CS3331 Concurrent Computing Exam 1 Solutions
Spring 2016

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?

   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 instruction finishes.
   - No interrupts can occur.

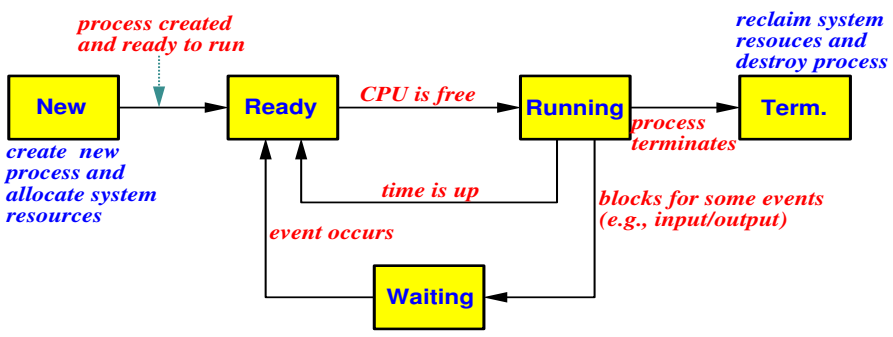
   If two such instructions are issued at the same time on different CPUs/cores, they will be executed sequentially in an arbitrary order determined by the hardware.

   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`. ∎

3. **Threads**

(a) **[10 points]** Explain the one-to-one, many-to-one, and many-to-many thread models. **Make sure you explain each model clearly.**

Answer: The three thread models are defined as follows:

- **One-to-One Model:** Each process runs one thread and is associated with one kernel thread. There is no thread support at all, and the CPU scheduler dispatches processes or kernel threads. This is the traditional Unix system. On the other hand, in some systems each user thread is associated with a kernel thread, and, as a result, it is also an one-to-one model.
- **Many-to-One Model:** A process may run multiple (user) threads, and is associated with only one kernel thread. Therefore, the CPU scheduler dispatches kernel threads and does not know the existence of user threads. If the containing process (or the associated kernel thread) is blocked, all of its user threads are blocked.
- **Many-to-Many Model:** A process may have multiple user threads and is associated with multiple kernel threads. Each of these associated kernel threads can be attached to a user thread dynamically by the scheduler in the thread library. In this way, unless all kernel threads associated with the process are blocked, at least one user thread can run.

See pp. 8–12 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 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` 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++`" 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 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 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 execution sequences. *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 may work properly (condition 3). Therefore, we have a race condition. ■

5. **Problem Solving:**

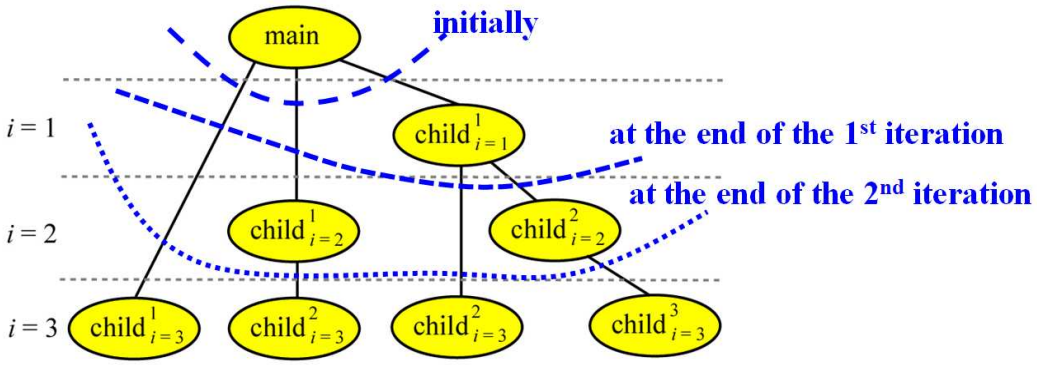
(a) **[10 points]** Consider the following program segment. Suppose all `fork()` calls are successful. Answer the following questions: **(1)** Draw a diagram showing the parent-child relationship of *all* involved processes, the main program included, and provide an explanation how this relationship is obtained. **Vague and not convincing arguments receive zero point. (2)** This program segment uses `n = 3`. How many processes will be created if `n` is set to a positive integer k? You don't need to draw a diagram; but, you still have to provide a justification for your answer. **Vague and not convincing argument receive zero point.**

```
int i, n = 3;

for (i = 1; i <= n; i++)
     fork();
```

Answer: The following has the answer to each question:

Question (1): Keep in mind that each `fork()` creates a child process and the child process receives a separate address space that is identical to of that of the parent. As a result, after a child process is created, the value of `i` of this child is the same as that of the parent. After the `fork()` call, the parent and its child run concurrently. Initially, only the parent process runs. See the diagram below.



The creation steps are as follows:

- **i = 1**: the `main` creates a child, $\text{child}^1_{i=1}$, where the superscript 1 indicates the "generation" and the subscript has the value of i. Now we have two processes running, `main` and $\text{child}^1_{i=1}$. Both of them go back to the beginning of the `for` loop and advance to $i = 2$.
- **i = 2**: Both `main` and $\text{child}^1_{i=1}$ reach the `fork()` call and both create a child. The `main` creates a child $\text{child}^1_{i=2}$, which is still the first generation with $i = 2$ as shown in the diagram. Process $\text{child}^1_{i=1}$ also executes `fork()` to creates a child, $\text{child}^2_{i=2}$, which is the second generation in the hierarchy with $i = 2$. Now, we have four processes running, `main`, $\text{child}^1_{i=1}$ (`main`'s first child created when $i = 1$), $\text{child}^1_{i=2}$ (`main`'s second child created when $i = 2$), and $\text{child}^2_{i=2}$ ($\text{child}^1_{i=1}$'s child created when $i = 2$).
- **i = 3**: Each of the above mentioned four processes executes the `fork()` call, and creates one child process. As a result, there are eight processes in total after the `for` loop completes as shown in the diagram.

Question (2): Since each process creates one and only one child process in each iteration, the number of processes is doubled. We have the `main` initially. At the end of $i = 1$, we have $2 = 2 \times 1 = 2^1$ processes; at the end of $i = 2$, we have $4 = 2^1 \times 2 = 2^2$ processes; and at the end of $i = 3$, we have $8 = 2^2 \times 2 = 2^3$ processes. Continuing this observation, at the end of $i = k$, we should have 2^k processes.

This observation can easily be proved with mathematical induction.

- **Base Case:** If $k = 0$, we have `main` only and hence $2^0 = 1$ process.
- **Induction Step:** Assume that the proposition holds for $k-1$. That is, we assume that at the end of $i = k-1$ there are 2^{k-1} processes. Since each process only creates one child process in each iteration, when $i = k$ each of the 2^{k-1} processes creates one child process, making the total number of processes $2 \times 2^{k-1} = 2^k$. Therefore, at the end of $i = k$ we have 2^k processes.

From the above induction proof, the number of processes at the end of iteration k is 2^k. ∎

(b) **[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
---------                ---------
x += 2;                  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. 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 6. It is non-negative, because the initial value is 0 and no subtraction is used. It cannot be larger than 6, because process *A* can at best increases `x` to `3` for process *B* to double it with `x = 2*x`.

With statement level interleaved execution, we are able to recognize three easy cases as follows. The possible values are 3, 5 and 6.

CASE 1: `x = 2*x` **is before the first statements of process** *A*

`x = 2*x` **is before both** `x++`		
Process 1	Process 2	`x` in memory
	`x = 2*x`	0
`x += 2`		2
`x++`		3

CASE 2: `x = 2*x` **is between the two statements of process** *A*

`x = 2*x` **is between the two** `x++`		
Process 1	Process 2	`x` in memory
`x += 2`		2
	`x = 2*x`	4
`x++`		5

CASE 3: `x = 2*x` **is after the second statement of process** *A*

`x = 2*x` **is after both** `x++`		
Process 1	Process 2	`x` in memory
`x +=2`		2
`x++`		3
	`x = 2*x`	6

Next, we look at possible instruction level interleaved execution. The statements of process *A* and the machine instructions of process *B* are shown below:

Process *A*	Process *B*
`x += 2`	`LOAD x`
`x++`	`MUL #2`
	`SAVE x`

The final results depend on the order of the `SAVE` instruction in `x++` and in `x = 2*x` and also depend on when the `LOAD` instruction loads. There are three possible cases.

CASE 4: `LOAD` loads *before* and `SAVE` saves *after* process *A* (Answer = `0`)

Process 1	Process 2	`x` in memory	Comments
	`LOAD x`	0	Load $x = 0$ into register
	`MUL #2`	0	Process 2's register is 0
`x += 2`		2	Process 1 adds 2 to x
`x ++`		3	Process 1 adds 1 to x
	`SAVE x`	0	Process 2 saves 0 to x

CASE 5: `SAVE` saves between the statements of process *A* (Answer = `1`)

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

CASE 6: `LOAD` loads between statements and `SAVE` saves at the end (Answer = `4`)

Process 1	Process 2	`x` in memory	Comments
`x += 2`		2	Process 1 adds 2 to x
	`LOAD x`	2	Load $x = 2$ into register
	`MUL #2`	2	Process 2's register is 4
`x++`		3	Process 1 adds 1 to x
	`SAVE x`	4	Process 2 saves 4 to x

Therefore, the possible answers are 0, 1, 3, 4, 5 and 6.

Note that 2 cannot occur. Because `x` is 0 initially, the results of `x += 2` and `x = 2*x` are always even integers. It is obvious that `x = 2*x` always produces even integers. Regardless of the execution order of `x += 2` and `x = 2*x`, even with instruction level interleaved execution, `x += 2` always produces an even integer. Refer to Cases 1, 2, 4 and 5 for the details. Then, the execution of `x++` just adds 1 to the result, and the final value is an odd integer, which cannot be 2. ∎

(c) **[15 points]** Consider the following solution to the mutual exclusion problem for two processes P_0 and P_1. In the following, $(a,b) \geq (c,d)$ holds if $(a > c)$ or $(a = c$ and $b \geq d)$. Note that $\max(a,b)$ is the maximum function that returns the larger of a and b. For example, $(3,5) > (2,7)$ and $(4,5) > (4,2)$ both hold, because $3 > 2$ for the former and $4 = 4$ and $5 > 2$ for the latter.

```
Bool  flag[2];   // initially FALSE
int   num[2];    // initially 0

Process 0                                Process 1
---------                                ---------
flag[0] = TRUE;                          flag[1] = TRUE;
num[0]  = 1+max(num[0], num[1]);         num[1]  = 1+max(num[0], num[1]);
flag[0] = FALSE;                         flag[1] = FALSE;
repeat                                   repeat
  until (flag[1] == FALSE);                until (flag[0] == FALSE);
repeat                                   repeat
  until (num[1] == 0 ||                    until (num[0] == 0 ||
         (num[1],1) >= (num[0],0));               (num[0],0) >= (num[1],1));
                            // critical section
num[0]  = 0;                             num[1] = 0;
```

Prove rigorously that this solution satisfies the mutual exclusion condition. *You will receive* **zero** *point if* **(1)** *you prove by example,* **(2)** *your proof is vague and/or unconvincing*, or **(3)** you enumerate a few possible execution sequences.

Answer: We shall prove the mutual exclusion condition by contradiction. Suppose both processes P_0 and P_1 are in the critical section. They must have executed the statement `1+max(num[0],num[1])` earlier, and `num[0]` and `num[1]` become positive. Now, if P_0 is in its critical section, the condition of the second `repeat-until` must be true. Since `num[1]` is positive, this means `(num[1],1) >= (num[0],0)` must be true. By the same reason, if P_1 is in the critical section, `(num[0],0) >= (num[1],1)` must be true. Because P_0 and P_1 are *both* in the critical section, `(num[1],1) >= (num[0],0)` and `(num[0],0) >= (num[1],1)` are true. Consequently, we have

```
(num[1],1) >= (num[0],0) >= (num[1],1)
```

Intuitively, it means `(num[1],1) = (num[0],0)`. Therefore, we have `num[0] = num[1]` and `0 = 1`. The latter (*i.e.*, `0 = 1`) is absurd and yields a contradiction. As a result, P_0 and P_1 cannot be in the critical section at the same time. ∎

A Simple Proof

Recall that $(a,b) \geq (c,d)$ holds if $(a > c)$ or $(a = c$ and $b \geq d)$. We shall prove if $(a,b) \geq (c,d) \geq (a,b)$, then $a = c$ and $b = d$ must both be true (*i.e.*, $(a,b) = (c,d)$). A simple enumeration is sufficient to prove the desired result. In the table below, the second column shows the conditions for $(a,b) \geq (c,d)$, the third column shows the conditions for $(c,d) \geq (a,b)$, and the fourth column has the comments.

Case	$(a,b) \geq (c,d)$	$(c,d) \geq (a,b)$	*Comments*
1	$a > c$	$c > a$	This is impossible
2		$c = a$ and $d \geq b$	This is impossible
3	$a = c$ and $b \geq d$	$c > a$	This is impossible
4		$c = a$ and $d \geq b$	This implies $b = d$

As you can see from this table, there are four combinations: **(1)** $a > c$ and $c > a$, which is impossible; **(2)** $a > c$ and ($c = a$ and $d \geq b$); which is also impossible due to $a > c$ and $c = a$, **(3)** ($a = c$ and $b \geq d$) and $c > a$, which is impossible for the same reason as in **(2)**; and **(4)** ($a = c$ and $b \geq d$) and ($c = a$ and $d \geq b$), which implies $a = c$ and $b = d$ due to $b \geq d$ and $d \geq b$. Hence, we conclude that $(a,b) \geq (c,d) \geq (a,b)$ implies $a = c$ and $b = d$. ∎