CS3331 Concurrent Computing Exam 1 Solutions
Fall 2016

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. 6–7 `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 without interleaving and cannot be split by other instructions. When an atomic instruction is recognized by the CPU, we have the following:

   - All other instructions being executed in various stages by the CPUs are suspended (and perhaps re-issued later) until this atomic instruction finishes.
   - No interrupts can occur.

   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 pp. 12–13 of `02-Hardware-OS.pdf`. ∎

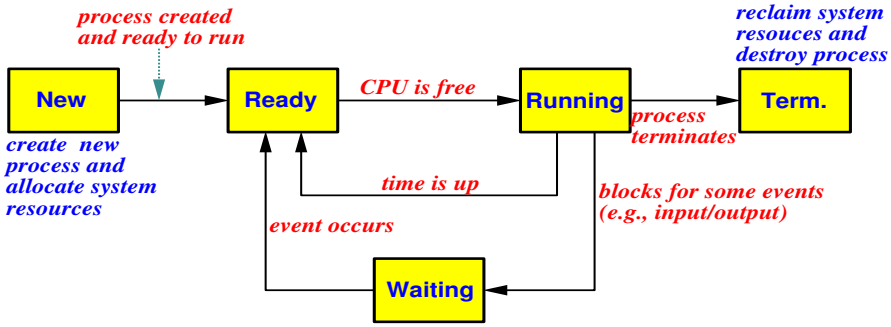
2. **Processes**

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

3. **Threads**

(a) **[10 points]** What is *thread cancellation*? How many commonly used thread cancellation types are there. Name the types and provide an explanation for each type. **Note that there are TWO questions.**

Answer: *Thread cancellation* means terminating a thread **before** its completion. The thread to be cancelled is the *target* thread.

There are two types:

- *Asynchronous Cancellation*: The target thread terminates immediately.

- *Deferred Cancellation*: The target thread can periodically check if it should terminate, allowing the target thread an opportunity to terminate itself in an orderly way. The point a thread can terminate itself is a *cancellation point*.

See pp. 19–20 of `04-Thread.pdf`. ∎

(b) **[5 points]** Explain the meaning of *thread safe*?

Answer: A library that can be used by multiple threads properly is a *thread-safe* one.

See p. 21 of `04-Thread.pdf`. ∎

4. **Synchronization**

(a) **[10 points]** A good solution to the critical section problem must satisfy three conditions: *mutual exclusion*, *progress* and *bounded waiting*. Explain the meaning of the *progress* condition. Does starvation violate the progress condition? **Note that there are two problems. You should provide a clear answer with a convincing argument. Otherwise, you will receive no credit.**

Answer: The *progress* condition consists of two components:

- If no process is executing in its critical section and some processes are waiting to enter their critical sections, then only those waiting processes can participate in the decision on which process can enter its critical section, and this decision must be made in finite time.
- Those processes that are not entering have no influence upon this decision.

Starvation means that a waiting process waits indefinitely and has no chance to enter. The *progress* condition only guarantees the decision of selecting a process to enter its critical section will be completed in finite time. It does not mean a waiting process will enter its critical section eventually. Therefore, starvation is not a violation of the progress condition. Instead, starvation violates the bounded waiting condition.

See pp. 19-20 and 24 of `05-Sync-Basics.pdf`. ∎

(b) **[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

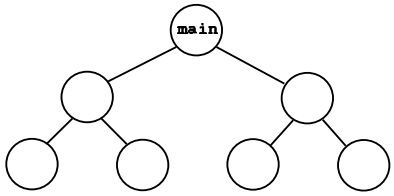
Stating that "`count++` followed by `count--`" or "`count--` followed by `count++`", even using machine instructions, produces different results and hence a race condition is **incomplete**, because the two processes do not access the shared variable `count` concurrently. Note that the use of higher-level 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 `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]** The following solution to the critical section problem has a global array `flag[2]` and an `int` variable `turn`.

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

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

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: This is a very simple problem and an exercise in a weekly reading list. In fact, it is an extension of Attempt II of `06-Sync-Soft-Hardware.pdf` (pp. 6–10) and its proof follows exactly the same logic.

For P_0 to enter its critical section, it sets `flag[0]` to `TRUE`. Once P_0 reaches its `while` loop, we have the following cases to consider:

- If P_0 is lucky and sees `flag[1]` being `FALSE` the first time, then P_0 enters its critical section. In this case, `flag[0]` and `flag[1]` are `TRUE` and `FALSE`, respectively.
- If P_0 enters the `while` loop because `flag[1]` is `TRUE`, then we have two cases:
  - If P_0 finds out `turn` is not 1, then P_0 goes back for the next iteration without changing the value of `flag[0]`.
  - If P_0 finds out `turn` is 1, then P_0 executes the "then" part of the `if` statement. Before it leaves the "then" part, P_0 sets `flag[0]` to `TRUE` again.

  Hence, no matter what has happened to the `if` statement, at the end of the `while` loop, `flag[0]` is always `TRUE`. If P_0 eventually enters its critical section, `flag[0]` is always `TRUE` and `flag[1]` is always `FALSE`.

In summary, if P_0 is in its critical section, we have `flag[0] = TRUE` and `flag[1] = FALSE`. Similarly, if P_1 is in its critical section, we have `flag[1] = TRUE` and `flag[0] = FALSE`.

Now, if P_0 and P_1 are *both* in their critical sections, `flag[0]` is set to `TRUE` by P_0 and sees `flag[1]` being `FALSE` from its `while` loop. By the same reason, P_1 sets `flag[1]` to `TRUE` and sees `flag[1]` being `FALSE` from its `while` loop. As a result, `flag[0]` (and `flag[1]`) must be both `TRUE` and `FALSE` at the same time. Because a variable can only holds one value, we have a contradiction. Consequently, the mutual exclusion condition is satisfied.

Note that no matter what the value of `turn` is, a process can enter its critical section only if it can pass its `while` loop. Therefore, the control variables are `flag[0]` and `flag[1]` rather than `turn`. You should read the code carefully. ∎