CS2106 Operating Systems
Semester 1 2020/2021

Week of 1st March 2021
Tutorial 5 **Suggested Solutions**
Synchronization

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. [*Race Conditions*] 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.

Ou	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 step-by-step execution sequence of the above tasks to show all possible results.

ANS:

Load and store instructions to variable *x* are the only instructions whose interleaving can affect the value of *x*. The sequences of such instructions for both processes are shown below:

Task A	Task B
A1: load x A2: store x A4: load x A4: store x	B1: load x B2: store x

Interleaving scenarios can be conveniently represented as a sequence (e.g., A1-A2-A3-A4-B1-B2). How many possible interleaving patterns are there?

Lets start with A1-A2-A3-A4 (the order of instructions cannot change) and see in how many ways we can insert B1 and B2 so that B1 is before B2. We can insert B1 into one of 5 slots (before each A instruction and after the last one). In each of these insertions we have created one new slot for B2, which can now be inserted into one of 6 slots – 30 possible sequences in total. However, in only half of these B1 is before B2, so 5x6/2=15. All of them are shown below, together with the outcomes:

B1-A1-A2-A3-A4-B2 → $x = 0$
B1-A1-A2-A3-B2-A4 → $x = 2$
B1-A1-A2-B2-A3-A4 → $x = 1$
B1-A1- B2-A2-A3-A4 → $x = 2$
B1-B2-A1-A2-A3-A4 → $x = 2$

A1-B1-A2-A3-A4-B2 → $x = 0$
A1-B1-A2-A3-B2-A4 → $x = 2$
A1-B1-A2-B2-A3-A4 → $x = 1$
A1-B1-B2-A2-A3-A4 → $x = 2$

A1-A2-B1-A3-A4-B2 → $x = 2$
A1-A2-B1-A3-B2-A4 → $x = 2$
A1-A2-B1-B2-A3-A4 → $x = 3$

A1-A2-A3-B1-A4-B2 → $x = 2$
A1-A2-A3-B1-B2-A4 → $x = 2$

A1-A2-A3-A4-B1-B2 → $x = 4$

There are five possible outcomes: 0, 1, 2, 3, 4. Note that the outcome would be non-deterministic even if each statement was executed atomically, because x=2*x and x++ are not commutative operations.

2. [Critical Section] Can disabling interrupts avoid race conditions? If yes, would disabling interrupts be a good way of avoiding race conditions? Explain.

ANS:

Yes, disabling interrupts and enabling them back is equivalent to acquiring a universal lock and releasing it, respectively. Without interrupts, there will be no quantum-based process switching (because even timer interrupts cannot happen); hence only one process is running. However:

- This will not work in a multi-core/multi-processor environment since another process may enter the critical section while running on a different core.
- User code may not have the privileges needed to disable timer interrupts.
- Disabling timer interrupts means that many scheduling algorithms will not work properly.
- Disabling non-timer interrupts means that high-priority interrupts that may not even share any data with the critical section may be missed.
- Many important wakeup signals are provided by interrupt service routines and these would be missed by the running process. A process can easily block on a semaphore and stay blocked indefinitely, because there is nobody to send a wakeup signal.
- If a program disables interrupts and hangs, the entire system will no longer work since it cannot switch tasks and perform anything else.

Therefore disabling interrupts is not a good choice for the purpose of locking.

3. [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() resepectively.

ANS:

Semaphore S1 is initialized to 1, which means it acts like a mutex lock and ensures mutually exclusive access to variable *x* for all three processes.

Because semaphore S2 has been initialized to 0, process A can execute only after it receives signal V(S2) from process C, which is after variable *x* was accessed by process C. V(S2) is in critical section guarded by S1 → process A will certainly not proceed before C sends signal V(S1).

How many possible executions do we have? The number of executions is equal to the number of permutations of A, B, and C in which A happens after C. Which is 3!/2=3:

B-C-A → x = 6
C-A-B → x = 36
C-B-A → x = 18
Possible outcomes are 6, 18, and 36.

4. [Semaphore] In cooperating concurrent tasks, sometimes we need to ensure that all N tasks reached 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.

ANS:

```
int arrived = 0; //shared variable
Semaphore mutex = 1; //binary semaphore to provide mutual exclusion
Semaphore waitQ = 0; //for N-1 process to blocks


Barrier( N ) {
      wait( mutex );
      arrived ++;
      signal( mutex );

      if (arrived == N )
            signal( waitQ )

      wait( waitQ );
      signal( waitQ );
}
```

[Note to instructors] Point out that this is not a **reusable barrier**, as the **waitQ** may have undefined value afterwards, e.g. when task interleave just before the "if (arrived == N)" ➔ multiple tasks can execute the first "signal(waitQ)" resulting in >0 waitQ at the end. Note that this does not affect correctness.