Process Synchronization

Notice: The slides for this lecture have been largely based on those accompanying the textbook *Operating Systems Concepts with Java*, by Silberschatz, Galvin, and Gagne (2003). Many, if not all, the illustrations contained in this presentation come from this source.
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
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++;
}
```
Producer-Consumer
Race Condition

The Consumer does:

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

- 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}
# 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, T₀ and T₁ (Tᵢ and Tⱼ)
- 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

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

    Worker(String name, int id, MutualExclusion mutex) {
        this.name = name;
        this.id = id;
        this.mutex = mutex;
    }

    void run() {
        while (true) {
            mutex.enteringCriticalSection(id);
            MutualExclusionUtilitiesriticalSection(name);
            mutex.leavingCriticalSection(id);
            MutualExclusionUtilities.nonCriticalSection(name);
        }
    }
}
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 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;
    }
}

02/11/2004 CSCI 315 Operating Systems Design 12
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

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;
    turn = other;
    if (t == 0) {
        flag0 = true;
        while(flag1 == true && turn == other)
            Thread.yield();
    }
    else {
        flag1 = true;
        while (flag0 == true && turn == other)
            Thread.yield();
    }
} // Continued on Next Slide
```
Algo. 3 –
leavingCriticalSection()

```java
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.
Using Hardware Solutions

```java
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);
    }
}
```

```java
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

// 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 can be used to modify $S$: `acquire()` and `release()` (originally called `P()` and `V()`: *proberen*, *verhogen*).

• Can only be accessed via two 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).

- Note that one can implement a counting semaphore $S$ as a binary semaphore.

- Provides **mutual exclusion**:

  ```
  Semaphore S(1); // initialized to 1
  acquire(S);
  criticalSection();
  release(S);
  ```
Semaphore Implementation

```plaintext
acquire(S) {
    value--;    
    if (value < 0) {
        add this process to list
        block;
    }
}

release(S) {
    value++;    
    if (value <= 0) {
        remove some process P
        from list
        wakeup(P);
    }
}
```
Semaphore Implementation

- Must guarantee that no two processes can execute acquire() and release() on the same semaphore at the same time.

- The 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 section
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
  
  \[
  P_0 \quad P_1
  \]
  
  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.
The *Dining-Philosophers* Problem