CS3331 Concurrent Computing Solution 1
Spring 2015

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 *uninterruptible* unit, where " uninterruptible" means that 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 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*.

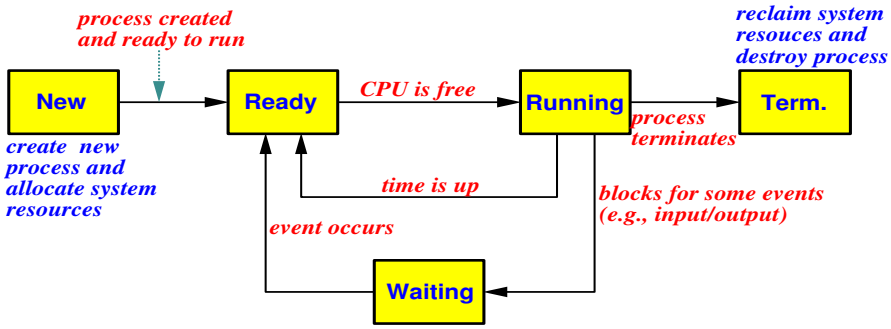
   Answer: A process needs some system resources (*e.g.*, memory and files) to run properly. These system resources and other information of a process include process ID, process state, registers, memory areas (for instructions, local and global variables, stack and so on), various tables (*e.g.,*, PCB), 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 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]** 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 page 8 to page 12 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` 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 showing difference results 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 page 5 to page 10 of `05-Sync-Basics.pdf`. ∎

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*

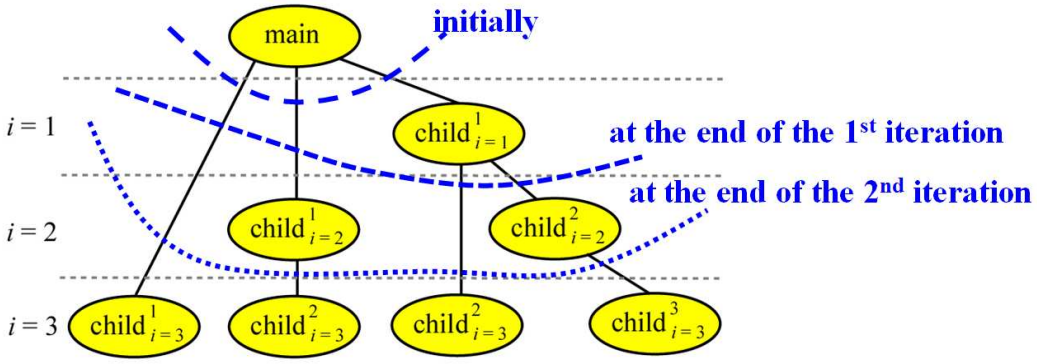
involved processes, the main program included, and provide an explanation how this relationship is obtained. **Vague and not convincing arguments receive zero point. (2)** This program segment uses `n = 3`. How many processes will be created if `n` is set to a positive integer k? You don't need to draw 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:

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` also 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. 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
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 8 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. ∎

Further Notes: If the above proof is good enough, what is the purpose of this note? Its purpose is to help you think deeper and perhaps prove mutual exclusion directly rather than going for a proof-by-contradiction.

Consider P_i's loop `while (turn != i && flag[j] != OUT_CS)`. This implies that P_i waits as long as it is not P_i's turn **and** P_j is not out of the critical section. This condition fails if `turn == i` **or** `flag[i] == OUT_CS`. Because `turn` can only hold one value, either `turn == 0` for P_0 or `turn == 1` for P_1 holds but not both. Consequently, only one process can pass its `while` loop and set its `flag[ ]` to `IN_CS`. Note that setting `flag[i]` to `IN_CS` does not imply that P_i can enter the critical section, because there is one more test in the `until` part (*i.e.*, `flag[j] != IN_CS`). Therefore, P_i sets `flag[i]` to `IN_CS` to make a claim to enter. If P_j is in the critical section or has made a claim to enter, P_i waits by returning to the top of the `repeat-until` loop and try again!

Because each process can have three possible states: `REQ`, `IN` and `OUT`, all possible state combinations of P_0 and P_1 are `REQ-REQ`, `REQ-IN`, `REQ-OUT`, `IN-IN`, `IN-OUT`, and `OUT-OUT`. Note that only six are listed rather than nine because the code is symmetric, and switching the roles of P_0 and P_1 would get one of the listed six (*i.e.*, `IN-OUT` is equivalent to `OUT-IN`).

Any state with `OUT` will not affect our discussion because **(1)** the process with `OUT` is not interested in entering the critical section, and because **(2)** the process with `OUT` will eventually become `REQ` which makes the states to be `REQ-REQ`, `REQ-IN` and `IN-IN`. Therefore, we actually have only three cases to consider: `REQ-REQ`, `REQ-IN` and `IN-IN`.

- **Case**: `REQ-REQ`
  If both P_0 and P_1 are in the `REQ` state, they must be executing the `while` loop. As a result, only one of them can pass through due to the value of `turn`. If P_0 has its turn, P_0 sets `flag[0]` to `IN`, passes through the `until` (because P_1 is in `REQ`), and enters the critical section.
- **Case**: `REQ-IN`
  Without loss of generality, let P_0 and P_1 be in `REQ` and `IN`, respectively. If P_1 is in `IN`, which means P_1 could be before `until` and after `flag[1] = IN_CS`, or P_1 is actually in its critical section. In both cases, `flag[1] != OUT_CS` holds in P_0.
  - If P_1 is in its critical section, P_1 sets `turn` to 1 at the end of its enter protocol. As a result, P_0 stays in `while` and cannot enter.
  - If P_1 is in the enter protocol right after `flag[1] = IN_CS` and before `until`. In this case, `flag[1] != OUT_CS` still holds. Depending on the value of `turn`, P_0 may stay in `while` or proceed to execute `until`. However, because `flag[1]` is `IN_CS`, P_0 loops back and cannot enter. Of course, P_1 will enter eventually. Once this happens, we have the above case.
- **Case**: `IN-IN`
  Assume that P_0 is between `flag[0] = IN_CS` and `until`. Then, P_1 may be at the same spot or P_1 may be in its critical section.
  - Both P_0 and P_1 are between `flag[0] = IN_CS` and `until`:
    They both see the condition in `until` holds and they both loop back. As a result, we have the earlier `REQ-REQ` case.
  - P_0 is between `flag[0] = IN_CS` and `until` but P_1 is in its critical section:
    In this case P_0 loops back to become `REQ`. Again, it is the earlier `REQ-IN` case!

  Because P_0 and P_1 cannot be in the critical section at the same time in all possible cases, mutual exclusion holds.
  Note that this "direct" observation, although a bit lengthy, actually can help you understand more about this algorithm. This solution is the two-process version of a general n-process version due to Donald Knuth. ∎