Module 7a: Classic Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization

Background

- Concurrent access to shared data may result in data inconsistency
  - where is this shared data?
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Shared-memory solution to bounded-butter problem (Chapter 4) has a race condition on the class data count.
  - what is a race condition?

Race Condition

The Producer calls

```
while (1) {
        while (count == BUFFER_SIZE)
                ; // do nothing
        // produce an item and put in nextProduced
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
}
```

Race Condition

The Consumer calls

```
while (1) {
        while (count == 0)
              ; // do nothing
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        // consume the item in nextConsumed
}
```

Race Condition

- count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving:

S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6 }
S5: consumer execute count = register2 {count = 4}

Solution to Critical-Section Problem

- 1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
  - critical section? what's that?
- 2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
  - why?
- 3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the N processes

Algorithm 1

- Threads share a common integer variable turn
- If turn == i, thread i is allowed to execute
- Does not satisfy progress requirement
  - Why?

Algorithm 2

- Add more state information
  - Boolean flags to indicate thread's interest in entering critical section
- Progress requirement still not met
  - Why?

Algorithm 3

- Combine ideas from 1 and 2
- Can this meet critical section requirements?

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts, but...
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - no, NOT nuclear powered!

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts, but...
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - no, NOT nuclear powered!
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Thread Using get-and-set Lock

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);
while (true) {
    while (lock.getAndSet(true))
        Thread.yield();
    criticalSection();
    lock.set(false);
    nonCriticalSection();
}
```

Thread Using swap Instruction

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);
// each thread has a local copy of key
HardwareData key = new HardwareData(true);

    while (true) {
            key.set(true);
            do {
                  lock.swap(key);
            }
            while (key.get() == true);
            criticalSection();
            lock.set(false);
            nonCriticalSection();
    }
```

Semaphore

- Synchronization tool that does not require busy waiting (spin lock)
- Semaphore S – integer variable
- Two standard operations modify S: acquire() and release()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
acquire(S) {
    while S <= 0
            ; // no-op
    S--;
}
release(S) {
    S++;
}
```

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore S; // initialized to 1

acquire(S);
criticalSection();
release(S);
```

Semaphore Implementation

- Must guarantee that no two processes can execute acquire() and release() on the same semaphore at the same time
- Thus implementation becomes the critical section problem
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
  - Applications may spend lots of time in critical sections
    - Performance issues addressed throughout this lecture

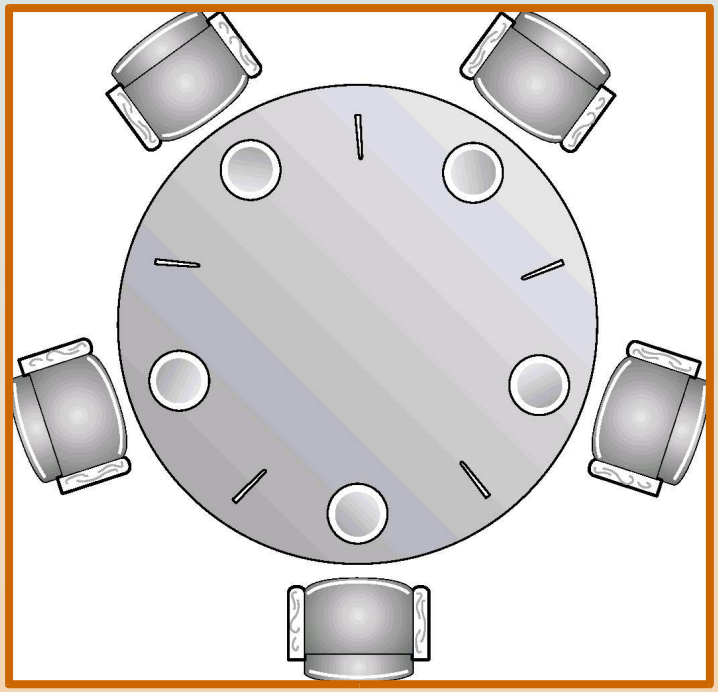
Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
acquire(S);	acquire(Q);
acquire(Q);	acquire(S);
.	.
.	.
.	.
release(S);	release(Q);
release(Q);	release(S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Dining-Philosophers Problem



Dining-Philosophers Problem (Cont.)

- Philosopher i:

```
while (true) {
    // get left chopstick
    chopStick[i].acquire();
    // get right chopstick
    chopStick[(i + 1) % 5].acquire();
    eating();
    // return left chopstick
    chopStick[i].release();
    // return right chopstick
    chopStick[(i + 1) % 5].release();
    thinking();
}
```

- What “data” is being shared? What problem can occur?
- What REAL OS problem is being modeled?

Program Demos

Classical Problems of Synchronization

- Chapter 7.6
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
- Chapter 7 Homework
  - Read Java Synchronization paper (course website)
  - Read Pg 278, problem 7.16 (Sleeping Barber Problem), and Pg 278, 7.17 (Cigarette Smokers Problem). Choose ONE of these, and:
    - Read about, research (use Google), and discuss briefly what REAL operating system synchronization challenge is modeled by the problem. Describe what synchronization methods can be used to solve the problem
  - Possible short paper topic: One-Lane Bridge Synchronization Problem
    - Describe & discuss the problem...what REAL OS challenge is modeled? How is it solved? Write a simple program that illustrates the solution.

Synchronization & Deadlocks
Chapters 7 & 8

To be continued next week...