CS3331 Concurrent Computing Exam I Solutions
Fall 2020

1. **Practice**

   (a) **[10 points]** In *Programming Assignment I* you were asked to write a program using the `long` type to compute the factorials of 1!, 2!, 3!, . . ., 25!. Answer the following questions as accurate as possible. Vague answers receive no point.

      i. **[3 points]** What were your findings?

      ii. **[7 points]** Provide a detailed discussion to explain why you get the results in Problem 1(a)i.

   Answer: The following has the answer to both questions together.

   Question (i): Your program output should look like the following if you ran your program on an Intel-based CPU (*i.e.*, a CS lab machine):

```
 0! =                    1
 1! =                    1
 2! =                    2
 3! =                    6
 4! =                   24
 5! =                  120
 6! =                  720
 7! =                 5040
 8! =                40320
 9! =               362880
10! =              3628800
11! =             39916800
12! =            479001600
13! =           6227020800
14! =          87178291200
15! =        1307674368000
16! =       20922789888000
17! =      355687428096000
18! =     6402373705728000
19! =   121645100408832000
20! =  2432902008176640000
21! = -4249290049419214848
22! = -1250660718674968576
23! =  8128291617894825984
24! = -7835185981329244160
25! =  7034535277573963776
```

   The C language has the minimum and maximum of each type declared in the header file `limits.h`. On my MacBook Air/Pro, Mac Mini and iMac Pro under `gcc`, the minimum and maximum values of the `long int` type, which are the same as the `long long int` type, are `-9223372036854775808` and `9223372036854775807`, respectively. This is, of course, machine and compiler dependent.

   Your findings should include the following: (1) up to 20! the output looks reasonable and correct, (2) 21! turns to be negative (which should not be the case), and (3) looking further down 25! is smaller than 23!. These are all odd things.

   Question (ii): There was no programming error and there was no interrupt due to these strange phenomena, because unlike floating-point arithmetic, integer overflow and underflow do not cause interrupts. When multiplying two large integers with binary arithmetic, if the result cannot fit into a register, the stored result only includes the least significant bits. More precisely, if we use $k+1$ bits for signed integer computation with 1 sign bit, the minimum and maximum are likely to be $-2k-1$ and $2k-1$, respectively. If the computed result is larger than $2k-1$, only the last k bits would be stored. This is usually referred to as *wrapping*.

   For example, suppose we use a 4-bit register for multiplication. Then, multiplying $0110_2 = 6_{10}$ and $0101_2 = 5_{10}$ yields $30_{10} = 11110_2$. Because we only use a 4-bit register for computation, the stored result would be the last 4 bits 1110_2. However, because we use signed integers, the first bit is the sign bit

(*i.e.*, 0 positive and 1 negative), and 1110_2 actually means -2_{10} in the 2's complement representation. Note that different computer architectures could produce different results.

In summary, the main reason of getting negative results and non-decreasing results (*i.e.*, $(k+h)! < k!$ where h is a positive integer) in the process of computing $n!$ is just because integer overflow could make the sign bit 1. This concept should have been covered in your computer organization or a similar course. ∎

2. **Basic Concepts**

   (a) **[10 points]** What are the 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 mode. 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]** Define *interrupts* and *traps*, and provide a detailed account of the procedure that an operating system handles an interrupt. **There are two questions**.

   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 corresponding interrupt-specific handler.
   - After the interrupt is processed, a context switch transfers control back to a suspended process and resumes its execution. Of course, mode switch may be needed. Note that the resumed process may not be the suspended one due to this interrupt.

   See pp. 6–7 of `02-Hardware-OS.pdf`. ∎

3. **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.

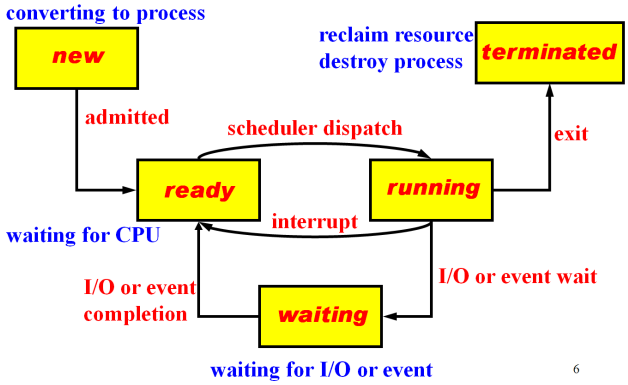
   Answer: The following state diagram is taken from my class note.

   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*? What are the most important items in a context? Provide a detail description of *all* activities of a *context switch*. **There are three questions**.

Answer: A process needs some system resources (*e.g.*, registers, 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`. ∎

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.**

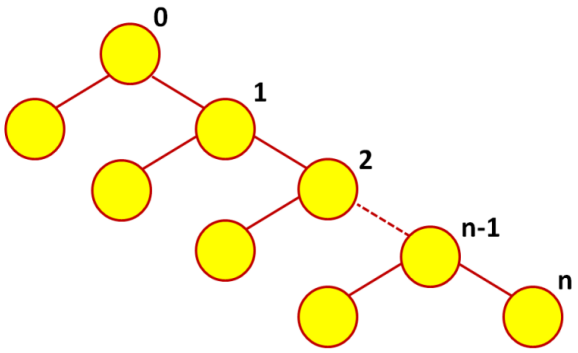
Thread_1	Thread_2	*Comment*
do somthing	do somthing	`count = 10` initially
`LOAD count`		`Thread_1` executes `count++`
`ADD #1`		
	`LOAD count`	`Thread_2` executes `count--`
	`SUB #1`	
`SAVE count`		`count` is 11 in memory
	`SAVE count`	Now, `count` is 9 in memory

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]** Write a C program segment to create a set of processes so that each of which has two child processes. One of these two child processes exits quickly and the other continues this process as shown in the diagram below:



The above diagram shows a tree of processes of depth `n`, where the `main()` is marked as `0` and the last level is `n`. The value of `n` is an input from a command line argument from `argv[1]`. Your program segment must be correct for any valid value of $n > 0$. Only providing a program segment for a special number such as 2 or 3 will receive **zero** point. To save your time, you do not have to perform error checking, you may assume that the calls to `fork()` are always successful, and you may use `printf()` to print the PID and PPID of a processes. However, proper `wait()` and `exit()` are required.

Answer: This program requires that each process creates two child processes, one of which would end immediately after printing the needed information (the left leg), while the other prints the needed information and continues to create more child processes (the right leg). Therefore, each process creates a child process which prints the needed output and exits, creates a second process to print its information, and then waits for both child processes to complete and exits. The second created process **DOES NOT** exit after printing its information. Instead it loops back to start the next iteration doing essentially the same as its parent does. The following is a possible program to implement stated above:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int    n, i;
    pid_t  pid;

    printf("The root process %d, ppid = %d\n\n", getpid(), getppid());

    n = atoi(argv[1]);
    for (i = 1; i <= n; i++) {
        if ((pid = fork()) == 0)  {  // child
             printf("My ID = %d   My PPID = %d\n", getpid(), getppid());
             exit(0);
        }
        else { // parent
            if ((pid = fork()) == 0) {
                printf("My ID = %d   My PPID = %d\n", getpid(), getppid());
            }
            else {
                wait(NULL);
                wait(NULL);
                exit(0);
            }
        }
    }
}
```

This is a rather simple problem similar to the problems asked in *Programming Assignment I*. ■

(b) **[15 points]** Consider the following two processes, *A* and *B*, to be run concurrently using a shared memory for the `int` variable `x`.

```
Process A                    Process B
---------                    ---------
for (i = 1; i <= 2; i++)     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. **You must use clear step-by-step execution sequences of the above processes with a convincing argument. Any vague and unconvincing argument receives no points.**

Answer: Obviously, the answer must be in the range of 0 and 4. It is non-negative, because the initial value is 0 and no subtraction is used. It cannot be larger than 4, because the two `x++` statements and `x = 2*x` together can at most double the value of `x` twice.

The easiest answers are 2, 3 and 4 if `x = 2*x` executes before, between and after the two `x++` statements, respectively. The following shows the possible execution sequences in higher level statement interleaving.

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

`x = 2*x` **is between the two** `x++`		
Process 1	Process 2	`x` in memory
`x++`		1
	`x = 2*x`	2
`x++`		3

`x = 2*x` **is after both** `x++`		
Process 1	Process 2	`x` in memory
`x++`		1
`x++`		2
	`x = 2*x`	4

The situation is a bit more complex with instruction interleaving. Process B's `x = 2*x` may be translated to the following machine instructions:

```
LOAD x
MUL  #2
SAVE x
```

Because the `LOAD` retrieves the value of `x`, and the `SAVE` may change the current value of `x`, the results depend on the positions of `LOAD` and `SAVE`. The following shows the result being 0. In this case, `LOAD` loads 0 *before* both `x++` statements, and the result 0 is saved *after* both `x++` statements.

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

If the `SAVE` executes between the two `x++` statements, the result is 1.

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

You may try other instruction interleaving possibilities (*e.g.*, replacing the `x++` with machine instructions) and the answers should still be in the range of 0 and 4. ■

(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 two constants indicating the status of a process, where `COMPETING` and `OUT_CS` mean competing to enter the critical section and out of the critical section, respectively. Note that `status[ ]` and `turn` are global variables shared by both processes, and `status[ ]` and `turn` are global variables shared by both processes, and `status[ ]` are initialized to `OUT_CS` while `turn` is initialized to either 0 or 1.

```
int  status[2];   // status of a process initialized to OUT_CS
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 == -1 || 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      = -1;  //I am OUT        turn = -1;   // I am OUT
```

Use **proof-by-contradiction** to show 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 do not prove by contradiction.*

Answer: Checking the short code you should be able to see the following:

- When a process reaches its `while` loop, whether it is the first time or loops back, this process always has its own `status[ ]` set to `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` due to the statements on Line 1 and Line 6, **AND** `status[1]` is not `COMPETING` due to the `while` statement. By the same reason, if P_1 is in its critical section `status[1]` is `COMPETING` **AND** `status[0]` is not `COMPETING`. This is good enough to derive a contradiction for proving mutual exclusion.
- However, 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 enters its critical section without getting into its `while` loop, and because we do not know which process modifies `turn` first, the role of `turn` and `status[ ]` being `OUT_CS` are not useful and should not be used. Thus, 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`.

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. ∎