User Id: mame23

CSE 303: Quiz #2

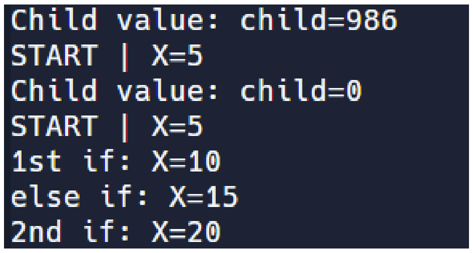
Due Friday Oct 7th, 2022 at 11:59 PM

The quiz has THREE questions. Please submit your answer on CourseSite as a pdf whose name is exactly your user Id and the "pdf" extension (e.g., abc123.pdf) before the deadline.

Question 1: Consider the following program (assume that fork will never fail).

```
main (int argc, char ** argv) {

      int child = fork(); //1
      int x = 5;          //2
      if (child == 0) {   //3
            x += 5;      //4
      } else {            //5
            child = fork(); //6
            x += 10;        //7
            if(child) {     //8
                  x += 5;   //9
            }
      }
}
```



```
Child value: child=986
START | X=5
Child value: child=0
START | X=5
1st if: X=10
else if: X=15
2nd if: X=20
```

How many different copies of the variable x are there? What are their values when their process finishes? Explain your answer.

From running the program: A child will be created using the fork () process, so there are two processes running the main function concurrently. X = 5 in line 2 for the parent and the child. If child == 0 checks if the child process is the parent. X will not increment on line 4, so it will create another child on line 6 with x = 5. Line 7 will increment x to be 10 and then 15 on line 9. In the end, there will be three processes: A parent whose X value is 5, a child process whose X value is 5, and another child process out of the first child process whose X value is 15.

Question 2: Assume that we want to implement a **2-thread lock** that is hand-crafted to be used by ONLY two threads. One possible implementation of this lock is the following:

```
std::atomic<bool> t1_flag(flase);
std::atomic<bool> t2_flag(false);
```

```
Thread 1                              Thread 2

// lock                               // lock
t1_flag = true;                       t2_flag = true;
while(t2_flag == true);               while(t1_flag == true);

// Critical Section                   // Critical Section

// unlock                             // unlock
t1_flag = false;                      t2_flag = false;
```

(a) Does this implementation guarantee mutual exclusion (i.e., the two threads will never be in the critical section simultaneously)? Why or why not?

Yes, it does. In conjunction with the atomic condition variable, which disables optimization of code from the compiler, when each thread locks, it guarantees no thread receives information before the other thread.

(b) Does this implementation guarantee progress (i.e., at least one thread attempting to execute the critical section will succeed)? Why or why not?

No, it does not. There is no equality of which thread executes the critical section first i.e., a queue that prioritizes threads that joined the queue first.

User Id: mame23

Question 3:

(a) In the *Thread Life Cycle* diagram discussed in lectures, some state transitions are missing. Briefly discuss why those transitions are not allowed.
(b) Discuss three different cases at which the thread will be in the *waiting* state.
(c) Is it possible to have an alternative Thread Life Cycle in which the *waiting* state does not exist? Explain using the examples you give in part (b).

Please use the remainder of this page to provide your answer to the question. Give a detailed answer but keep your entire response to a single side of an 8.5x11 page.

(a) The state transitions that are missing include ready to waiting and waiting to running. The ready to waiting transition cannot occur because it is a contradiction. For a thread to be in the ready state means that it is waiting to be scheduled a task. For a thread to be in the waiting state means that it has just completed a task and is joining the end of the thread pool.
(b) A thread can be in the waiting state when the sleep method is called, when the thread blocks on an I/O device, or when the thread unsuccessfully attempts to acquire an object's lock.
(c) The purpose of the waiting state is for any thread that is waiting for another thread to perform a particular action or calculation. If you remove this thread state, you loose the ability to communicate between threads, which decreases productivity and throughput; This is the tradeoff.