Process Synchronization

Notice: The slides for this lecture have been largely based on those accompanying an earlier edition of the course text Operating Systems Concepts with Java, by Silberschatz, Galvin, and Gagne. Many, if not all, the illustrations contained in this presentation come from this source.
Race Condition

A race occurs when the correctness of a program depends on one thread reaching point $x$ in its control flow before another thread reaches point $y$.

Races usually occur because programmers assume that threads will take some particular trajectory through the execution space, forgetting the golden rule that threaded programs must work correctly for any feasible trajectory.

Computer Systems
A Programmer's Perspective
Randal Bryant and David O'Hallaron

02/12/2010        CSCI 315 Operating Systems Design
The Synchronization Problem

- Concurrent access to shared data may result in data inconsistency.

- Maintaining data consistency requires mechanisms to ensure the "orderly" execution of cooperating processes.
Producer-Consumer
Race Condition

The **Producer** does:

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

02/12/2010  CSCI 315 Operating Systems Design  4
Producer-Consumer
Race Condition

The **Consumer** does:

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

- \texttt{count++} could be implemented as
  \begin{align*}
  \text{register1} &= \text{count} \\
  \text{register1} &= \text{register1} + 1 \\
  \text{count} &= \text{register1}
  \end{align*}

- \texttt{count--} could be implemented as
  \begin{align*}
  \text{register2} &= \text{count} \\
  \text{register2} &= \text{register2} - 1 \\
  \text{count} &= \text{register2}
  \end{align*}

- Consider this execution interleaving:
  \begin{align*}
  \text{S0: producer execute } \text{register1} &= \text{count} \quad \{\text{register1} = 5\} \\
  \text{S1: producer execute } \text{register1} &= \text{register1} + 1 \quad \{\text{register1} = 6\} \\
  \text{S2: consumer execute } \text{register2} &= \text{count} \quad \{\text{register2} = 5\} \\
  \text{S3: consumer execute } \text{register2} &= \text{register2} - 1 \quad \{\text{register2} = 4\} \\
  \text{S4: producer execute } \text{count} &= \text{register1} \quad \{\text{count} = 6\} \\
  \text{S5: consumer execute } \text{count} &= \text{register2} \quad \{\text{count} = 4\}
  \end{align*}
The Critical-Section Problem
Solution

1. **Mutual Exclusion** - If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections.

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.

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.)
Two-task Solution

- Two tasks, $T_0$ and $T_1$ ($T_i$ and $T_j$)
- Three solutions presented. All implement this MutualExclusion interface:

```java
public interface MutualExclusion {
    public static final int TURN_0 = 0;
    public static final int TURN_1 = 1;

    public abstract void enteringCriticalSection(int turn);
    public abstract void leavingCriticalSection(int turn);
}
```
Algorithm Factory class

Used to create two threads and to test each algorithm

```java
public class AlgorithmFactory {
    public static void main(String args[]) {
        MutualExclusion alg = new Algorithm1();
        Thread first = new Thread(new Worker("Worker 0", 0, alg));
        Thread second = new Thread(new Worker("Worker 1", 1, alg));

        first.start();
        second.start();
    }
}
```

Worker Thread

public class Worker implements Runnable
{
    private String name;
    private int id;
    private MutualExclusion mutex;

    public Worker(String name, int id, MutualExclusion mutex) {
        this.name = name;
        this.id = id;
        this.mutex = mutex;
    }
    public void run() {
        while (true) {
            mutex.enteringCriticalSection(id);
            MutualExclusionUtilities.criticalSection(name);
            mutex.leavingCriticalSection(id);
            MutualExclusionUtilities.nonCriticalSection(name);
        }
    }
}

02/12/2010          CSCI 315 Operating Systems Design
Algorithm 1

• Threads share a common integer variable `turn`.

• If `turn == i`, thread `i` is allowed to execute.

• Does not satisfy progress requirement… Why?

02/12/2010

CSCI 315 Operating Systems Design
Algorithm 1

public class Algorithm_1 implements MutualExclusion
{
    private volatile int turn;

    public Algorithm 1()
    {
        turn = TURN 0;
    }
    public void enteringCriticalSection(int t)
    {
        while (turn != t)
        {
            Thread.yield();
        }
    }
    public void leavingCriticalSection(int t)
    {
        turn = 1 - t;
    }
}
Algorithm 2

• Add more state information:
  – Boolean flags to indicate thread’s interest in entering critical section.

• Progress requirement still not met…
  Why?
Algorithm 2

```java
public class Algorithm_2 implements MutualExclusion {
    private volatile boolean flag0, flag1;
    public Algorithm 2() {
        flag0 = false; flag1 = false;
    }
    public void enteringCriticalSection(int t) {
        if (t == 0) {
            flag0 = true;
            while(flag1 == true)
                Thread.yield();
        } else {
            flag1 = true;
            while(flag0 == true)
                Thread.yield();
        }
    }
    public void leavingCriticalSection(int t) {
        if (t == 0)
            flag0 = false;
        else
            flag1 = false;
    }
}
```
Algorithm 3

• Combine ideas from 1 and 2
• Does it meet critical section requirements?
Algorithm 3

```java
public class Algorithm_3 implements MutualExclusion {
    private volatile boolean flag0;
    private volatile boolean flag1;
    private volatile int turn;
    public Algorithm_3() {
        flag0 = false;
        flag1 = false;
        turn = TURN_0;
    }
    // Continued on Next Slide
```
Algorithm 3 - enteringCriticalSection

```java
public void enteringCriticalSection(int t) {
    int other = 1 - t;
    if (t == 0) {
        flag0 = true;
        turn = other;
        while (flag1 == true && turn == other)
            Thread.yield();
    }
    else {
        flag1 = true;
        turn = other;
        while (flag0 == true && turn == other)
            Thread.yield();
    }
}
```

// Continued on Next Slide
Algo. 3 – leavingCriticalSection()

    public void leavingCriticalSection(int t) {
        if (t == 0)
            flag0 = false;
        else
            flag1 = false;
    }
}
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:
  - Test memory word and set value.
  - Swap the contents of two memory words.
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;
    }
}

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);
}

Using the **get-and-set** instruction

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

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