of a traditional process as a thread with all the state for maintaining its own memory space (hence the term *heavy-weight thread*)
4. And thus a process can carry with it multiple threads. It is the OS's responsibility to properly pick the thread and the process from the next thread. (more on this later)
5. Threads allow for easy communication through the shared memory.

Here are a set of review questions from last week that you should be able to answer
Question: explain how the terms relate: Process, heavyweight-thread, thread, context switch, inter-process communication, message passing, shared memory, memory space

Question: We keep harping on the fact that two threads communicate through the shared memory that their process uses, which region of memory do we use?

Question: What types of situations would you want to construct applications using processes? When would you use threads?

Question: What is the inherent problem in using shared memory as the communication medium? Why do we still use it?

5 Thread Dispatching and Context Switching

Threads have very little point if you can't switch between them. So we need to think about the different mechanisms to switch the threads. This is just a rehash of stuff from Monday's lecture. There were a few points that I think need clarification / stressing. This week we'll start exploring the requirements of the scheduler and what it means to "schedule" a thread. Note that from now i'll say schedule a thread because a process is simply a thread with extra state. So if we want to switch to a different process we simply select a thread to run within that process. For now ignore the costs of the virtual memory switch in order to do this.

5.1 The Loop

The professor mentioned the main operating system loop as follows:

```
loop
  run a thread()
  pick a new thread to run()
  save the current state()
  load the next state()
end loop
```

This loop is deceptively simple. Behind each one of those functions there are many details that are being abstracted away. However that is it. That is what an OS does. But lets talk more about one of the important sections: the switch.

5.2 Switching Threads

From previous lectures and section we saw the life cycle of a process (or a thread). Lets revisit that for a moment in Figure 1. I shall use the term thread lifecycle and process lifecycle interchangeably because we can think of a process as a memory space plus a thread of execution. We'll brush other details of initiation and completion of these two under the covers for now.

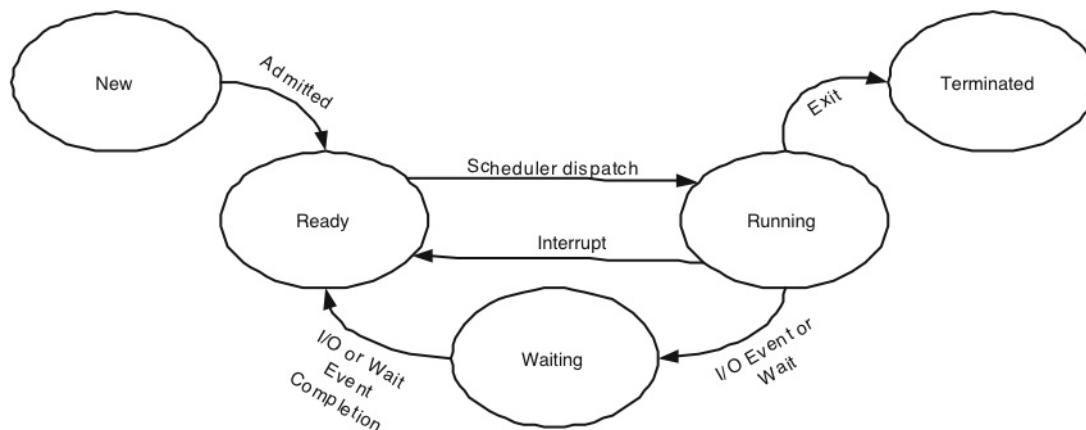


Figure 1: The life cycle of a thread

This week we'll mainly be interested in the transition from *run* to *ready* and visa-versa. The other arcs are not directly under the control of the scheduler, they are determined by external events.

Question: Is it required to only have one ready queue? Is this done for performance or correctness?

Since the scheduler has to decide what thread to run next we have to make sure that we pick the right thread for our application. But the goal of any scheduler is to make sure

that the processor is running user code most of the time and not switching threads out and back. (i.e. we don't want the scheduler to always choose the idle thread when there are more important threads)

Question: How do we ensure that the idle thread doesn't get selected while there are more important things on the queue?

(Hint:) You're writing the design doc for it very soon.

5.3 The Stack

Understanding the stack is one of the keys to understand how a stack switch actually happens. I think this is the hardest conceptual part about a context switch. The rest (managing the virtual memory translations) is routine housekeeping that we'll learn about later. So first in order to understand how the stack switch happens we need to make sure that the how the stack works. For that, turn to the CS61C text book and the sections on the growth of the stack. I'll assume that you are already familiar with that. However there are a couple of things from that book that is worth clearing up here. The first is that the stack grows (in any direction) but it grows as the layers of function calls increase. The second is that the stack pointer (which is a register in MIPS) is the only way that we know where our stack is. If we screw that up, game over.

5.4 The Switch

Let's say we are working in a world without preemption and our threads perfectly cooperate with each other. Figure 2 shows how the stacks for two threads would look in this case. Notice that the stacks grow down as indicated by the *sp* variable for each call. Lets just walk through the example. Thread A is running *My Function* and decides to share and gives up the thread. This then eventually leads to the call of `yield`, `kernel yield` and eventually `runNextThread` (which does the same thing as the Nachos `runNextThread` by picking the `nextThread`) Then the switch. Here I present the switch slightly different from what the book and lecture did. It is still an accurate drawing but I hope it stresses that the switch "function" isn't actually a function with stack pointers and such. It is actually one function calls that calls at one spot and returns at another (i.e. enters at the join of Thread A and leaves at Thread B) This way the stacks are "switched out" and a new stack is in place and then the returns from that will continue as normal. So that means that Thread B will think that the switch "function" returned and then it'll just go back up the stack to Thread B's "My Function." Note that at this point Thread A is suspended and it is suspended in a state that thinks "i'm waiting for switch to return ... " The key is that it won't return until it is Thread A's turn to run again.

This is not an easy concept ... ask questions. (*cough* *cough* midterm *cough* *cough*)

Question: How much changes if we have to deal with preemption?

Question: Hard: Lets imagine a world in which we were guaranteed that there were only two unique stacks and lets use the idea for a second that the two stacks grow in opposite directions mentioned in class, what changes need to be made to switch? Are you sure?

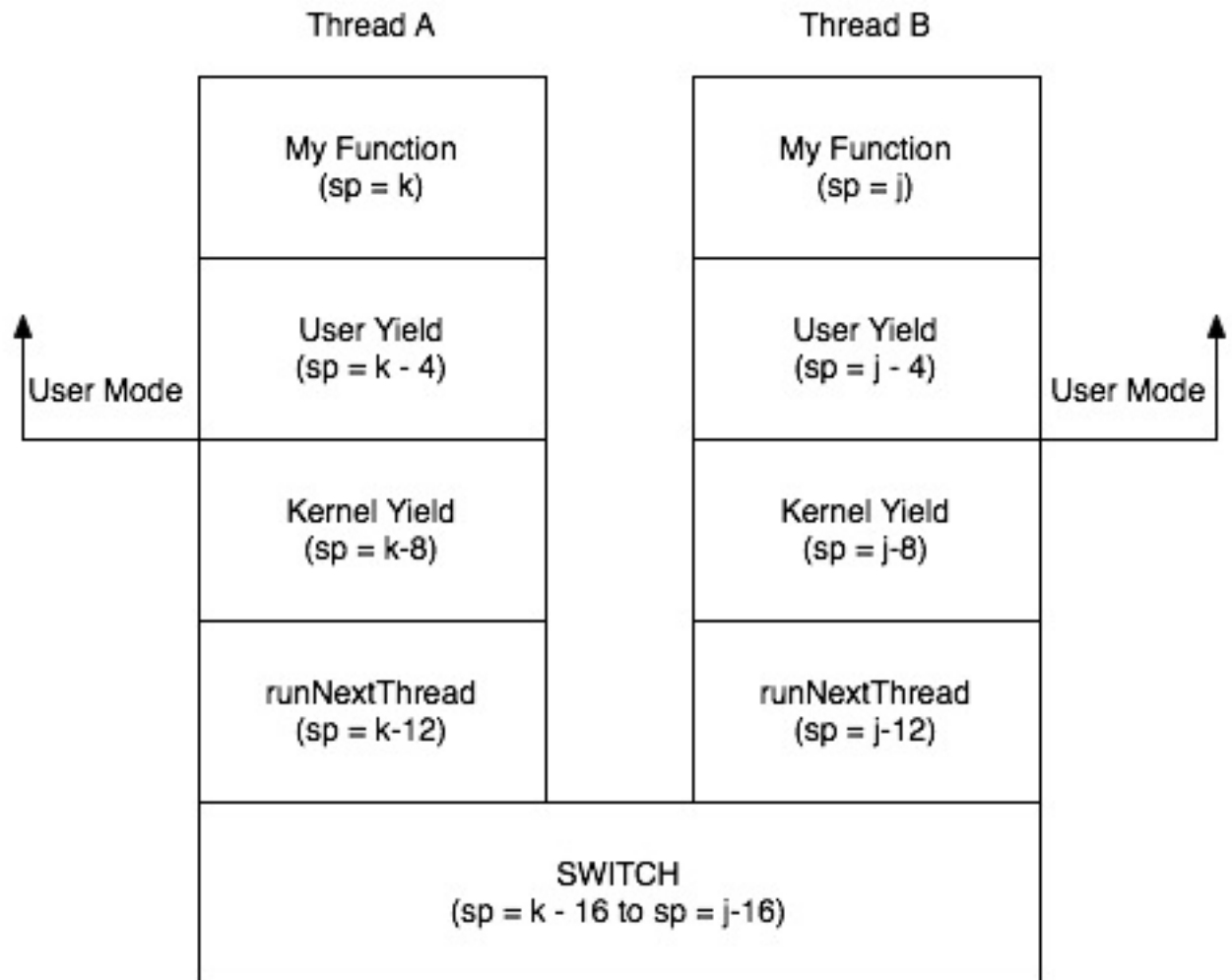


Figure 2: The stacks of two different running threads

(Hint:)think about return values from functions and CS61C

Question: Hard: Last week we mentioned user level threads and user level scheduling. For switching between user level threads can we write a switch in user space or do we need to rely on the kernel switch?

(Hint:)the kernel switches routine switches out registers ... can / how do we do that in user land? If you are really curious about this take a look at the GNU PTH. Its an interesting case study.

6 Thread and Process Creation

Again I think the book does a great job of explaining this so I won't go into much detail. I think the main thing to remember about creating processes is the idea of `fork()`. I'll refer you back to my notes last week for a more thorough treatment on the topic rather than cutting and pasting it here. Also remember that the processes are arranged in a tree structure because the fork function forks off a child. I also think that the book does a good job of talking about this so I won't bother trying to clear it up here. Note that since the processes are connected as trees, all the standard tree traversal algorithms from 170 and 61B apply.

7 Cooperating Threads

We have learned that threads are wonderful and that they are useful to perform a task by parallelizing it. Now lets explore a couple of the models that are used in order to get the threads to properly cooperate with each other. We'll learn that cooperating threads are a lovely model for programming since they encourage relationships that we'd see in the real world to get work done. However there is a HUGE drawback. This is the world of parallel programming. If you don't understand why its considered hard, don't worry, you will. I think it would also be appropriate to mention here that there is a second style of programming based on event loops. We'll go into this later in the term but realize that threads are not the only way to get some of this done. I'm going to touch upon some of the various models in which multiple threads can be used. They really aren't explicitly written in the book or the notes but I think understanding them will give you a better lead in into why some of the different synchronization mechanisms are useful.

7.1 Producer / Consumer

This is probably the simplest relationship between two threads. Thread A needs something that Thread B is producing. Thus Thread A waits for Thread B to finish producing it and Thread A consumes that resource or data. A lot of interesting examples can be built up from this model though I won't waste time with it here.

7.2 Thread Pools

Thread pools are the next simplest threading model (and perhaps the most popular). Think of the bellhops at a hotel. They are all idle and waiting for work until their boss wakes them up and the boss then tells them what to do and what room to take the luggage to. Once they complete this they go back and relax with their bellhop buddies. This is pretty much what a thread pool behaves like. There is one thread that dispatches other threads to do the work and waits for more work. Web servers work on this principle. There is one thread that spins waiting for connections. Once the connection is made it picks a thread and lets that thread handle the connection and return.

Question: Why do we want to create a pool? Is there any difference between this method and creating and destroying threads every single time?

7.3 Pipelining of Threads

This method is very similar to how instructions get run on pipelined processors that you guys learned about in CS61C. Imagine you have a Task X that can be broken up into parts A, B, C, D, and E. We first create a Thread that only understands how to accomplish part A and then create a thread that only knows how to do part B and so forth. In order to run a thread we first run it on Thread A and hand it off to Thread B etc etc until we hit Thread E when we retire the task. This way we can leverage parallelism to increase throughput of the tasks in the same way that a pipelined processor from CS61C increases throughput of instructions.

Question: Abstract: what does a pipeline hazard correspond to and do we have them?

8 Conclusion

I think that the main points that we can glean out of this week are the following:

1. Threads share address space with other threads in the same process
2. A process is a thread with more state
3. Switching between threads deals with playing with how the stack pointers
4. Process creation can take on many different forms but the common one is that processes can always be structured as trees rooted at some root process the OS provides.
5. Threads can be grouped together in different ways to yield the best interactivity, however be careful about race conditions and synchronization. They are not as easy as they look.