CS3331 Concurrent Computing Exam 1 Solutions
Spring 2017

1. **Basic Concepts**

   (a) **[10 points]** Explain what are CPU modes. Explain their uses. How does the CPU know what mode it is in? **There are three questions.**

   Answer: The following has the answers.

   - CPU modes are operating modes of the CPU. Modern CPUs have two execution modes: the user mode and the supervisor (or system, kernel, privileged) mode, controlled by a mode bit.
   - The OS runs in the supervisor mode and all user programs run in the user mode. Some instructions that may do harm to the OS (*e.g.*, I/O and CPU mode change) are privileged instructions. Privileged instructions, for most cases, can only be used in the supervisor model. When execution switches to the OS (*resp.*, a user program), execution mode is changed to the supervisor (*resp.*, user) mode.
   - A mode bit can be set by the operating system, indicating the current CPU mode.

   See page 5 of `02-Hardware-OS.pdf`. ∎

   (b) **[10 points]** What is an atomic instruction? What would happen if multiple CPUs/cores execute their atomic instructions? **Make sure you will explain atomic instructions fully. Otherwise, you may receive a lower or very low score.**

   Answer: An atomic instruction is a machine instruction that executes as one *uninterruptible* unit without interleaving and cannot be split by other instructions. When an atomic instruction is recognized by the CPU, we have the following:

   - All other instructions being executed in various stages by the CPUs are suspended (and perhaps re-issued later) until this atomic instruction finishes.
   - No interrupts can occur.

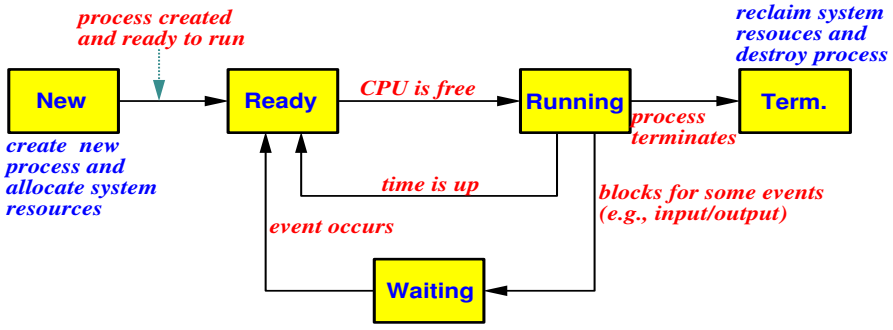
   If two such instructions are issued at the same time on different CPUs/cores, they will be executed sequentially in an arbitrary order determined by the hardware.

   See pp. 12–13 of `02-Hardware-OS.pdf`. ∎

2. **Processes**

   (a) **[10 points]** Draw the state diagram of a process from its creation to termination, including all transitions. Make sure you will elaborate **every state** and **every transition** in the diagram.

   Answer: The following state diagram is taken from my class note.



   There are five states: **new**, **ready**, **running**, **waiting**, and **terminated**.

   - **New**: The process is being created.
   - **Ready**: The process has everything but the CPU, and is waiting to be assigned to a processor.

- **Running**: The process is executing on a CPU.
- **Waiting**: The process is waiting for some event to occur (*e.g.*, I/O completion or some resource).
- **Terminated**: The process has finished execution.

The transitions between states are as follows:

- **New→Ready**: The process has been created and is ready to run.
- **Ready→Running**: The process is selected by the CPU scheduler and runs on a CPU/core.
- **Running→Ready**: An interrupt has occurred forcing the process to wait for the CPU.
- **Running→Waiting**: The process must wait for an event (*e.g.*, I/O completion or a resource).
- **Waiting→Ready**: The event the process is waiting has occurred, and the process is now ready for execution.
- **Running→Terminated**: The process exits.

See pp. 5–6 of `03-Process.pdf`. ∎

(b) **[10 points]** What is a *context*? Provide a detail description of *all* activities of a *context switch*.

Answer: A process needs some system resources (*e.g.*, memory and files) to run properly. These system resources and other information of a process include process ID, process state, registers, memory areas (for instructions, local and global variables, stack and so on), various tables (*e.g.*,, PCB), a program counter to indicate the next instruction to be executed, etc. They form the *environment* or *context* of a process. The steps of switching process *A* to process *B* are as follows:

- The operating system suspends *A*'s execution. A CPU mode switch may be needed.
- Transfer the control to the CPU scheduler.
- Save *A*'s context to its PCB and other tables.
- Load *B*'s context to register, etc. from *B*'s PCB.
- Resume *B*'s execution of the instruction at *B*'s program counter. A CPU mode switch may be needed.

See pp. 10–11 of `03-Process.pdf`. ∎

3. **Threads**

(a) **[10 points]** Why is handling threads cheaper than handling processes? **Provide an accurate account of your findings. Otherwise, you risk a lower grade.**

Answer: There are two major points.

- **Resource Consumption and Sharing:** A thread only requires a thread ID, a program counter, a register set and a stack, and shares with other peer threads in the same process its code section, data section, and other OS resources (*e.g.*, files and signals). Fewer resource consumption means less allocation overhead.
- **Faster in Handling Context Switching:** Since a thread has fewer items than a process does, it is faster to perform thread-based context switching as fewer data items have to be saved and loaded. Moreover, the creation, destroy and joining of threads would also be cheaper.

Therefore, handling threads is cheaper than handling processes.

See pp. 2–4 of `04-Thread.pdf`. ∎

4. **Synchronization**

(a) **[10 points]** Define the meaning of a *race condition*? Answer the question first and use execution sequences with a clear and convincing argument to illustrate your answer. **You must explain step-by-step why your example causes a race condition.**

Answer: A *race condition* is a situation in which *more than one* processes or threads manipulate a shared resource *concurrently*, and the result depends on *the order of execution*.

The following is a simple counter updating example discussed in class. The value of `count` may be 9, 10 or 11, depending on the order of execution of the **machine instructions** of `count++` and `count--`.

```
int         count = 10;  // shared variable

Process 1                Process 2

count++;                 count--;
```

The following execution sequence shows a race condition. Two processes run concurrently (condition 1). Both processes access the shared variable `count` concurrently (condition 2) because `count` is accessed in an interleaved way. Finally, the computation result depends on the order of execution of the `SAVE` instructions (condition 3). The execution sequence below shows the result being 9; however, switching the two `SAVE` instructions yields 11. Since all conditions are met, we have a race condition. **Note that you have to provide TWO execution sequences, one for each possible result, to justify the existence of a race condition.**

Thread_1	Thread_2	*Comment*
do somthing	do somthing	`count = 10` initially
`LOAD count`		`Thread_1` executes `count++`
`ADD #1`		
	`LOAD count`	`Thread_2` executes `count--`
	`SUB #1`	
`SAVE count`		`count` is 11 in memory
	`SAVE count`	Now, `count` is 9 in memory

Stating that "`count++` followed by `count--`" or "`count--` followed by `count++`", even using machine instructions, produces different results and hence a race condition is **incomplete**, because the two processes do not access the shared variable `count` concurrently. Note that the use of higher-level language statement interleaved execution may not reveal the key concept of "sharing" as discussed in class. Therefore, use instruction level interleaved instead.

See pp. 5–12 of `05-Sync-Basics.pdf`. ■

(b) **[10 points]** A good solution to the critical section problem must satisfy three conditions: *mutual exclusion*, *progress* and *bounded waiting*. Both progress and bounded waiting involve some form of waiting. Explain and differentiate the waiting in progress and bounded waiting. **You should provide a clear answer with a convincing argument. Otherwise, you will receive no credit.**

Answer: A process in the enter protocol has two forms of "waiting": (a) waiting for a decision to be made to determine who can enter, and (b) waiting to enter if it is not chosen. The *progress* condition states that a decision to determine who can enter must be made in finite time, while the *bounded waiting* condition states that a process waits for a bounded number of times before it can enter. Therefore, the major difference is that the "waiting" in *progress* is waiting for a decision to be made and the "waiting" in *bounded waiting* is waiting to enter a critical section.

See pp. 18-20 of `05-Sync-Basics.pdf`. ■

5. **Problem Solving:**

(a) **[15 points]** Consider the following two processes, *A* and *B*, to be run concurrently using a shared memory for the `int` variable `x`.

```
Process A                    Process B
---------                    ---------
for (i = 1; i <= 2; i++)     x = 2*x;
   x++;
```

Assume that `x` is initialized to 0, and `x` must be loaded into a register before further computations can take place. What are **all possible** values of `x` after both processes have terminated. **You must use clear step-by-step execution sequences of the above processes with a convincing argument. Any vague and unconvincing argument receives no points.**

Answer: Obviously, the answer must be in the range of 0 and 4. It is non-negative, because the initial value is 0 and no subtraction is used. It cannot be larger than 4, because the two `x++` statements and `x = 2*x` together can at most double the value of `x` twice.

The easiest answers are 2, 3 and 4 if `x = 2*x` executes before, between and after the two `x++` statements, respectively. The following shows the possible execution sequences.

`x = 2*x` **is before both** `x++`		
Process 1	Process 2	`x` in memory
	`x = 2*x`	0
`x++`		1
`x++`		2

`x = 2*x` **is between the two** `x++`		
Process 1	Process 2	`x` in memory
`x++`		1
	`x = 2*x`	2
`x++`		3

`x = 2*x` **is after both** `x++`		
Process 1	Process 2	`x` in memory
`x++`		1
`x++`		2
	`x = 2*x`	4

The situation is a bit more complex with instruction interleaving. Process B's `x = 2*x` may be translated to the following machine instructions:

```
LOAD x
MUL  #2
SAVE x
```

Because the `LOAD` retrieves the value of `x`, and the `SAVE` may change the current value of `x`, the results depend on the positions of `LOAD` and `SAVE`. The following shows the result being 0. In this case, `LOAD` loads 0 *before* both `x++` statements, and the result 0 is saved *after* both `x++` statements.

Process 1	Process 2	`x` in memory	Comments
	`LOAD x`	0	Load $x = 2$ into register
	`MUL #2`	0	Process 2's register is 0
`x++`		1	Process 1 adds 1 to x
`x++`		2	Process 1 adds 1 to x
	`SAVE x`	0	Process 2 saves 0 to x

If the `SAVE` executes between the two `x++` statements, the result is 1.

Process 1	Process 2	`x` in memory	Comments
	`LOAD x`	0	Load $x = 2$ into register
	`MUL #2`	0	Process 2's register is 0
`x++`		1	Process 1 adds 1 to x
	`SAVE x`	0	Process 2 saves 0 to x
`x++`		1	Process 1 adds 1 to x

You may try other instruction interleaving possibilities and the answers should still be in the range of 0 and 4. ∎

(b) **[15 points]** Consider the following solution to the mutual exclusion problem for two processes P_0 and P_1. A process can be making a request `REQUESTING`, executing in the critical section `IN_CS`, or having nothing to do with the critical section `OUT_CS`. This status information, which is represented by an `int`, is saved in `flag[i]` of process P_i. Moreover, variable `turn` is initialized elsewhere to be `0` or `1`. Note that `flag[]` and `turn` are global variables shared by both P_0 and P_1.

```
int   flag[2];   // global flags, initialized to OUT_CS
int   turn;      // global turn variable, initialized to 0 or 1

Process i (i = 0 or 1)

// Enter Protocol
repeat                                      // repeat the following
   flag[i] = REQUESTING;                    // making a request to enter
   while (turn != i && flag[j] != OUT_CS)   // as long as it is not my turn and
      ;                                     //    the other is not out, wait
   flag[i] = IN_CS;                         // OK, I am in (well, maybe); but,
until flag[j] != IN_CS;                     //    must wait until the other is not in
turn = i;                                   // the other is out and it is my turn!

// critical section

// Exit Protocol
turn = j;                                   // yield the CS to the other
flag[i] = OUT_CS;                           // I am out of the CS
```

Prove rigorously that this solution satisfies the mutual exclusion condition. *You will receive* **zero** *point if* **(1)** *you prove by example, or* **(2)** *your proof is vague and/or not convincing.*

Answer: A process that enters its critical section must first set `flag[i] = IN_CS` and then see `flag[j] != IN_CS` being true at the end of the `repeat-until` loop. Therefore, if process P_0 is in the critical section, it had executed `flag[0] = IN_CS` followed by seeing `flag[1] != IN_CS`. By the same reason, if process P_1 is in the critical section, it had executed `flag[1] = IN_CS` followed by seeing `flag[0] != IN_CS`. Consequently, if P_0 and P_1 are both in the critical section, we have `flag[0] = IN_CS` and `flag[1] != IN_CS` (from P_0's point of view) and `flag[1] = IN_CS` and `flag[0] != IN_CS` (from P_1's point of view). As a result, `flag[0]` and `flag[1]` are equal to `IN_CS` and not equal to `IN_CS` at the same time. This is impossible, and the mutual exclusion condition holds.

Note that the variable `turn` does not play a role here. Right after P_0 and P_1 pass their `repeat-until` loop, they will store some value to `turn`. At this point, because P_0 and P_1 will be in their critical sections without any obstruction, and the value in `turn` does not matter.

See page 7 of `06-Sync-Soft-Hardware.pdf`. This is the same technique as the one we used to show that Attempt II satisfies the mutual exclusion condition. ∎