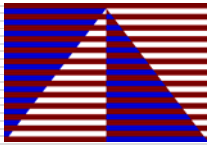
ECE3055
Computer Architecture and Operating Systems

Lecture: Process Synchronization



Prof. Hsien-Hsin Sean Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology

Process Synchronization

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

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Shared-memory solution to bounded-buffer problem has a race condition on the class data **count**.

Race Condition

The Producer calls

```
while (1) {
        while (count == BUFFER_SIZE)
                ; // do nothing

        // add an item to the buffer
        ++count;
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
    }
```

Race Condition

The Consumer calls

```
while (1) {
        while (count == 0)
                ; // do nothing

        // remove an item from the buffer
        --count;
        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
}
```


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}

Critical Section Problem

- N processes are competing shared data
- Each process has a code segment, called critical section, in which the shared data is accessed
- Goal: when one process is executing in its critical section, no other process can execute in its critical section.
- Each process looks like this

```
while (true) {
entry section
        critical section
exit section
        non critical section
}
```


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

   Safety

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

   Liveness

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

   Starvation

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - 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
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Data Structure for Hardware Solutions

```
public class HardwareData
{
	private boolean data;
	public HardwareData(boolean data) {
			this.data = data;
	}
	public boolean get() {
			return data;
	}
	public void set(boolean data) {
			this.data = data;
	}
	// Continued on Next Slide
```

Data Structure for Hardware Solutions

```
public boolean getAndSet(boolean data) {
        boolean oldValue = this.get();
        this.set(data);
        return oldValue;
}
public void swap(HardwareData other) {
        boolean temp = this.get();
        this.set(other.get());
        other.set(temp);
}
}
```

Atomic instruction (read-modify-write) — applies to the getAndSet body

Atomic instruction — applies to the swap body

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();
}
```

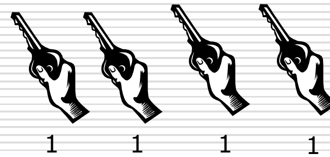
Thread Using swap Instruction

Georgia Institute of Technology

```
// 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();
    }
```

0



1 1 1 1

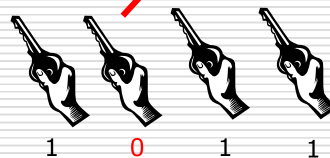
Thread Using swap Instruction

Georgia Institute of Technology

```
// 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();
    }
```

1



1 0 1 1

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

Georgia Institute of Technology

- 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

Georgia Institute of Technology

- 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.

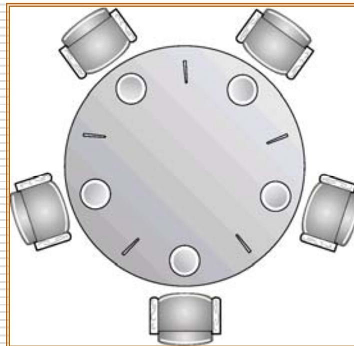
Georgia Institute of Technology

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Georgia Institute of Technology

Dining-Philosophers Problem



- Shared data

```
Semaphore chopStick[]  = new Semaphore[5];
```

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();
}
```