CSMC 412

Operating Systems
Prof. Ashok K Agrawala

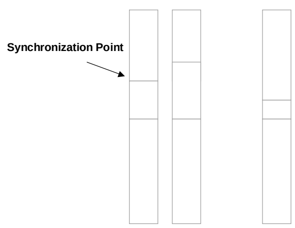
Set 7

Process Synchronization

- Assumptions
  - A collection of N independent, cooperating processes
  - A process may be blocked for arbitrary length of time without affecting its functionality
  - No knowledge about
    - » Processor speeds
    - » Structure of other processes
- Characteristics of the Solution
  - Must meet the synchronization requirements
  - Malfunction of one process must not affect the other processes
  - All processes must make reasonable progress

Synchronization Requirements

- At Synchronization Point a set of conditions must be true for the process to proceed. Else it must wait
- Conditions depend on the state of other processes
- The decision to proceed has to be taken by this process
- How does this process get the information about the state of other processes ?



State Information

- Each process has its own state information
- How should the state information of the other processes be:
  - Accessed
    - » Ask the other process?
      - Which ones? In which order? What mechanism? Implications
  - Transferred
    - » Delay – state may change in the mean time
    - » Mechanism
      - Messages
      - Global Variables

Message Passing

- Type
  - One-to-One
  - Multicasting
  - Broadcasting
- Setup
- Delay
  - State may change

Global Variables

- Shared Memory
  - Directly
  - Through Kernel services
- Concurrent access
  - Reading
  - Modifying
  - Writing

Synchronization

- Process must access the global state information and decide whether to proceed or not assuring that the action is correct with respect to the global state when it proceeds and not when it accessed the state information
- Mechanisms ???

Centralized Approach

- One single central entity retains the global information and carries out all decision making about the progress of a process.
- Every process must inform about its state at all times
- Can this approach Work???
  - Scalability
  - Robustness

Decentralized Approach

- Every process decides for itself
- Mechanisms
  - Gathering state information
  - Blocking itself
  - Release from blocked state
  - Tie breakers

Atomic Action

- Global state changes
  - No partial action is possible
- Example

  X = X + A

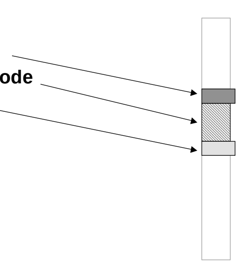
  - Is it Atomic?
  - Can it be made atomic ?

Blocking

- Busy Wait
  - Tight Loop checking for the condition
- Waiting on a queue
  - How to get there?
  - How to get out of there?

Structure

- Entry Code
- Protected Code
- Exit Code



Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Java Synchronization
- Synchronization in Solaris 2
- Synchronization in Windows NT

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-butter problem (Chapter 4) has a race condition on the class data *count*

Bounded Buffer

```
public class BoundedBuffer {
  public void enter(Object item) {
        // producer calls this method
  }

  public Object remove() {
        // consumer calls this method
  }
  // potential race condition on count
  private volatile int count;
}
```


enter() Method

```
// producer calls this method
public void enter(Object item) {
        while (count == BUFFER_SIZE)
                ; // do nothing
        // add an item to the buffer
        ++count;
        buffer[in] = item;
         in = (in + 1) % BUFFER_SIZE;
}
```


remove() Method

```
// consumer calls this method
  public Object remove() {
    Object item;
        while (count == 0)
                ; // do nothing
         // remove an item from the buffer
         --count;
         item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        return item;
  }
```


Solution to Critical-Section Problem

1. Mutual Exclusion. **If process *Pi* 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.

Worker Thread

```
public class Worker extends Thread {
  public Worker(String n, int i, MutualExclusion s) {
    name = n;
    id = i;
    shared = s;
  }
  public void run() { /* see next slide */ }

  private String name;
  private int id;
  private MutualExclusion shared;
}
```


run() Method of Worker Thread

```
public void run() {
    while (true) {
        shared.enteringCriticalSection(id);
        // in critical section code
        shared.leavingCriticalSection(id);
        // out of critical section code
    }
}
```


Mutual Exclusion Abstract Class

```
public abstract class MutualExclusion {
  public static void criticalSection() {
        // simulate the critical section
  }

  public static void nonCriticalSection() {
        // simulate the non-critical section
  }
  public abstract void enteringCriticalSection(int t);
  public abstract void leavingCriticalSection(int t);
  public static final int TURN_0 = 0;
  public static final int TURN_1 = 1;
}
```


Testing Each Algorithm

```
public class TestAlgorithm
{
  public static void main(String args[]) {
        MutualExclusion alg = new Algorithm_1();

        Worker first = new Worker("Runner 0", 0, alg);
        Worker second = new Worker("Runner 1", 1, alg);

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


Algorithm 1

```
public class Algorithm_1 extends MutualExclusion {
  public Algorithm_1() {
        turn = TURN_0;
  }
  public void enteringCriticalSection(int t) {
        while (turn != t)
              Thread.yield();
  }
  public void leavingCriticalSection(int t) {
        turn = 1 - t;
  }
  private volatile int turn;
}
```


Algorithm 1

Initialize turn = 0

- Process 0

While turn != 0 wait
Enter CS

CS Code

Exit CS
turn = 1

- Process 1

While turn != 1 wait
Enter CS

CS Code

Exit CS
turn = 0

Algorithm 2

```
public class Algorithm_2 extends MutualExclusion  {
  public Algorithm_2() {
        flag[0] = false;
        flag[1] = false;
  }
  public void enteringCriticalSection(int t) {
        // see next slide
  }
  public void leavingCriticalSection(int t) {
         flag[t] = false;
  }

  private volatile boolean[] flag = new boolean[2];
}
```


Algorithm 2 – enteringCriticalSection()

```
public void enteringCriticalSection(int t) {
   int other = 1 - t;
   flag[t] = true;
         while (flag[other] == true)
                  Thread.yield();
   }
```


Algorithm 2

Initialize both flags to false

Process 0	Process 1
Flag[0] = true	Flag[1] = true
While flag[1] = true wait	While flag[0] = true wait
Enter CS	Enter CS
CS Code	CS Code
Exit CS	Exit CS
Flag[0]=false	Flag[1] = false

Algorithm 3

```
public class Algorithm_3 extends MutualExclusion {
  public Algorithm_3() {
        flag[0] = false;
        flag[1] = false;
        turn = TURN_0;
  }
  public void enteringCriticalSection(int t) {/* see next slides */
  }
  public void leavingCriticalSection(int t) {{/* see next slides */
  }
  private volatile int turn;
  private volatile boolean[] flag = new boolean[2];
}
```


Algorithm 3 – enteringCriticalSection()

```
public void enteringCriticalSection(int t) {
   int other = 1 - t;
   flag[t] = true;
   turn = other;

   while ( (flag[other] == true) && (turn == other) )
          Thread.yield();
   }
```


Algo. 3 – leaving CriticalSection()

```
public void leavingCriticalSection(int t) {
        flag[t] = false;
}
```

Algorithm 3

Initialize both flags to false, turn = 0

- Process 0

Flag[0] = true
Turn = 1
While flag[1] = true and turn=1 wait
Enter CS

CS Code

Exit CS
Flag[0]=false

- Process 1

Flag[1] = true
Turn = 0
While flag[0] = true and turn=0 wait
Enter CS

CS Code

Exit CS
Flag[1] = false

Synchronization Hardware

```
public class HardwareData {
  public HardwareData(boolean v) {
       data = v;
  }
  public boolean get() {
       return data;
  }
  public void set(boolean v) {
       data = v;
  }
  private boolean data;
}
```


Test-and-Set Instruction (in Java)

```
public class HardwareSolution
{
   public static boolean testAndSet(HardwareData target) {
         HardwareData temp = new HardwareData(target.get());

         target.set(true);

         return temp.get();
   }
}
```


Thread using Test-and-Set

```
HardwareData lock = new HardwareData(false);

while (true) {
   while (HardwareSolution.testAndSet(lock))
         Thread.yield(); // do nothing
   // now in critical section code
   lock.set(false);
   // out of critical section
}
```


Swap instruction

```
public static void swap(HardwareData a,
   HardwareData b) {
   HardwareData temp = new HardwareData(a.get());
   a.set(b.get());
   b.set(temp.get());
}
```


Thread using Swap

```
HardwareData lock = new HardwareData(false);
HardwareData key = new HardwareData(true);

while (true) {
   key.set(true);
   do {
         HardwareSolution.swap(lock,key);
   } while (key.get() == true);
   // now in critical section code
   lock.set(false);
   // out of critical section
}
```

Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations

P* (*S*):** while *S*≤ **0** do ***no-op;
S--;

***V*(*S*): *S*++;**

Semaphore as General Synchronization Tool

Semaphore S; // initialized to 1

P(S);
CriticalSection()
V(S);

Semaphore Eliminating Busy-Waiting

```
P(S) {
  value--;
  if (value < 0) {
        add this process to list
        block
  }
}
V(S) {
  value++;
  if (value <= 0) {
        remove a process P from list
        wakeup(P);
  }
}
```


Synchronization Using Semaphores

```
public class FirstSemaphore {
  public static void main(String args[]) {
    Semaphore sem = new Semaphore(1);
    Worker[] bees = new Worker[5];

    for (int i = 0; i < 5; i++)
      bees[i] = new Worker(sem, "Worker " + (new
  Integer(i)).toString() );

    for (int i = 0; i < 5; i++)
      bees[i].start();
  }
}
```


Worker Thread

```
public class Worker extends Thread {
  public Worker(Semaphore) { sem = s;}
  public void run() {
        while (true) {
                sem.P();
                // in critical section
                sem.V();
                // out of critical section
        }
  }
  private Semaphore sem;
}
```


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
P(*S*);	*P*(*Q*);
P(*Q*);	P(*S*);
⋮	⋮
V(*S*);	V(*Q*);
V(*Q*)	V(*S*);

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