CS3331 Concurrent Computing Exam 1
Fall 2013

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]** 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`. ∎

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:

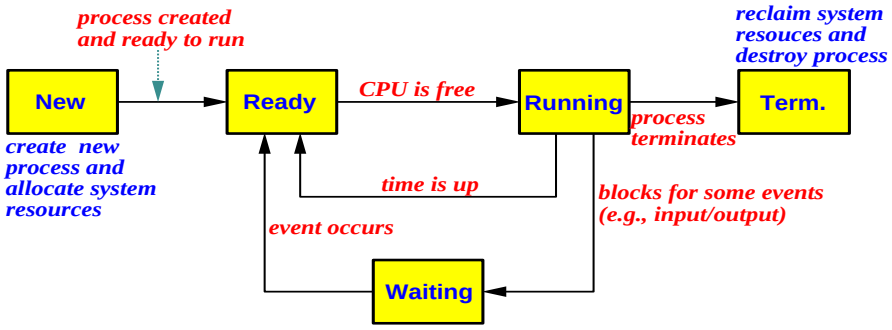
   - Suspend A's execution
   - Transfer the control to the CPU scheduler. A CPU mode switch may be needed.

- 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]** Enumerate the major differences between kernel-supported threads and user-level threads.

Answer: Kernel-supported threads are threads directly supported by the kernel. The kernel does thread creation, termination, joining, and scheduling in kernel space. User threads are supported at the user level and are not recognized by the kernel. Usually, a library running in user space provides all support for thread creation, termination, joining and scheduling. Due the kernel involvement, the overhead of managing kernel-supported threads is higher than that of user threads.

Since there is no kernel intervention, user threads are more efficient than kernel threads. On the other hand, in a multiprocessor environment, the kernel may schedule kernel-supported threads to run on multiple processors, which is impossible for user threads because the kernel does not schedule user threads. Additionally, since the kernel does not recognize and schedule user threads, if the containing process or its associated kernel-supported thread is blocked, all user threads of that process (or kernel thread) are also blocked. However, blocking a kernel-supported thread will not cause all threads of the containing process to be blocked.

See page 5 and page 6 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.

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

(b) **[10 points]** Explain the progress and bounded waiting conditions and enumerate their differences. **Note that there are two questions.**

Answer:

- **Progress:** If no process is executing in its critical section and some processes wish to enter their corresponding critical sections, then
  - Only those processes that are waiting to enter can participate in the competition (to enter their critical sections).
  - No other processes can influence this decision.
  - This decision cannot be postponed indefinitely (*i.e.*, making a decision in finite time).
- **Bounded Waiting:** After a process made a request to enter its critical section and before it is granted the permission to enter, there exists a bound on the number of times that other processes are allowed to enter. Hence, even though a process may be blocked by other waiting processes, it will only wait a bounded number of turns before it can enter.

The progress condition only guarantees the decision of selecting a process to enter a critical section will not be postponed indefinitely. It does not mean a waiting process will enter its critical section eventually. In fact, a process may wait forever because it may never be selected, even though every decision is made in finite time.

On the other hand, the bounded waiting condition guarantees that a process will enter the critical section after a bounded number of turns. Therefore, starvation is not a violation of the progress condition. Instead, starvation violates the bounded waiting condition.

See page 16 and page 17 of `05-Sync-Basics.pdf`. ∎

5. **Problem Solving:**

(a) **[15 points]** Jason and John are sharing an apartment, and they also share the responsibility of buying milk. To avoid the possibility of both buying, they come up with the following "algorithm":

> As soon as you arrive home, you leave a signed note on the fridge's door. Only then you check, and if you find that there is no milk and there is no note (other than yours), then you go and buy milk, put the milk in the fridge, and remove your note.

Their pseudo-code looks like the following:

Program for Jason	**Program for John**
Jason leaves note	John leaves note
if (no note from John) **then**	**if** (no note from Jason) **then**
if (no milk) **then**	**if** (no milk) **then**
buy milk	buy milk
end if	**end if**
end if	**end if**
remove Jason's note	remove John's note

Jason and John both suspect that they may end up no milk at all! Since they did not take a concurrent computing course, they are not very certain how this can happen. Use

a step-by-step execution of the above "algorithm" to show that Jason and John **can** end up with no milk at all. Note that *Jason and John cannot see and talk to each other*. **You must provide a clear step-by-step execution of the algorithm with a convincing argument. Vague and unconvincing arguments receive no points.**

Answer: This is a very easy problem. The following is a possible execution sequence showing Jason and John can end up no milk at all.

Jason	John	Comments
Leave a note signed Jason		Jason is home
	Leave a note signed John	John is home
"no note John" is false		Jason sees John's note
	"no note Jason" is false	John sees Jason's note
Remove Jason's note	Remove John's note	Remove their notes

In this way, Jason and John do not check to see if the fridge has milk or not, and, as a result, if the fridge does not have milk they will end up no milk. ∎

(b) **[15 points]** Consider the following solution to the critical section problem for two processes P_0 and P_1.

```
bool  flag[2];   // global flags, initially FALSE
int   turn;      // global turn variable, initially 0 or 1

Process 0                Process 1

// enter protocol        // enter protocol
flag[0] = TRUE;          flag[1] = TRUE;        // I am interested
while (flag[1]) {        while (flag[0]) {      // wait if you are interested
  if (turn == 1) {         if (turn == 0) {     //   if it is your turn ...
    flag[0] = FALSE;         flag[1] = FALSE;   //     I am no more interested
    while (turn != 0)        while (turn != 1)  //     wait for my turn
      ;                        ;
    flag[0] = TRUE;          flag[1] = TRUE;    //     let me try again
  }                        }
}                        }

// in critical section   // in critical section

// exit protocol         // exit protocol
turn = 1;                turn = 0;              // it is your turn now
flag[0] = FALSE;         flag[1] = FALSE;       // I am not interested
```

Show that this solution satisfies the mutual exclusion condition. *You will receive* **zero** *point if you provide a vague answer* **without** *a convincing argument and/or you prove this by example.*

Answer: Process 0 can enter its critical section if the condition of the `while` loop fails. This means `flag[1]` must be `FALSE`. Process 0 sets `flag[0]` to `TRUE` before entering or going back for the next iteration of the `while` loop. Therefore, process 0 can enter its critical section if `flag[0]` is `TRUE` **and** `flag[1]` is `FALSE`. Note that `turn` plays no role here. Suppose process 0 runs fast, so fast that it sets `flag[0]` to `TRUE` and sees `flag[1]` being `FALSE`. Then, process 0 enters no matter what value `turn` has.

By the same reason, process 1 can enter its critical section if `flag[1]` is `TRUE` **and** `flag[0]` is `FALSE`.

If process 0 and process 1 are both in their critical sections at the same time, `flag[0]` (and `flag[1]`) is both `TRUE` and `FALSE`, which is impossible. Consequently, process 0 and process 1 cannot be in their critical section at the time, and the mutual exclusion conditions is satisfied. ∎