Basic concepts

Concurrent programming is concerned with writing programs having multiple processes that execute in parallel. A process is simply a sequential program that is being executed in the computer. Processes cooperate by the direct or indirect exchange of messages. Exchanging messages means *communicating* and *synchronizing*. Mechanisms for communicating and synchronizing must be offered by any concurrent programming language.

Communication and synchronization can be done via shared variables or via explicit message exchange. Communication using shared variables is the fastest solution on a single processor machine or on a multi-processor machine where each processor can access the same memory. Message passing is most appropriate in distributed environments (but can, of course, also be used in shared-memory environments).

Some Definitions:

a) concurrent program: two or more processes or threads that cooperate to perform a task.
b) process: A sequential program that executes a sequence of statements; can be composed of multiple threads that each execute portions of code.
c) atomic action: a computation that cannot be interrupted
d) interleaving: execution of a sequence of atomic actions, independent of the process
e) critical section: statements of a process which access shared objects
f) mutual exclusion: ensuring that critical sections of processes are not executed simultaneously

Question: What is the difference between **processes** and **threads** ?

Critical section example:

```
int tail = 0;
SomeObject queue[]= new SomeObject[QUEUESIZE];

process p1::
            SomeObject data1;
            ...
            queue[tail] = data1;
            tail = tail + 1;
            ...

process p2::
            SomeObject data1;
            ...
            queue[tail] = data1;
            tail = tail + 1;
            ...
```

What is an atomic action ?

Example:

```
int n = 0;

process P1::
      ...
         n = n + 1;
      ...

process P2::
      ...
         n ++;
      ...
```

Computer with atomic INC instruction:

Process	instruction	value of n
(initially)		0
P1	INC n	1
P2	INC n	2

Process	instruction	value of n
(initially)		0
P2	INC n	1
P1	INC n	2

Computer with ADD instruction:

Process	instruction	value of n	Reg(1)	Reg(2)
(initially)		0		
P1	LOAD REG, n	0	0	
P2	LOAD REG, n	0	0	0
P1	ADD REG, #1	0	1	0
P2	ADD REG, #1	0	1	1
P1	STORE REG, n	1	1	1
P2	STORE REG, n	1	1	1

Conclusion: you can't trust incrementation or decrementation; only assignment of base types (int, char, short, ...).

Critical Section (CS) Problem

Mutual exclusion property: at most one process at a time is executing its CS

Absence of Deadlock (livelock): if two or more processes are trying to enter their Css, then at least one will succeed

Absence of unnecessary delay: if a process is trying to enter its CS and the other processes are executing their non-CSs or have terminated, the first process is not prevented from entering its CS.

Eventual entry: a process that is attempting to enter its CS will eventually succeed.

General form of program

```
// pseudo-code of the critical section problem
public class GeneralForm {
   private static final int S = 4; // number of "sessions"
   private static final int N = 6; // the number of processes
   private static int x = 0;       // the "critical section"

   // a shorthand for threads
   private static process p( (int i = 0; i < N; i++) ) {
      for (int s = 1; s <= S; s++) {
         // non-critical section
         ...
         // entry protocol
         // critical section
         x += 3;
         // exit protocol
      }
   }

   // program entry and setting up environment
   public static void main(String [] args) {
      // normally set up threads for running
   }
}
```

Critical section: attempt 1

```
private static int turn = 1; // who is in CS ?
private static int x=0;

private static process p1 {
   for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      while ( turn != 1 )
         ;
      // critical section
      x += 3;
      // exit protocol
      turn = 2;
   }
}
```

```
private static process p2 {
   for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      while ( turn != 2 )
         ;
      // critical section
      x += 3;
      // exit protocol
      turn = 1;
   }
}
```

Critical section: attempt 2

```
private static int c1 = 0; // p1 sets to 1 in CS
private static int c2 = 0; // p2 sets to 1 in CS
private static int x = 0;
```

```
private static process p1 {
   for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      while ( c2 == 1 )
         ;
      c1 = 1;
      // critical section
      x += 3;
      // exit protocol
      c1 = 0;
   }
}
```

```
private static process p2 {
   for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      while ( c1 == 1 )
         ;
      c2 = 1;
      // critical section
      x += 3;
      // exit protocol
      c2 = 0;
   }
}
```

Critical section: attempt 3

```
private static int c1 = 0; // p1 sets to 1 in CS
private static int c2 = 0; // p2 sets to 1 in CS
private static int x=0;
```

```
private static process p1 {
   for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      c1 = 1;
      while ( c2 == 1 )
         ;
      // critical section
      x += 3;
      // exit protocol
      c1 = 0;
   }
}
```

```
private static process p2 {
   for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      c2 = 1;
      while ( c1 == 1 )
         ;
      // critical section
      x += 3;
      // exit protocol
      c2 = 0;
   }
}
```

Critical section: attempt 4

```
private static int c1 = 0; // p1 sets to 1 in CS
private static int c2 = 0; // p2 sets to 1 in CS
private static int x=0;
```

```
private static process p1 {
   for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      c1 = 1;
      while ( c2 == 1 ) {
         c1 = 0;
         c1 = 1;
      }
      // critical section
      x += 3;
      // exit protocol
      c1 = 0;
   }
}
```

```
private static process p2 {
   for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      c2 = 1;
      while ( c1 == 1 ) {
         c2 = 0;
         c2 = 1;
      }
      // critical section
      x += 3;
      // exit protocol
      c2 = 0;
   }
}
```

Critical Section – attempt 5

```
private static boolean enter1 = false;
private static boolean enter2 = false;
private static int turn = 1, x = 0;

private static process p1 {
   for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      enter1 = true;
      while (enter2) {
         if (turn == 2) {
            enter1 = false;
            while (turn == 2) ;
            enter1 = true;
         }
      }
      // critical section
      x += 3;
      // exit protocol
      enter1 = false;  turn = 2;
   }
}
```

```
private static process p2 {
   for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      enter2 = true;
      while (enter1) {
         if (turn == 1) {
            enter2 = false;
            while(turn == 1) ;
            enter2 = true;
         }
      }
      // critical section
      x += 3;
      // exit protocol
      enter2 = false; turn = 1;
   }
```