IT5002 Computer Systems and Applications
Tutorial 8
Inter-process Communications and Memory Management

Note: Synchronization is important to both multithreaded and multi-process programs. Hence, we will use the term **task** in this tutorial, i.e. do not distinguish between process and thread.

1. Consider the following two tasks, *A* and *B*, to be run concurrently and use a shared variable *x*. Assume that:
   - load and store of *x* is atomic
   - *x* is initialized to 0
   - *x* must be loaded into a register before further computations can take place.

Task A	Task B
`x++; x++;`	`x = 2*x;`

   How many **relevant** interleaving scenarios are possible when the two threads are executed? What are all possible values of *x* after both tasks have terminated? Use a stepby-step execution sequence of the above tasks to show all possible results.

2. [Semaphore] Consider three concurrently executing tasks using two semaphores S1 and S2 and a shared variable *x*. Assume S1 has been initialized to 1, while S2 has been initialized to 0. What are the possible values of the global variable *x*, initialized to 0, after all three tasks have terminated?

A	B	C

`P(S2);`
`P(S1);`
`x = x*2;`
`V(S1);`
	`P(S1);`
	`x = x * x;`
	`V(S1);`
		`P(S1);`
		`x = x + 3;`
		`V(S2);`
		`V(S1);`

*Note: P(), V() are a common alternative name for Wait() and Signal() respectively.

3. [Semaphore] In cooperating concurrent tasks, sometimes we need to ensure that all N tasks reach a certain point in code before proceeding. This specific synchronization mechanism is commonly known as a **barrier**. Example usage:

```
//some code

Barrier( N );  //The first N-1 tasks reaching this point
               // will be blocked.
               //The arrival of the Nth task will release
               //  all N tasks.

//Code here only get executed after all N processes
// reached the barrier above.
```

Use semaphores to implement a **one-time use `Barrier()`** function **without using any form of loops.** Remember to indicate the variables declarations clearly.

4. [**Low Level Implementation of CS**] Multi-core platform X does not support semaphores or mutexes. However, platform **X** supports the following atomic function:

```
bool _sync_bool_compare_and_swap (int* t, int v, int n);
```

The above function atomically compares the value at location pointed by `t` with value `v`. If equal, the function will replace the content of the location with a new value `n`, and return **`1`** (`true`), otherwise return **`0`** (`false`).

Your task is to implement function `atomic_increment` on platform **X**. Your function should always return the incremented value of referenced location `t`, and be free of race conditions. The use of busy waiting is allowed.

```
int atomic_increment( int* t ) {
   //your code here







}
```

5. Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	1327	580
3	1952	96

What are the physical addresses for the following logical addresses?

(a) (0,430)
(b) (1,10)
(c) (2,500)
(d) (3,400)

Which of these addresses, if accessed, would result in a segmentation violation?

6. Assume there is a 1,024KB segment where memory is allocated using the buddy system. Describe the configuration of the system as the following allocation requests are served:

(a) Request 240 bytes
(b) Request 120 bytes
(c) Request 60 bytes
(d) Request 130 bytes

Next, explain the configuration of the system as the blocks allocated first, third, and fourth are released.