CS3331 Concurrent Computing Solution 1
Spring 2014

1. **Basic Concepts**

   (a) **[10 points]** Explain *interrupts* and *traps*, and provide a detailed account of the procedure that an operating system handles an interrupt.

   Answer: An *interrupt* is an event that requires the attention of the operating system. These events include the completion of an I/O, a key press, the alarm clock going off, division by zero, accessing a memory area that does not belong to the running program, and so on. Interrupts may be generated by hardware or software. A *trap* is an interrupt generated by software (*e.g.*, division by 0 and system call).

   When an interrupt occurs, the following steps will take place to handle the interrupt:

   - The executing program is suspended and control is transferred to the operating system. Mode switch may be needed.
   - A general routine in the operating system examines the received interrupt and calls the interrupt-specific handler.
   - After the interrupt is processed, a context switch transfers control back to a suspended process. Of course, mode switch may be needed.

   See pp. 7-8 `02-Hardware-OS.pdf`. ∎

   (b) **[10 points]** What is an atomic instruction? What would happen if multiple CPUs/cores execute their atomic instructions?

   Answer: An atomic instruction is a machine instruction that executes as one uninterruptable unit. More precisely, when such an instruction runs, all other instructions being executed in various stages by the CPUs will be stopped (and perhaps re-issued later) until this instruction finishes. If two such instructions are issued at the same time, even though on different CPUs/cores, they will be executed sequentially in an arbitrary order determined by the hardware.

   See page 6 of `02-Hardware-OS.pdf`. ∎

2. **Processes**

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

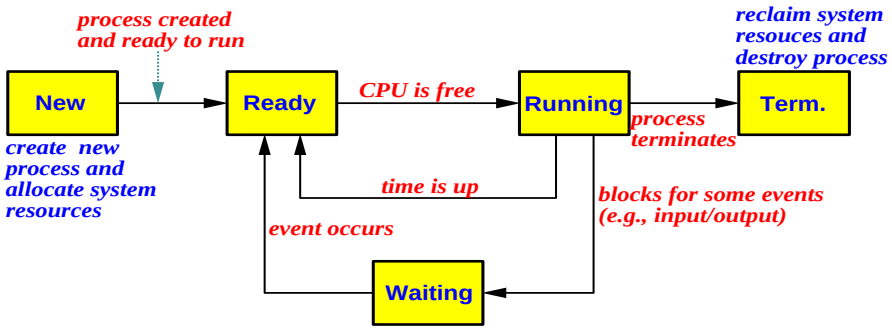
   Answer: A process needs some system resources (*e.g.*, memory and files) to run properly. These system resources and other information of a process include process ID, process state, registers, memory areas (for instructions, local and global variables, stack and so on), various tables (*i.e.*, process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

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

See page 10 and page 11 of `03-Process.pdf`. ∎

(b) **[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 page 5 and page 6 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 threads in the same process its code section, data section, and other OS resources (*e.g.*, files and signals). Fewer resource consumption means fewer 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.

Therefore, handling threads is cheaper than handling processes.

See page 2 to page 4 of `04-Thread.pdf`. ∎

4. **Synchronization**

(a) **[10 points]** Define the meaning of a *race condition*? Answer the question first and use an execution sequence 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 access 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` at the same time (condition 2). 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.

`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++`" produces different results and hence a race condition is at least **incomplete**, because the two processes do not access the shared variable `count` concurrently. Note that the use of higher-level language statement interleaving may not reveal the key concept of "sharing" as discussed in class. Therefore, use instruction level interleaving instead.

See page 5 to page 9 of `05-Sync-Basics.pdf`. ∎

(b) **[10 points]** A computer system has two CPUs that share the same memory. All processes are stored in the shared memory but can be run on either CPU. To gain efficiency, the designers chose the following CPU scheduling policy:

- There is only one ready queue and is stored in the shared memory.
- Each CPU has its own CPU scheduler.

- When a CPU is free, the scheduler of that CPU picks up the first process in the ready queue to run on the same CPU.

Do you think this policy works well? State your claim first and justify your claim step-by-step with an execution sequence. *You will receive* **no** *credit if you only provide an answer* **without** *a convincing argument or if your answer is* **vague**.

Answer: This policy does not work well because a race condition may occur. The ready queue, which is stored in the shared memory, is a shared resource (condition 1) that can be read and modified by both CPUs at the same time (condition 2). If both CPUs become free and access the ready queue, it is possible that they pick the first process in the queue and let it run. Consequently, two copies of this process will run, one on each CPUs. On the other hand, if the CPUs access the ready queue at different time, this policy would work properly (condition 3). Therefore, we have a race condition. ∎

5. **Problem Solving:**

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

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

Assume that load and store of `x` is atomic, `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. Use a step-by-step execution sequence of the above processes to show all possible results. **You must provide a clear step-by-step execution of the above algorithm 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 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
```

The `LOAD` retrieves the value of `x`, and the `SAVE` may change the current value of `x`. Therefore, 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 := x + 1`		1	Process 1 adds 1 to x
`x := x + 1`		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 := x + 1`		1	Process 1 adds 1 to x
	`SAVE x`	0	Process 2 saves 0 to x
`x := x + 1`		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_1 and P_2. This solution uses two global `int` variables, `x` and `y`. Both `x` and `y` are initialized to `0`.

```
int x = 0, y = 0;
```

Process 1

```
START:
     x = 1;
     if (y != 0) {
          repeat until (y == 0);
          goto START;
     }
     y = 1;
     if (x != 1) {
          y = 0;
          repeat until (x == 0);
          goto START;
     }
// critical section
     x = y = 0;
```

Process 2

```
START:                                  // All start from here
     x = 2;                             // set my ID to x
     if (y != 0) {                      // if y is non-zero
          repeat until (y == 0);        //    wait until y = 0
          goto START;                   //    then try again
     }                                  // second section
     y = 1;                             // set y to 1
     if (x != 2) {                      // if x is not my ID
          y = 0;                        //    set y to 0
          repeat until (x == 0);        //    wait until x = 0
          goto START;                   //    then try again
     }
// critical section
     x = y = 0;                         // set x and y to 0
```

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 unconvincing.*

Answer: We shall prove the mutual exclusion property by contradiction. Consider process P_1 first. If P_1 is in its critical section, its execution must have passed the first `if` statement, set `y` to `1`, and seen `x != 1` being false (*i.e.*, `x = 1` being true). Therefore, if P_1 is in its critical section, `x` and `y` must both be `1`. By the same reason, if P_2 is in its critical section, `x` and `y` must be `2` and `1`, respectively. Now, if P_1 and P_2 are **both** in their critical sections, `x` must be both `1` and `2`. This is impossible because a variable can only hold one value. As a result, the assumption that P_1 and P_2 are both in their critical sections cannot hold, and, the mutual exclusion condition is satisfied. ∎