UNIT 1
PROCESSES AND THREADS

LECTURE 1.1 PROCESSES AND THREADS

COP4600

Dr. Matthew Gerber

1/25/2016

Processes (3.1)

- A process is a program in execution
- It's also called a *job*
- To start a process, the operating system must:
  - Load the program into main memory
  - And…

Processes (3.1)

- Let's talk about registers for a moment
  - The CPU has a bunch of them
  - Some of them are important for our purposes
  - The most important, at least for now, is the *program counter*
  - The program counter points at where the *next* instruction that's going to be executed is
  - (It doesn't point at the current instruction – that's what's already being executed)

Processes (3.1)

- So to start a process, the operating system must:
  - Load the program into main memory
  - Set the program counter to the entry point of the program
  - And…

Processes (3.1)

- The operating system needs to know some things about the process
- It needs to know some things regardless, but it *most especially* does if it's a multiprogrammed OS
- The OS stores all this information (at least logically) in a centrally located structure, so it can access it easily at any time
  - The OS is going to need to access the information a lot
- This structure is called the *Process Control Block*

Processes (3.1)

- Each PCB contains at least the following:
  - State – generally new, ready, waiting, running or terminated
  - Program counter – we talked about this before
  - Register set – what the process thinks the state of all the other registers currently is
  - Scheduling information – the process's priority, and anything else the OS uses to decide how to schedule it
  - Memory-management information – we'll get to this later
  - Accounting information – who's responsible for this process, how much time it's taken, etc.
  - I/O status information – what files and devices this process has open, what it's waiting on, etc.

Processes (3.1)

- So to start a process, the operating system must:
  - Create and initialize a PCB for the process (State: new)
  - Load the program into main memory
  - Set the PCB program counter to the entry point of the program
  - Set the PCB state to ready
- But when does the process get to actually run?

Scheduling (3.2)

- Scheduling is simply deciding which process to run…
- …from the processes that *can* be run
- It's all about queues
  - The *job queue* contains *all* the processes in the system
  - The *ready queue* contains the processes that *are* in main memory and *aren't* waiting
  - Each I/O device has a *device queue*
    - It contains the processes that are waiting on that device

Scheduling: The Schedulers (3.2.2)

- The *long-term scheduler*, in systems that have one, loads entire batch-scheduled processes from disk
  - If you can use one of these it's great, because you can get a good mix of CPU-bound and I/O-bound processes going
  - Sadly, they're useless for fully-interactive operating systems
- The *medium term scheduler*, in systems that have one, kicks processes right back out to disk
- We don't really use either of those any more except for specialized systems
- The one we care about is the *CPU scheduler*, which selects from ready processes and decides which one to run

Scheduling: Context Switching (3.2.3)

- In multitasking systems, we need to freeze processes in the middle to give others their turn
- To do so, we:
  - Use an interrupt to return control to the OS
  - Copy the frozen process's current state – including the registers, including the program counter – into its PCB
  - Change the frozen process's state from running to ready
  - Select the new process
  - Copy the new process's state from its PCB into the registers
  - Change the new process's state from ready to running
  - Return control to the new process at its current program counter
- The running process has control of the program counter and registers, but the others have their own copies ready to be loaded back in
  - Of course, context switching is an operation that costs time too…

Scheduling: Creating Processes (3.3.1)

- Every *child* process is created by a *parent* process
- When a process creates a child process, the child process will need resources
  - It may be constrained by the resources of its parent process, or may be allowed to request resources from the OS itself
- The child process can receive parameters from the parent process
- The child process may execute concurrently with the parent, or the parent may wait for it to finish
- The child process may be a duplicate of the parent, or may have a new program entirely loaded

Scheduling: Creating Processes (3.3.1)

- On UNIX-like systems:
  - fork() creates a new copy of the current process as a child
  - If you're the child, you get back 0
  - Otherwise, you get back the child's process ID
  - exec() can then be used to run a different process
  - As the parent, you can either wait() for the child, or keep right on going
- In Windows, CreateProcess() is similar in some ways, but accepts ten different parameters to account for every possible permutation in one function call
- When a process is terminated, depending on the OS and context, its children may be terminated

Inter-Process Communication (3.4)

- A process is *independent* if, apart from being managed by the OS, it doesn't affect and isn't affected by any other processes
- A process is *cooperating* otherwise
- Any process that needs to share data with other processes needs to cooperate
- Cooperating processes require an IPC mechanism
- Much, much more on this later

Threads (Chapter 4)

- Up until now we've considered a process as the fundamental unit of execution
- Each process has a single program counter that represents the next instruction it's going to execute
- In the *threading* model, processes are allowed to have any number of program counters
- This means that each *thread* needs to have its own:
  - Program counter
  - Register set
  - Call stack (whether in kernel space or user space)
- Let's look at that more carefully…

Control Blocks in the Threaded Model

Process Model

```
define pcb {
  int             process_id
  enumeration     state
  register        program_counter
  list<register>  registers
  scb             schedule
  mcb             memory
  acb             account
  iocb            io
}
```

Thread Model

```
define tcb {
  int             thread_id
  int             process_id
  enumeration     state
  register        program_counter
  list<register>  registers
}

define pcb {
  list<tcb>       threads
  scb             schedule
  mcb             memory
  acb             account
  iocb            io
}
```

Practical Use of Threads (4.2.2)

- Threads are used as a simple form of *parallelism*
  - Parallelism has entire courses dedicated to it, but for our purposes, it is *doing different parts of the same job on different processors*
- There are two fundamental types of parallelism
  - *Data parallelism* divides up data, using multiple processors to run the same code on different data slices
    - Simple example: Image processing
  - *Task parallelism* divides up code, using multiple processors to run different parts of the program
    - Simple example: Sockets

Thread Implementation (4.4)

- Differs *radically* between systems, and even between languages
- Most UNIX-like systems support the pthreads library, but…
- That's not how Linux actually implements threads
  - Linux doesn't actually *have* threads
  - It just has processes with different levels of sharing
  - Remember how we mentioned to be careful to keep track of what fork() does and does not clone?
  - clone() lets you *decide* what to clone and not clone

Thread Implementation (4.4)

- Windows, by contrast, implements threads directly
  - CreateThread() function analogous to CreateProcess()
- Java does its own thing entirely
  - Any class can be made a thread by implementing the `Runnable` interface

Thread Implementation (4.4)

- Windows, by contrast, implements threads directly
  - CreateThread() function analogous to CreateProcess()
- Java does its own thing entirely
  - Any class can be made a thread by implementing the `Runnable` interface

JUST BECAUSE YOU CAN DOESN'T MEAN YOU SHOULD

What Threads are Good (and Bad) For (4.1)

Benefits

- *Responsiveness* - Threads can allow for an application to be more responsive without having to logically interrupt its own work
- *Resource sharing* – Threads can implement parallelism without the need for message passing or shared memory
- *Scalability* – Exploitation of multiple processors
  - …and with almost all computers now having multi-core processors, this is a major concern

Issues

- *Identifying tasks* – dividing up the work in terms of operations
- *Data splitting* – dividing up the work in terms of data
- *Balancing* – making sure every thread gets enough work
- *Dependency* – managing dependent work (remember Producer/Consumer?)
- *Debugging* – debugging multithreaded processes is *much* harder

NEXT TIME: CPU SCHEDULING