Synchronization

CSE 410, Spring 2009
Computer Systems

http://www.cs.washington.edu/410
Readings and References

• Reading

  » Chapter 6, *Operating System Concepts*, Silberschatz, Galvin, and Gagne. Read 6.1, 6.2, 6.3 (skim), 6.4-6.5, 6.6 (skim), 6.7
Synchronization

- Threads cooperate in multithreaded programs
  - to share resources, access shared data structures
  - e.g., threads accessing a memory cache in a web server
  - also, to coordinate their execution
    - e.g., a disk reader thread hands off blocks to a network writer thread through a circular buffer
Synchronization

• For correctness, we have to control this cooperation
  » must assume threads *interleave executions arbitrarily* and at *different rates*
  • Modern OS’s are preemptive
  • scheduling is not under application writers’ control (except for real-time, but that’s not of interest here).

• We control cooperation using *synchronization*
  » enables us to restrict the interleaving of executions

• Note: this also applies to processes, not just threads
  » (I may never say “process” again! Then again, I might say it a lot.)

• It also applies across machines in a distributed system
Shared resources

• We’ll focus on coordinating access to shared resources
  » basic problem:
    • two concurrent threads are accessing a shared variable
    • if the variable is read/modified/written by both threads, then access to the variable must be controlled
    • otherwise, unexpected results may occur
The classic example

- Suppose we have to implement a function to withdraw money from a bank account:

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}
```

- Now suppose that you and your S.O. share a bank account with a balance of $100.00
  - what happens if you both go to separate ATM machines, and simultaneously withdraw $10.00 from the account?
Your Bank’s Computer

• Represent the situation by creating a separate thread for each person to do the withdrawals
  » have both threads run on the same bank mainframe:

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}
```
Interleaved schedules

• The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:

```java
balance = get_balance(account);
balance -= amount;

balance = get_balance(account);
balance -= amount;
put_balance(account, balance);

put_balance(account, balance);
```

Execution sequence as seen by CPU

• What’s the account balance after this sequence?
  » who’s happy, the bank or you?
• How often is this unfortunate sequence likely to occur?
The crux of the matter

• The problem is that two concurrent threads (or processes) access a shared resource (account) without any synchronization
  » creates a race condition
    • output is non-deterministic, depends on timing

• We need mechanisms for controlling access to shared resources in the face of concurrency
  » so we can reason about the operation of programs
    • essentially, re-introducing determinism

• Synchronization is necessary for any shared data structure
  » buffers, queues, lists, hash tables, scalars, …
What resources are shared?

• Local variables are not shared
  » refer to data on the stack, each thread has its own stack
  » never pass/share/store a pointer to a local variable on another thread’s stack!

• Global variables are shared
  » stored in the static data segment, accessible by any thread

• Dynamic objects are shared
  » stored in the heap, shared if you can name it
Mutual exclusion

- We want to use mutual exclusion to synchronize access to shared resources
- Mutual exclusion makes reasoning about program behavior easier
  » making reasoning easier leads to fewer bugs
- Code that uses mutual exclusion to synchronize its execution is called a critical section
  » only one thread at a time can execute in the critical section
  » all other threads are forced to wait on entry
  » when a thread leaves a critical section, another can enter
Critical section requirements

• Critical sections have the following requirements
  » mutual exclusion
    • at most one thread is in the critical section
  » progress
    • if thread T is outside the critical section, then T cannot prevent
      thread S from entering the critical section
  » bounded waiting (no starvation)
    • if thread T is waiting on the critical section, then T will eventually
      enter the critical section
      assumes threads eventually leave critical sections
    • vs. fairness?
  » performance
    • the overhead of entering and exiting the critical section is small
      with respect to the work being done within it
Mechanisms for building critical sections

- **Locks**
  - very primitive, minimal semantics; used to build others

- **Semaphores**
  - basic, easy to get the hang of, hard to program with

- **Monitors**
  - high level, requires language support, implicit operations
  - easy (easier) to program with; Java `synchronized()` as an example

- **Messages**
  - simple model of communication and synchronization based on (atomic) transfer of data across a channel
  - direct application to distributed systems

We will survey the first three
Locks

• A lock is an object (in memory) that provides the following two operations:
  » acquire(): a thread calls this before entering a critical section
  » release(): a thread calls this after leaving a critical section

• Threads pair up calls to acquire() and release()
  » between acquire() and release(), the thread holds the lock
  » acquire() does not return until the caller holds the lock
    • at most one thread can hold a lock at a time (usually)
    » so: what can happen if the calls aren’t paired?

• Two basic flavors of locks
  » spinlock
  » blocking (a.k.a. “mutex”)
Using locks

```c
int withdraw(account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
```

- What happens when green tries to acquire the lock?
- Why is the “return” outside the critical section?
  » is this ok?
Spinlocks

• How do we implement locks? Here’s one attempt:

```c
struct lock {
    int held = 0;
}
void acquire(lock) {
    while (lock->held);
    lock->held = 1;
}
void release(lock) {
    lock->held = 0;
}
```

• Why doesn’t this work?
  » where is the race condition?

the caller “busy-waits”, or spins, for lock to be released ⇒ hence spinlock
Implementing locks (cont.)

- Problem is that implementation of locks has critical sections, too!
  - the acquire/release must be atomic
    - atomic == executes as though it could not be interrupted
    - code that executes “all or nothing”
- Need help from the hardware
  - atomic instructions
    - test-and-set, compare-and-swap, …
    - see text for examples
  - disable/reenable interrupts
    - to prevent context switches
    - crude – and can only be done in the kernel
Summary so far

• Synchronization can be provided by locks, semaphores, monitors, messages …

• Locks are the lowest-level mechanism
  » very primitive in terms of semantics – error-prone
  » implemented by spin-waiting (crude) or by disabling interrupts (also crude, and can only be done in the kernel)

• In our next exciting episode …
  » semaphores are a slightly higher level abstraction
    • less crude implementation too
  » monitors are significantly higher level
    • utilize programming language support to reduce errors
Semaphores

• Semaphore = a synchronization primitive
  » higher level of abstraction than locks
  » invented by Dijkstra in 1968, as part of the THE operating system

• A semaphore is:
  » a variable that is manipulated through two operations, P and V (Dutch for “test” and “increment”)
    • P(sem) (wait)
      block until sem > 0, then subtract 1 from sem and proceed
    • V(sem) (signal)
      add 1 to sem

• Do these operations *atomically*
Blocking in semaphores

- Each semaphore has an associated queue of threads
  - when P(sem) is called by a thread,
    - if sem was “available” (>0), decrement sem and let thread continue
    - if sem was “unavailable” (<=0), place thread on associated queue; dispatch some other runnable thread
  - when V(sem) is called by a thread
    - if thread(s) are waiting on the associated queue, unblock one place it on the ready queue might as well let the “V-ing” thread continue execution or not, depending on priority
    - otherwise (when no threads are waiting on the sem), increment sem the signal is “remembered” for next time P(sem) is called
- Semaphores thus have history
Abstract implementation

» P/wait/(sem)
  • acquire “real” mutual exclusion
    if sem is “available” (>0), decrement sem; release “real” mutual exclusion; let thread continue
    otherwise, place thread on associated queue; release “real” mutual exclusion; run some other thread

» V/signal(sem)
  • acquire “real” mutual exclusion
    if thread(s) are waiting on the associated queue, unblock one (place it on the ready queue)
    if no threads are on the queue, sem is incremented
    the signal is “remembered” for next time P(sem) is called
  • release “real” mutual exclusion
  • [the “V-ing” thread continues execution or is preempted]
Two types of semaphores

• **Binary semaphore (aka mutex semaphore)**
  » sem is initialized to 1
  » guarantees mutually exclusive access to resource (e.g., a critical section of code)
  » only one thread/process allowed entry at a time

• **Counting semaphore**
  » sem is initialized to N
    • N = number of units available
  » represents resources with many (identical) units available
  » allows threads to enter as long as more units are available
Usage

- From the programmer’s perspective, P and V on a binary semaphore are just like Acquire and Release on a lock
  
  ```
  P(sem)
  ...
  do whatever stuff requires mutual exclusion; could conceivably be a lot of code
  ...
  V(sem)
  ```

  » same lack of programming language support for correct usage

- Important differences in the underlying implementation, however
Semaphores vs. Locks

• Threads that are blocked by the semaphore P operation are placed on queues, rather than busy-waiting

• Busy-waiting may be used for the “real” mutual exclusion required to implement P and V
  » but these are very short critical sections – totally independent of program logic
Problems with semaphores (and locks)

• They can be used to solve any of the traditional synchronization problems, but:
  » semaphores are essentially shared global variables
    • can be accessed from anywhere (bad software engineering)
  » there is no connection between the semaphore and the data being controlled by it
  » used for both critical sections (mutual exclusion) and for coordination (scheduling)
  » no control over their use, no guarantee of proper usage

• Thus, they are prone to bugs
  » another (better?) approach: use programming language support
One More Approach: Monitors

• A monitor is a **programming language** construct that supports controlled access to shared data
  » synchronization code is added by the compiler

• A monitor encapsulates:
  » **shared data** structures
  » **procedures** that operate on the shared data
  » **synchronization** between concurrent threads that invoke those procedures

• Data can only be accessed from within the monitor, using the provided procedures
  » protects the data from unstructured access

• Addresses the key usability issues that arise with semaphores
A monitor

- Waiting queue of threads trying to enter the monitor
- At most one thread in monitor at a time

Shared data

Operations (methods)

Proc A
Proc B
Proc C
Monitor facilities

• “Automatic” mutual exclusion
  » only one thread can be executing inside at any time
    • thus, synchronization is implicitly associated with the monitor – it “comes for free”
  » if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor
    • more restrictive than semaphores
    • but easier to use (most of the time)

• But, there’s a problem…
Example: Bounded Buffer Scenario

- Buffer is empty
- Now what?
Example: Bounded Buffer Scenario

- Buffer is full
- Now what?
Condition variables

• A place to wait; sometimes called a rendezvous point
• “Required” for monitors
  » So useful they’re often provided even when monitors aren’t available
• Three operations on condition variables
  » wait(c)
    • release monitor lock, so somebody else can get in
    • wait for somebody else to signal condition
    • thus, condition variables have associated wait queues
  » signal(c)
    • wake up at most one waiting thread
    • if no waiting threads, signal is lost
      this is different than semaphores: no history!
  » broadcast(c)
    • wake up all waiting threads
A monitor (including CVs)

- Wait queue for cond. var.
  - buff_empty

- Waiting queue of threads trying to enter the monitor

- At most one thread in monitor at a time

- Shared data

- Operations (methods)
Bounded buffer using (Hoare) monitors

Monitor bounded_buffer {
    buffer resources[N];
    condition not_full, not_empty;

    produce(resource x) {
        if (array "resources" is full)
            wait(not_full);
        insert "x" in array "resources"
        signal(not_empty);
    }

    consume(resource *x) {
        if (array "resources" is empty)
            wait(not_empty);
        *x = get resource from array "resources"
        signal(not_full);
    }
}
Monitor Summary

• Language supports monitors
• Compiler understands them
  » compiler inserts calls to runtime routines for
    • monitor entry
    • monitor exit
    • signal
    • Wait
  » Language/object encapsulation ensures correctness
    • Sometimes! With conditions you STILL need to think about synchronization and state of monitor invariants on wait/signal
• Runtime system implements these routines
  » moves threads on and off queues
  » *ensures mutual exclusion!*