CS3331 Concurrent Computing Exam 1 Solutions Spring 2019

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]** 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:

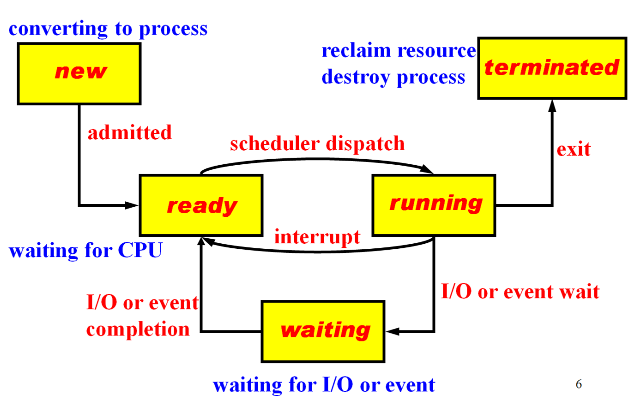
   - The executing program is suspended and control is transferred to the operating system. Mode switch may be needed.
   - A general routine in the operating system examines the received interrupt and calls the interrupt-specific handler.
   - After the interrupt is processed, a context switch transfers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

   See pp. 6–7 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`. ■

(b) **[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.

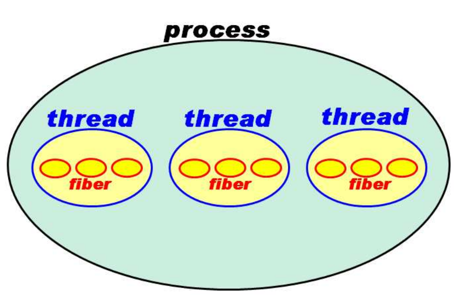
See pp. 10–11 of `03-Process.pdf`. ■

3. **Threads**

(a) **[10 points]** Do your best to define and compare the concepts of *process*, *thread* and *fiber*. In particular, how are they scheduled? **You have to define each first, followed by a comparison of these three. You are expected to explain the way of scheduling each. Without doing so, you will receive no credit for this problem.**

Answer: A *process* is the execution of a program; a *thread* (*i.e.*, *lightweight process*) is a unit of CPU execution that is created by a process; and a *fiber* is a lightweight thread, created by a thread, just like a thread being a lightweight process. A comparison can be based on hierarchy, resource usage and scheduling as follows:

- **Hierarchy:** Processes create threads, and threads create fibers. Thus, a process contains the threads it creates, and a thread contains the fibers it creates. This hierarchy is shown below:



- **Resource Sharing:** A *process* must acquire all resources for that process to run properly. These resources include files, memory, registers, etc.
  A *thread* is a lightweight process. All peer threads created within a process share with each other files opened by the containing process, memory allocated to the containing process, etc. Thus, a thread has a program counter, a register set, and a stack, and shares the resources acquired by the containing process with other peer threads.
  A *fiber* also has a program counter, a subset of the registers, and a stack. A fiber shares all resources acquired by the containing threads with other peer fibers.
- **Scheduling:** *processes* are scheduled by the CPU scheduling in the kernel. *Threads* can be user-level threads that are scheduled by a thread scheduler built into a thread library in a user address space, or can be kernel-supported threads that are scheduled by the thread scheduler in the kernel. *Fibers*, on the other hand, are scheduled by a co-operative policy. In other words, a fiber gives up the CPU in a voluntary way, usually through the execution of the `YIELD` statement.

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 manipulate 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

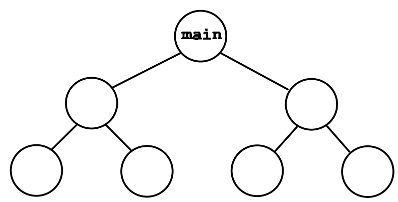
Stating that "`count++` followed by `count--`" or "`count--` followed by `count++`", even using machine instructions, produces different results and hence a race condition is **incomplete**, because the two processes do not access the shared variable `count` concurrently. Note that the use of higher-level 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`.

5. **Problem Solving:**

(a) **[10 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 been 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 $\mathtt{n} > 0$. 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 own 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 leaf 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 {                    // leaf 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);
}
```

There are two commonly seen problems in your solutions. **First**, no `wait()` statements were included. If you do not include `wait()` statements, some child processes can become orphans whose parent is the `init` process with PID being 1, and, as a result, the parent-child relationship is incorrect. **Second**, even though you are allowed to use `printf()`, separating the parent PID and the child PID into two `printf()` calls is an incorrect approach. It is because the two `printf()` calls may prints their output lines scattered all over in the output report. As a result, the parent-child relationship is, again, not shown properly. ■

(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 step-by-step <u>**execution sequences**</u> 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 <u>no</u> points.**

<u>**Answer**</u>: 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*** **(Answer =** `3`**)**

`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*** **(Answer =** `5`**)**

`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*** **(Answer =** `6`**)**

`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, where `status[ ]` is a Boolean array of two elements and `turn` is an integer variable. Furthermore, there are three constants indicating the status of a process, where `COMPETING`, `IN_CS` and `OUT_CS` mean competing to enter the critical section, in the critical section, and out of the critical section. Note that `status[ ]` and `turn` are global variables shared by both processes.

```
int  status[2];   // status of a process
int  turn;        // initialized to either 0 or 1

        Process 0                          Process 1
        =========                          =========
 1 status[0] = COMPETING;             status[1] = COMPETING;
 2 while (status[1] == COMPETING) {   while (status[0] == COMPETING) {
 3    status[0] = OUT_CS;                status[1] = OUT_CS;
 4    repeat until (turn == 0);          repeat until (turn == 0 || turn == 1);
 5    turn = 0;                          turn = 1;
 6    status[0] = COMPETING;             status[1] = COMPETING;
 7 }                                  }
                          // critical section
 8 status[0] = OUT_CS;                status[1] = OUT_CS;
 9 turn      = 0;                     turn = 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: Checking the short code you should be able to see the following:

- A process that reaches its `while` loop, whether it is the first time or loops back, always has its own `status[ ]` being `COMPETING` (Line 1 and Line 6).
- Then, if the other process is **NOT** `COMPETING`, this process breaks its own `while` loop and enters its critical section.
- So far, we know that P_0 is in its critical section `status[0]` is `COMPETING` **AND** `status[1]` is not `COMPETING`. By the same reason, if P_1 is in its critical section `status[1]` is `COMPETING` **AND** `status[0]` is not `COMPETING`.
- There is more to say. Let us look at the variable `turn`. Suppose P_1 is not `COMPETING`. In this case P_0 enters its critical section immediately and `turn` plays no role. On the other hand, if P_0 enters its `while` loop, P_0 sets `turn` to 0 before reaching the end of its `while` loop. Therefore, as long as P_0 enters its `while` loop, we have `status[0]` and `status[1]` being `COMPETING` and not `COMPETING`, respectively, and `turn` being 0. By the same reason, as long as P_1 enters its `while` loop, P_1 can enter its critical section, we have `status[0]` and `status[1]` being not `COMPETING` and `COMPETING`, respectively, and `turn` being 1.

Because we do not know whether a process in its critical section enters its critical section without getting into its `while` loop, the role of `turn` and `status[ ]` being `OUT_CS` may not be useful and hence should not be used. As a result, we can only rely on `status[ ]` and we have:

- P_0 is in its critical section if and only if `status[0]` is `COMPETING` and `status[1]` is not `COMPETING`.
- P_1 is in its critical section if and only if `status[1]` is `COMPETING` and `status[0]` is not `COMPETING`.

Consequently, if P_0 and P_1 are both in their critical section at the same time, then `status[0]` (and `status[1]`) must be `COMPETING` and not `COMPETING` at the same time, which is absurd. Thus, we have a contradiction and the mutual exclusion condition is met. ∎