Chapter 6. Process Synchronization

- Background
- The Critical-Section Problem
- Two-task Solutions
- Synchronization Hardware
- Semaphores
- Monitors

1. Background

- Concurrent access from cooperating processes or threads to shared data may result in data inconsistency.
- Shared-memory solution to bounded-butter problem (Chapter 3) has a race condition on the class data **count**.
- ☺ *Any good idea?*
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

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;
     counter++;
}
```

Shared data

Race Condition - continued

The Consumer calls

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

Shared data

Race Condition - continued

- `count++` in Producer could be implemented as
```
register1 = count
register1 = register1 + 1
count = register1
```

- `count--` in Consumer 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}
```

- ☺ ***What is wrong here?***

Supposed to be 5

2. Critical-Section Problem

- **Critical section**
  - A section of code, in which threads may change common variables, updates a table, writes a file and so on.

  - Three requirements for a solution to the critical-section problem:
    1. **Mutual Exclusion** - If a thread is executing in its critical section, then no other threads can be executing in their critical sections.
    2. **Progress** - If no thread is executing in its critical section and there exist some threads that wish to enter their critical section, then the selection of the threads 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 threads are allowed to enter their critical sections after a thread has made a request to enter its critical section and before that request is granted.

    - Assume that each thread executes at a nonzero speed.
    - No assumption concerning relative speed of the threads

3. Two-task Solutions

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

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

     public abstract void enteringCriticalSection(int turn);
     public asbtract void leavingCriticalSection(int turn);
}
```

Algorithm Factory class

Used to create two threads and to test each algorithm

```
public class AlgorithmFactory
{
   public static void main(String args[]) {
        MutualExclusion alg = new Algorithm_1();
        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);
        ...    // jobs in the critical section
        mutex.leavingCriticalSection(id);
        ...    // jobs in non-critical section
     }
   }
}
```

Algorithm_1

```
public class Algorithm_1 implements MutualExclusion
{
   private volatile int turn;  // caching in a CPU register

   public Algorithm_1() {
        turn = TURN_0;         // Initially, 0 first
   }
   public void enteringCriticalSection(int t) {
     while (turn != t)         // while the other is in service
        Thread.yield();
   }
   public void leavingCriticalSection(int t) {
        turn = 1 - t;          // yield to another
   }
}
```

Algorithm_1 - continued

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

`turn = 1 - t;`	
`...`	
`while (turn != t) => true`	

Algorithm_2

```
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;              // I want
         while(flag1 == true)       // but wait as long as
             Thread.yield();            //   the other wants
     } else {
         flag1 = true;
         while (flag0 == true)
             Thread.yield();
     }
   }
   public void leavingCriticalSection(int t) {
     if (t == 0)
       flag0 = false;          // I am done
     else
       flag1 = false;
   }
}
```

Algorithm_2 - continued

- Add more state information
  - Boolean flags to indicate thread’s interest in entering critical section
- ☺ *Mutual exclusion ???*
- ☺ *Progress requirement still not met ???*
  - ☺ *Why?*

`flag0 = true;`	
`while (flag1 == true) => false`	
`...`	`flag1 = true;`
`flag0 = true;`	
`while (flag1 == true) => true`	`while(flag0 == true) => true`

Algorithm_3

```
public class Algorithm_3 implements MutualExclusion
{
   private volatile boolean flag0;     // 0 is trying to enter
   private volatile boolean flag1;     // 1 is trying to enter
   private volatile int turn;          // whose turn?

   public Algorithm_3() {
        flag0 = false;
        flag1 = false;
        turn = TURN_0;                 // 0’s turn initially
   }
   // Continued on Next Slide
```

Algorithm_3 - enteringCriticalSection

```
public void enteringCriticalSection(int t) {
     int other = 1 - t;
     turn = other;           // You first ***
     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 – leavingingCriticalSection()

```
   public void leavingCriticalSection(int t) {
        if (t == 0)
            flag0 = false;
        else
            flag1 = false;
   }
}
```

Algorithm_3 - continued

- Combine ideas from 1 and 2.
- ☺ *Does it meet critical section requirements ???*

- ☺ *Can we extend the algorithm for several threads?*

4. Synchronization Hardware

- Many systems provide hardware support for critical section code.
- Uniprocessor systems – 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 - continued

```
    public boolean getAndSet(boolean data) {   // atomic
         boolean oldValue = this.get();
         this.set(data);
         return oldValue;
    }

    public void swap(HardwareData other) {     // atomic
         boolean temp = this.get();
         this.set(other.get());
         other.set(temp);
    }
}
```

Thread Using get-and-set Lock

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

while (true) {
   while (lock.getAndSet(true))    // true means "occupied"
        Thread.yield();
   // criticalSection();
   lock.set(false);                // release
   // 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);    // true means "occupied"
   // criticalSection();
   lock.set(false);                // release
   // nonCriticalSection();
}
```

☺ ***Any problem in the previous algorithms?***

5. Semaphore

- Synchronization tool that does not require busy waiting (***spin lock***)
- **Semaphore** *S* – integer variable having the next two operations
- 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.
  - Provides mutual exclusion.

```
Semaphore S; // initialized to 1

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

Synchronization using Semaphores Implementation - SemaphoreFactory

```
public class SemaphoreFactory
{
   public static void main(String args[]) {
         Semaphore sem = new Semaphore(1);
         Thread[] bees = new Thread[5];
         for (int i = 0; i < 5; i++)
             bees[i] = new Thread(new Worker
                 (sem, "Worker " + (new Integer(i)).toString() ));
         for (int i = 0; i < 5; i++)
             bees[i].start();
    }
}
```

Synchronization using Semaphores Implementation - Worker

```
public class Worker implements Runnable
{
   private Semaphore sem;
   private String name;
   public Worker(Semaphore sem, String name) {
        this.sem = sem;
        this.name = name;
   }
   public void run() {
        while (true) {
            sem.acquire();
            criticalSection();
            sem.release();
            nonCriticalSection();
        }
   }
}
```

Semaphore Implementation

```
public Semaphore(int v) {
   value = v;
}

public void acquire(){
   value--;
   if (value < 0) {
        // add this process to list
        block;
   }
}

public void release(){
   value++;
   if (value <= 0) {
        // remove a process P from list
        wakeup(P);
   }
}
```

Semaphore Implementation – cont.

- ☺ *Any problem?*

Semaphore Implementation – cont.

- Must guarantee that no two processes can execute acquire() and release() on the same semaphore at the same time.
- ☺ *Is it possible?* Note acquire() and release() are atomic.
- 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 presentation.

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

```
P0                    P1
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.

7. Monitors

- Wrong use of semaphore
  - `mutex.release(); ...; mutex.acquire();`
  - `mutex.acquire(); ...; mutex.acquire();`
  - `mutex.acquire(); ...; ...`
- Or dinning-philosopher's problem: dead lock situation

- A **monitor** is a high-level abstraction (abstract data type) that provides thread safety.
- Only one thread may be active within the monitor at a time, i.e., mutual exclusion within the monitor.

```
monitor monitor-name
{
  // variable declarations
  public entry p1(…) {
     …
  }
  public entry p2(…) {
     …
  }
}
```

Monitors

- Private variables within the monitor can be accessed only through procedures within the monitor.
- Concurrent access to procedures defined within the monitor is prohibited.

Condition Variables

- **condition** x, y;
- A thread that invokes x.wait is suspended until another thread invokes x.signal
- See Figure 6.24, “Schematic view of a monitor”.
- See Figure 6.25, “Monitor with condition variables”.