CS3331 Concurrent Computing Exam 1 Solutions
Fall 2018

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 of `02-Hardware-OS.pdf`. ∎

   (b) **[10 points]** What is an atomic instruction? What would happen if multiple CPUs/cores execute their atomic instructions? **Make sure you will explain atomic instructions fully. Otherwise, you may receive a lower or very low score.**

   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`. ∎

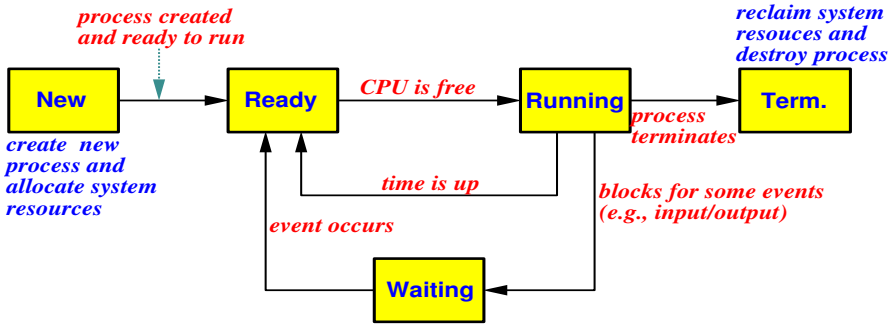
2. **Processes**

   (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]** The following is a buggy program using shared memory. It has multiple processes, each of which computes the sum of two elements with an increasing gap. More precisely, for process `i` the first iteration computes `a[i]+a[i-1]` and saves the result back to `a[i]`; the second iteration computes `a[i]+a[i-2]` and saves the result back to `a[i]`; the third iteration computes `a[i]+a[i-4]` and saves the result back to `a[i]`, etc. The gap is doubled in each iteration. This process continues until `i-gap`< 0. The programmer who wrote this program is certain that the bug is the way of using shared memory. Help him pinpoint the major problem. **You should elaborate your answer. A vague and/or unconvincing answer receives zero point. Additionally, the answer to this problem is very short; but, an elaboration is REQUIRED.**

```
 1. void main(int argc, char **argv)
 2. {
 3.     int   n, status, i;
 4.     char  buf[1024], temp[20];
 5.     int   *ShmPTR;

    // get shared memory and save the pointer to ShmPTR

 6.     n = atoi(argv[1]);
 7.     for (i = 2; i < n+2 ; i++)
 8.          ShmPTR[i] = atoi(argv[i]);

    // the parent creates n child processes and
    //    calls the i-th one with Child(i, ShmPTR)
    //    where i = 0, 1, 2, ..., n-1

 9.     for (i = 0; i < n; i++)  // wait all child process
10.          wait(&status);
11.     for (i = 0; i < n; i++) {
12.         sprintf(temp, "%5d", ShmPTR[i]);
13.         strcat(buf, temp);
14.     }
15.     strcat(buf, "\n");
16.     write(1, buf, strlen(buf));
17.     exit(0);
18. }

19. void Child(int i, int *ShmPTR)
20. {
21.     int  gap = 1;

22.     while (i-gap >= 0) {
23.          ShmPTR[i] = ShmPTR[i] + ShmPTR[i-gap];
24.          gap = gap + gap;
25.     }
26.     exit(0);
27. }
```

Answer: The major problem is in the indeterministic execution order of the statement `a[i] = a[i-gap] + a[i]`. For simplicity, we only look at P_1 and P_2 with `gap` being 1. In this way, only array section `a[0..2]` is involved and P_1 and P_2 execute `a[1] = a[0] + a[1]` and `a[2] = a[1] + a[2]`, respectively. Other cases and `gap` values can be treated the same way. Assume initially `a[2] = { 3, 5, 7 }`.

- If P_1 runs faster and executes `a[1] = a[0] + a[1]` before P_2 can executes its counterpart.
  - P_1 sees `a[3] = { 3, 5, 7 }` and computes `a[1]` as $8 =$ `a[0]+a[1]` $= 3+5$.
  - After P_1 completes, `a[3] = { 3, 8, 7 }`.
  - When P_2 executes `a[2] = a[1] + a[2]`, the new value for `a[2]` is $15 =$ `a[1]+a[2]` $= 8+7$.
  - Therefore, the result is `a[3] = { 3, 8, 15 }`.
- If P_2 runs faster and executes `a[2] = a[1] + a[2]` before P_1 can executes its counterpart.
  - P_2 sees `a[3] = { 3, 5, 7 }` and computes `a[2]` as $12 =$ `a[1]+a[2]` $= 5+7$.
  - After P_2 completes, `a[3] = { 3, 5, 12 }`.
  - When P_1 executes `a[1] = a[0] + a[1]`, the new value for `a[1]` is $8 =$ `a[0]+a[1]` $= 3+5$.

- Therefore, the result is `a[3] = { 3, 8, 12 }`.

The above results depend on the order of execution of **higher-level statements**, and, as a result, the final outcome is non-deterministic. If you use instruction-level interleaving, the final outcome may be more complex.

Note that this problem is exactly the one shown on pp. 3-4 of `05-Sync-Basics.pdf`. ∎

3. **Threads**

(a) **[10 points]** Why is handling threads cheaper than handling processes? **Provide an accurate account of your findings. Otherwise, you risk a lower grade.**

Answer: There are two major points.

- **Resource Consumption and Sharing:** A thread only requires a thread ID, a program counter, a register set and a stack, and shares with other peer threads in the same process its code section, data section, and other OS resources (*e.g.*, files and signals). Fewer resource consumption means less allocation overhead.
- **Faster in Handling Context Switching:** Since a thread has fewer items than a process does, it is faster to perform thread-based context switching as fewer data items have to be saved and loaded. Moreover, the creation, destroy and joining of threads would also be cheaper.

Therefore, handling threads is cheaper than handling processes.

See pp. 2–4 of `04-Thread.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:**

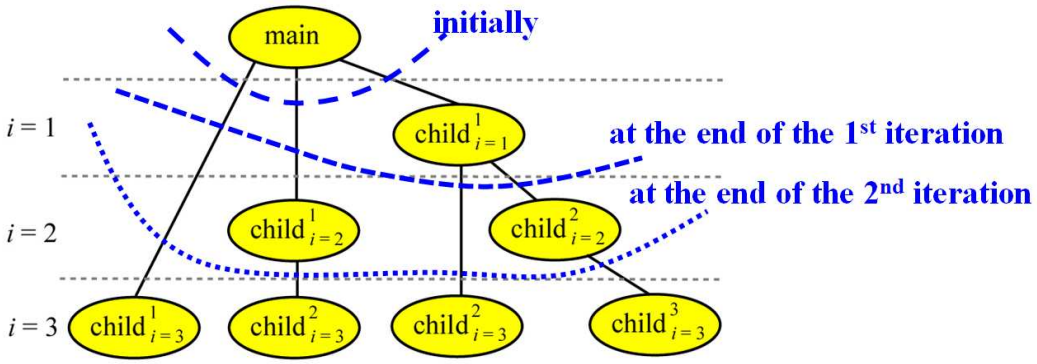
(a) **[10 points]** Consider the following program segment. Suppose all `fork()` calls are successful. Answer the following questions: **(1)** Draw a diagram showing the parent-child relationship of *all* involved processes, the main program included, and provide an explanation how this relationship is obtained. **Vague and not convincing arguments receive zero point.** **(2)** This program segment uses `n = 3`. How many processes will be created if `n` is set to a positive integer k? You don't need to draw a diagram; but, you still have to provide a justification for your answer. **Vague and not convincing argument receive zero point.**

```
int i, n = 3;

for (i = 1; i <= n; i++)
     fork();
```

Answer: The following has the answer to each question:

Question (1): Keep in mind that each `fork()` creates a child process and the child process receives a separate address space that is identical to of that of the parent. As a result, after a child process is created, the value of `i` of this child is the same as that of the parent. After the `fork()` call, the parent and its child run concurrently. Initially, only the parent process runs. See the diagram below.



The creation steps are as follows:

- **i = 1**: the `main` creates a child, $\text{child}^1_{i=1}$, where the superscript 1 indicates the "generation" and the subscript has the value of i. Now we have two processes running, `main` and $\text{child}^1_{i=1}$. Both of them go back to the beginning of the `for` loop and advance to $i = 2$.
- **i = 2**: Both `main` and $\text{child}^1_{i=1}$ reach the `fork()` call and both create a child. The `main` creates a child $\text{child}^1_{i=2}$, which is still the first generation with $i = 2$ as shown in the diagram. Process $\text{child}^1_{i=1}$ also executes `fork()` to creates a child, $\text{child}^2_{i=2}$, which is the second generation in the hierarchy with $i = 2$. Now, we have four processes running, `main`, $\text{child}^1_{i=1}$ (`main`'s first child created when $i = 1$), $\text{child}^1_{i=2}$ (`main`'s second child created when $i = 2$), and $\text{child}^2_{i=2}$ ($\text{child}^1_{i=1}$'s child created when $i = 2$).
- **i = 3**: Each of the above mentioned four processes executes the `fork()` call, and creates one child process. As a result, there are eight processes in total after the `for` loop completes as shown in the diagram.

Question (2): Since each process creates one and only one child process in each iteration, the number of processes is doubled. We have the `main` initially. At the end of $i = 1$, we have $2 = 2 \times 1 = 2^1$ processes; at the end of $i = 2$, we have $4 = 2^1 \times 2 = 2^2$ processes; and at the end of $i = 3$, we have $8 = 2^2 \times 2 = 2^3$ processes. Continuing this observation, at the end of $i = k$, we should have 2^k processes.

This observation can easily be proved with mathematical induction.

- **Base Case:** If $k = 0$, we have `main` only and hence $2^0 = 1$ process.
- **Induction Step:** Assume that the proposition holds for $k-1$. That is, we assume that at the end of $i = k-1$ there are 2^{k-1} processes. Since each process only creates one child process in each iteration, when $i = k$ each of the 2^{k-1} processes creates one child process, making the total number of processes $2 \times 2^{k-1} = 2^k$. Therefore, at the end of $i = k$ we have 2^k processes.

From the above induction proof, the number of processes at the end of iteration k is 2^k. ∎

(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 **execution sequences** 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 no points.**

Answer: 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]** The following solution to the critical section problem has a global array `flag[2]` and an `int` variable `turn`.

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

    Process 0                      Process 1
    ---------                      ---------
 3. flag[0] = TRUE;                flag[1] = TRUE;          // I am interested
 4. while (flag[1]) {              while (flag[0]) {        // wait if you are interested
 5.    if (turn == 1) {               if (turn == 0) {     //    if it is your turn ...
 6.       flag[0] = FALSE;               flag[1] = FALSE;  //       I am no more interested
 7.       while (turn != 0)              while (turn != 1) //       wait for my turn
 8.          ;                              ;
 9.       flag[0] = TRUE;                flag[1] = TRUE;   //       let me try again
10.    }                              }
11. }                              }
                 // in critical section
12. turn = 1;                      turn = 0;                // it is your turn now
13. 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` being not 1, then P_0 goes back for the next iteration without changing the value of `flag[0]`.
  - If P_0 finds out `turn` being 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. ∎