Module 3 -- Concurrency Control
1/31/19

In Module 1 we said that the OS is a "society of processes" but we did not answer questions that the kernel must solve to make this so. These questions include the mechanics of implementing multiple processes on machines with one (or a small number of) CPU, synchronizing them, and using them to solve common coordination problems within operating systems. We take these questions up now.

Implementation of processes. A process consists of at least one thread and a defined memory region. A thread is an execution stream – the sequence of instructions executed by the CPU as it follows the program code. To implement many processes on a machine with a single CPU, we need a means of preserving the *process image* – a complete record of the program's entire execution state. The process image includes

its current contents in RAM,
its stack,
its Processor Status Word (PSW), and
its process control block (PCB).

The PCB stores the CPU state, often called *CPU context*, a copy of all the volatile CPU registers, including the processor status word, that we must save in order to restore a non-running process to execution later. We add a *process identifier* (PID) register to the CPU so that we know which process is running on the CPU.

The PSW is a series of bits indicating contextual operating modes for the CPU – notably, a bit selecting between user and kernel mode, a series of bits indicating the current setting of the interrupt masks, and the program counter (PC), which is the address of the next instruction to be executed. The PSW is saved as part of "return information" in a procedure call and is restored upon return from the called procedure.

We assign a PCB to each process. The OS manages all the PCBs in a data structure that it owns in private memory. The PCB contains all the necessary CPU state information of each process, including the context. We allow a process to run on the CPU for a limited period of time called a *time slice*. At the end of a time slice, we stop the CPU and copy out its context to the PCB of the process assigned to the CPU. Then we load the context of another process and let it run on the CPU for a time slice. The operation of saving the CPU context to a PCB is generically called "save stateword" **`SAVESW`**. Likewise the operation of loading the CPU context from a PCB is called "load stateword" **`LOADSW`**. The three-action sequence

```
    (SAVESW, choose next process k, LOADSW(k))
```
is called a *context switch*.

Notice that so far we have defined two main states of a process: *running (on the CPU)* and *ready (off the CPU)*. Running means the process is executing instructions and Ready means it is not running but is ready to run the next time it is selected by context switch. The list of ready processes is called the *ready list* and can be implemented by chaining together the PCB's of all the ready processes.

At the end of a time slice we use a *timer interrupt* to trigger the context switch. A hardware timer on the CPU is initialized to the time slice length at each **`LOADSW`**. When it times out, it triggers the timer interrupt. The timer interrupt handler calls the OS dispatcher (scheduler) which gets the index **`k`** of the first process in the ready list, and places the index of the running process at the end. After the dispatcher returns, the timer interrupt handler performs the context switch.

As you can see, there are many intricate details to implement process multiplexing on a CPU. Once all this is done, all the user sees is a set of concurrent processes that appear to run in parallel. The details of the multiplexing are completely hidden.

Synchronizing processes. Processes need to be constrained so that they interact properly and do not cause problems for each other. *Synchronization* names the requirement that one process stops to wait for another do something. To implement synchronization, we add a third state – *waiting* – to a process. We represent the condition for which the process is waiting as a *semaphore* **`s`**. When a process potentially needs to wait for some synchronizing condition it executes the kernel operation **`WAIT(s)`**, which forces it into the wait state until another process indicates that the condition is satisfied via the kernel call **`SIGNAL(s)`**. It is possible that the signal arrives before the receiving process waits, in which case the receiver does not have to wait.

Semaphore synchronization is provided at a very low level in the kernel (Level 2). The text book indicates another method of synchronization, the *monitor*. This is a high level language construct that systems and applications programmers use. The compiler translates a monitor specification into semaphores and condition variables. You should read the examples of monitors in the textbook. We are not going to study them in depth because they are not part of the kernel. Once you understand semaphores, understanding how monitors and other synchronization methods work will be straightforward.

Coordination patterns. Certain patterns of synchronization arise so often that we have given them names and have specified how to set up semaphores for each case. The main patterns are:

- *Mutual exclusion.* A critical section of code manipulates shared data. To prevent races and conflicts on updating the shared data, we lock the critical section with a semaphore. Alternative name: serialization, since the shared critical section code is executed to completion in one process before the next can enter.
- *Producer-Consumer.* One process generates a stream of data into a buffer; another removes it from the buffer. The producer cannot proceed if the buffer is full. The consumer cannot proceed if the buffer is empty. In between, the consumer cannot be allowed to get ahead of the producer. Buffers of this form are used throughout the OS.

- *Private semaphore.* A semaphore that can be waited on only by its owner. Used to implement **`SLEEP`** and **`WAKEUP(k)`** operations. Process **`k`** goes to sleep while another process **`j`** does a task for it; on completion process **`j`** awakens process **`k`**.
- *Readers and Writers.* A set of processes can read a file and another set can write into it. If a writer is writing, no other process can read or write. If no process is writing any number of readers can read. How do set this up so that gangs of readers or writers cannot lock out the others?
- *Deadlock Avoidance.* A deadlock is a circular wait among 2 or more processes, in which each one is waiting for another to release a resource. Common strategies for deadlock avoidance include get-all-resources-up-front-before-staring, and ordered resource locking.
- *Dining Philosophers.* A common test scenario for whether a synchronization scheme will result in deadlock. N philosophers eat spaghetti from N plates around a round table. A fork is between each pair of plates (N forks). A philosopher needs both forks next to the plate to begin eating. How to manage synchronization so there is no deadlock?

Resources

Text Ch 5, especially §5.0-1 & §5.8 (concepts around semaphores)p

Text Ch 6, especially §6.5 (deadlocks)

Parallel Computing and the OS (video recording of Frans Kaashoek's 20-min talk at the SOSP history day 2015, accessible via on-campus network)

In Resources/documents directory:

Deadlocks.pdf (supplemental slides about deadlocks)

Parallelism.pdf (overview of parallelism including semaphores)

Dijkstra-1965.pdf (original solution to the critical section problem)

Monitors-3070.pdf (overview of the monitor construct)