CS3331 Concurrent Computing Solution 1
Fall 2014

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 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.

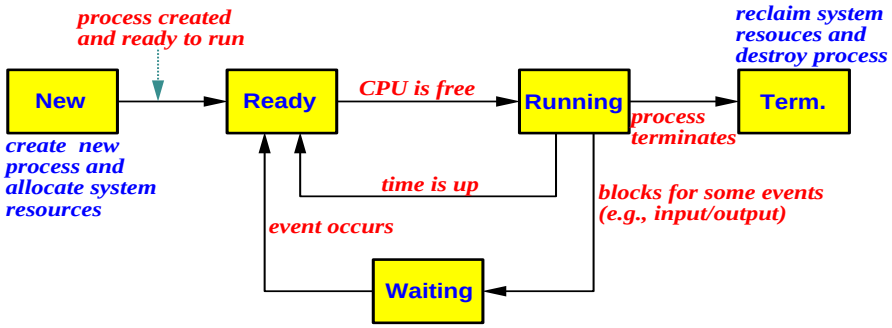
   See page 10 and page 11 of `03-Process.pdf`. ∎

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

Answer: Kernel-supported threads are threads directly handled 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 to 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. 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 10 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.

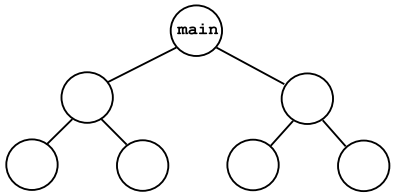
See page 16 and page 18 of `05-Sync-Basics.pdf`. ∎

5. **Problem Solving:**

(a) **[15 points]** Design a C program segment so that the `main()` creates two child processes with `fork()`, each of these two child processes creates two child processes, etc such that the parent-child relationship is a perfectly balanced binary tree of depth `n` with `main()` at the root. The

depth `n` have already stored a valid positive integer. The `main()` prints its PID, and each child process prints its PID and its parent's PID.

The following diagram shows a tree of processes of depth 3 (*i.e.*, `n` = 3). Your program segment must be correct for any valid value of `n` > 0. Only providing a program segment for `n` = 3 will receive **zero** point. To save your time, you do not have to perform error checking. However, proper wait and exit are expected.



Answer: The parent forks two child processes and waits for them. Each child process prints the needed information, and loops back to fork its child processes. However, if a child process is a leaf node, it just prints and exits. This idea is illustrated in the code segment below.

```
#include <stdio.h>

#define  PRINT { printf("My PID = %ld   My PPID = %ld\n", getpid(), getppid()); }

int main(int argc, char **argv)
{
   int i, n;

   printf("main()'s PID = %ld\n\n", getpid());
   n = atoi(argv[1])-1;
   for (i = 1; i <= n; i++)   // for each level
      if (fork() > 0)         // parent forks the 1st child
         if (fork() > 0) {    // parent forks the 2nd child
            wait(NULL);       // parent waits them to complete
            wait(NULL);
            exit(0);          // and exit
         }
         else {               // the 2nd child process
            PRINT             // print the needed info
            if (i == n)       // if it is a leaf node
               exit(0);       // exit w/o coming back to fork
         }
      else {                  // the 1st child process
            PRINT             // print the needed info
            if (i == n)       // if it is a level node
               exit(0);       // exit w/o coming back to fork
      }
}
```

You may also do the same using recursion as binary tree creation is basically recursive. Note that a process prints only if the level number is larger than 1. If the level number is 1, printing `getpid()` and `getppid()` will print the PID of the `main()` and the PID of the shell.

```
#include <stdio.h>

#define PRINT   { printf("My PID = %ld   My PPID = %ld\n", getpid(), getppid()); }

void pCreate(int i, int n)
{
   if (i < n) {              // still have level to go
      if (fork() > 0) {      // fork the 1st child
         if (fork() > 0) {   // fork the 2nd child
            if (i > 1)       // if this is higher than level 1
               PRINT         //    print
            wait(NULL);      // wait for both child processes
            wait(NULL);
            exit(0);         // exit
         }
         else                // the 2nd child goes for level i+1
            pCreate(i+1, n);
      }
      else                   // the 1st child goes for level i+1
         pCreate(i+1, n);
   }
   else {                    // lead node
      PRINT                  // print info only w/o forking
      exit(0);
   }
}


int main(int argc, char **argv)
{
   int i, n;

   printf("main()'s PID = %ld\n\n", getpid());
   n = atoi(argv[1]);
   pCreate(1, n);
   wait(NULL);
}
```

(b) [**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 $\texttt{max}(a, b)$ 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 $b = d$. ∎