Process Synchronization

- So far the granularity of concurrency has been at the machine-instruction level — preemption may occur at any time between *machine-level* instructions
- But, mix in shared data or resources and we run into potentially inconsistent state caused by inappropriate interleaving of concurrent operations, since many activities need > 1 machine instruction to complete
- Technical term = *race condition* = any situation whose outcome depends on the specific order that concurrent code accesses shared data

- *Process synchronization* or *coordination* seeks to make sure that concurrent processes or threads don't interfere with each other when accessing or changing shared resources
- Some terms to help frame the problem better:
  - Segments of code that touch shared resources are called *critical sections* — thus, no two processes should be in their critical sections at the same time
  - The *critical-section problem* is the problem of designing a protocol for ensuring that cooperating processes' critical sections don't interleave
- The overall subject of synchronization leads to a number of well-known "classic problems" and solutions

Critical Section Structure

Critical-section code has the same general structure:

- An *entry section* requests and waits for permission to enter the critical section
- The *critical section* itself follows — the process should be free to touch shared resources here without fear of interference
- The *exit section* handles anything that "closes" a process's presence in its critical section
- The *remainder section* is anything else after that

Critical Section Solution Requirements

Solutions to the critical-section problem must satisfy:

- *Mutual exclusion* — Only one process in its critical section at a time; i.e., no critical-section overlap
- *Progress* (a.k.a. *deadlock-free*) — If some process wants into its critical section, then some process gets in
- *Bounded waiting* (a.k.a. *lockout-free*) — Every process that wants into its critical section eventually gets in

Note how no lockout implies no deadlock — and in practice, we also want a reasonable maximum bound

Critical Sections in the Operating System Kernel

- Note how an OS has a lot of possible race conditions, due to shared data structures and resources
- A simple solution for this is to have a *nonpreemptive kernel* — never preempt kernel-mode processes, thus eliminating OS race conditions
- A *preemptive kernel* — in which kernel-mode code *can* be preempted — is much harder to do, but is needed for real-time operations and better responsiveness, plus it eliminates the risk of excessively long kernel code activities

Windows XP/2000	nonpreemptive
Traditional Unix	nonpreemptive
Linux	nonpreemptive < 2.6, preemptive thereafter
xnu (Mac OS X/ Darwin)	"preemptible" — off by default, until a real-time process is scheduled
Solaris	preemptive
IRIX	preemptive

"Classic" Solutions

- The critical-section problem dates back to…1965!
- First documented by Dijkstra, with early solutions from Dekker, Peterson, and Lamport (that would be Leslie Lamport, who invented LaTeX)
- Solutions initially expressed in high-level languages ("software-based")
- Eventually evolved into hardware primitives to ensure true *atomicity* of key operations (i.e., operations are completely self-contained, absolutely uninterruptible)

Peterson's Solution

- G. L. Peterson's algorithm provably possesses all three critical-section solution requirements

```
/* Entry section for process i. */
flag[i] = true;                    // Process i wants in...
turn = j;                          // ...but lets process j go first.
while (flag[j] && (turn == j));    // Wait until process j isn't interested.

/* Critical section. */

/* Exit section. */
flag[i] = false;

/* Remainder section */
```

Alternatively, the turn *variable may be called* victim *or* loser*, and is set to* i *instead of j, with the loop checking for* victim == i *…equivalent, right?*

- Nice, but some hardware architectures don't satisfy the algorithm's load/store assumptions

Hardware Solutions

- In general, solutions to the critical-section problem use the concept of a *lock* — locks are *acquired* before entering a critical section, then *released* on exit
- Note how Peterson's algorithm can be decomposed into *acquire(i)* and *release(i)* functions for the entry and exit sections, respectively
- Simple approach: disable interrupts, thus "locking" into the critical section sequence
  - Sufficient for nonpreemptive kernels in single-processor architectures
  - But what if you want to support a preemptive kernel, or multiple CPUs?

Atomic Instructions

- Key idea is to accomplish certain operations *atomically* — as a single, uninterruptible unit of work
- The following instructions, if supported in hardware, can be used to build mutual exclusion algorithms:
  - *TestAndSet(target)*: Return *target* then set to *true*
  - *Swap(a, b)*: Exchange the values of *a* and *b*
- Implementable, but not trivial, in multiprocessor environments (atomicity has to be across *all* CPUs)

Semaphores

- Introduced by E.W. Dijkstra, in his seminal *Cooperating Sequential Processes* paper (see course Web site for a URL that will lead you to full PDFs of his work)
- Main components:
  - A shared integer variable *S* — the *semaphore*
  - An atomic *wait()* operation (originally *P()* in Dutch)
  - An atomic *signal()* operation (originally *V()* in Dutch)

```
wait(S) {
    while (s <= 0);
    S--;
}
```

```
signal(S) {
    S++;
}
```

- If *S* can be any integer, then *S* is a *counting semaphore*; if *S* can only be 0 or 1, then *S* is a *binary semaphore*, also known as *mutex lock* (*mut*ual *ex*clusion *lock*)

- Using a semaphore is fairly simple: define a semaphore to be shared by processes that need to be synchronized, then call *wait(S)* in the entry section and *signal(S)* in the exit section — remember that *wait()* and *signal()* must be atomic by definition
- Implementation may be tricky: as previously defined, *wait()* performs *busy waiting* — it uses CPU cycles until *S* > 0; this implementation is known as a *spinlock*
- Alternatively, *wait()* can be implemented via a *block()* operation, which places the waiting process on a queue associated with *S*; *signal()* then calls a converse *wakeup()*
- Spinlocks do avoid context switches, so the ideal choice of implementation may vary based on the specific concurrent processes involved

"Classic" Synchronization Problems

In newer software development terms, these can be viewed as well-known "use cases" in concurrent programming — they represent broad categories of more specific synchronization situations

- *Bounded buffer* — Producer process delivers items to a buffer of fixed size (*n* items maximum), and needs to wait if the buffer is full; consumer process grabs items from there, and hangs around if it is empty
- *Readers-writers* — Concurrent access to shared data, and data consistency must be maintained
- *Dining philosophers* — 5 philosophers on a round table, with 5 chopsticks in between them; philosophers are either thinking or eating, and when one needs to eat, chopsticks are picked up one at a time
  - Represents the general problem of concurrent resource allocation
  - Rules for picking up chopsticks may vary (e.g., always-left-then-right, always-both, right-then-left-if-even-otherwise-left-then-right, etc.)
  - Adds the wrinkle of "starvation" — a philosopher can only wait a limited time before "starving to death"
- And many more…there's "cigarette smokers," "sleeping barber," "Santa Claus," "dining savages," etc.

Monitors

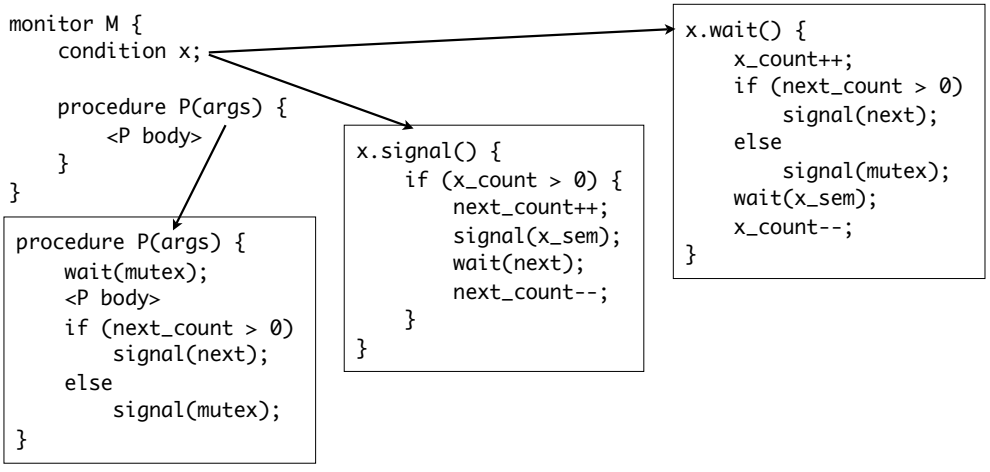
- Note that semaphores are *tools* for synchronization, and not critical-section solutions in and of themselves
- To address this, higher-level constructs facilitate proper use of these tools — monitors belong to this category
- A *monitor* is an abstraction that defines a set of operations that require mutual exclusion, along with the shared variables used by those operations
- *Conditions* in the monitor define synchronization variables, for which you can *wait()* and *signal()*

- A monitor condition's *wait()* and *signal()* are different from a semaphore's *P()* and *V()*
  - *x.wait()* suspends the calling process until another process invokes *x.signal()*
  - *x.signal()* resumes any process that is sitting on *x.wait()*; it does nothing if no processes are doing so at that time
  - Upon *x.signal()*, the signaler may either suspend itself immediately ("signal-and-wait"), or keep going until the monitor function ends ("signal-and-continue") — current monitor implementations favor signal-and-continue
- Typical monitor usage:
  - Initialize the monitor (primarily, set shared variables to initial states)
  - Launch the processes/threads that use the monitor
  - The monitor ensures that only one process at a time can be running its declared operations (i.e., mutual exclusion)
  - Processes use *x.wait()* and *x.signal()* to synchronize activities — for example, in a monitor-based dining philosophers solution, if no chopsticks are available for philosopher/process *i* (meaning *i*'s seatmates are eating), then that process must *wait()* until those seatmates put down their chopsticks and issue a *signal()*

Monitor Implementation

- Semaphores are frequently used to implement monitors; monitor-capable programming languages try to make the implementation transparent to the coder
  - For example, in Java, the *synchronized* keyword actually indicates a segment of code that must belong to some object's monitor
- Each monitor has a semaphore (say, *mutex*) that serves to protect each of its operations, whose bodies now serve as the critical sections of the compiled version; another semaphore, *next*, is also needed for signal-and-wait, so that the signaler knows when a resumed process leaves *its* monitor operation

- Each condition variable has a corresponding semaphore *x_sem* and an integer *x_count*, used by the *x.wait()* and *x.signal()* implementations



- Last implementation detail: when more than one process is waiting on the monitor, which one gets awakened? — Can be simple FIFO or conditional

Operating System Synchronization Facilities

Or, back from the abstract to the concrete…

- *Solaris*: semaphores, condition variables, *adaptive mutexes* (semaphores that spinlock or sleep, based on certain conditions — useful for multiple CPUs), *reader-writer locks* (for concurrent reads), *turnstiles* (queues for waiting threads, one per kernel thread)
- *Windows XP*: interrupt disabling in kernel, spinlocks for multiple processors; for user processes, *dispatcher objects* wrap various synchronization mechanisms (mutexes, semaphores, timers, events)

- *Linux*: preemptive kernel as of 2.6 uses spinlocks and semaphores, with reader-writer versions of them; spinlocks on SMP, then enabling/disabling of kernel preemption on single-processor systems
- *Mac OS X/Darwin*: multilayer mechanisms, with spinlocks, mutexes, and read-write locks in Mach/xnu layer, then *funnels* (construct for serializing access to non-reentrant code and synchronizing across multiple CPUs, built on Mach mutexes) at the BSD layer
- *Non-OS-specific*: *Pthreads* has mutex locks, condition variables, and read-write locks, with semaphores and spinlocks available in extensions; *Java* has the *synchronized* keyword for monitor-like functionality, with semaphores, conditions, mutex locks, etc., in 1.5