User Id: _Zhenyu Wu zhw323_____

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();
      int x = 5;
      if (child == 0) {
            x += 5;
      } else {
            child = fork();
            x += 10;
            if(child) {
                  x += 5;
            }
      }
}
```

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

There are total 3 processes created, and all of them start with x=5. After calling the fork for the 1st time, there will be a parent process(P1) and child process(child 1).The child process(child 1) will increment x by 5, which is 10. Therefore, first x is 10(x1). Then P1 will reach the fork() again to create another parent process(P2) and child process(child 2). Then both process(P2 and child 2) will increment x by 10, which is 15 in both process. Finally, the parent process(P2) will increment x by 5 again which is 20(x2), and the child 2 process will return current x which is 15(x3). Therefore, we have three x values, x1=10, x2=20, x3=15.

User Id: _Zhenyu Wu zhw323_____

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. Both the flag are atomic variable which guarantees that there is no data race between two threads. The Thread 1 can write t1_flag and read t2_flag, and Thread 2 can write t2_flag and read t1_flag. Since there is a spin block in each thread, and it will block each thread from entering the critical section without unlock from another thread. The only way to enter is to write the atomic Boolean variable to true. However, for example, in thread 1, as the t2_flag is write in thread 2, and it is not allowed to be write again in thread 1 at the same time. Therefore, thread 1 will have no chance to write t2_flag and break the spin lock by itself. It has to wait until thread 2 to unlock t2_flag(write t2_flag to false). Same thing happens to thread 2. Therefore, it guarantees mutual exclusion.

(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. If the executing order is following:

1.t1_flag = true

2.t2_flag = true

3.Then each reaches spin lock(while loop)

In this situation, both thread will stuck at spin lock forever for the reason that both flags are atomic variable which means that they can only be written in 1 thread to prevent data race. Thread 1 can't write t2_flag, and thread 2 can't write t1_flag. Therefore, they can't break the spin lock in their own thread, and they will have on chance to execute the critical section.

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.

Missing transition: ready -> init, ready -> finished, ready->waiting, waiting -> running, finished->running

1.For the ready to init, once the thread is created by the sthread.create(lambada), it will create a thread t, and this thread will run the lambada, it will return only after lambada return. Therefore, ready state can't go back to the init state once this thread has been created

2.For the ready to finished. In the ready state, the thread is created and waiting for the scheduler to give it a cpu to run, and each thread is independently scheduled. Once thread is assigned to a cpu to run, it will be interrupt and go back to the ready state from running state to do the context switch to allow other thread to run in order to achieve concurrency of a multithread process. If the thread can by pass running state, which means that once it was assigned to a cpu, it will execute whole content, and other threads need to wait for that, which violate the fairness of concurrency of a multithread process

3.For the ready to waiting. The waiting state means that this thread needs to wait for some results of events(for example I/O). If a thread hasn't been assigned to a cpu by scheduler to run, it shouldn't execute and shouldn't wait for any result of events.

4. For waiting to running. Like I mentioned, once thread needs to wait for some results of events to proceed, it will be transferred from running state to waiting state. During waiting, the scheduler would assign another thread to run to achieve concurrency. If this transition happens from waiting to running, which means that the once current waiting thread receive the result, it will go back to running where there is an another thread running on current cpu, which will ruin the context switch of multithread process.

5. For finished to running. Once the thread has executed the content and returned, it can't not be reloaded to the cpu and run again.

b.

1. When a thread makes a system call of I/O

2. When a main thread needs to wait all the child threads to be returned

3. When one thread needs the result of another thread(for example, one thread is writing a global variable and another will read after writing)

c.

I think it is possible to have a thread life cycle without waiting state if the program doesn't need any result from any event, but it is extremely dangerous and violate concurrency of multithread process. If the program does not require any result from any event, it can terminate without waiting. However, it is still necessary for a main thread to wait all the thread to terminate for the reason that if any joinable thread is be terminated without detaching, it will become a zombie process which will keep consuming and wasting resources that we could have assigned to other processes. However, for program that require results from some events, it is impossible. If one thread makes a system call and needs the result, without waiting state, it will still run on the current cpu and move on without the result for the reason that it will not wait system call to return the result back, which may makes the following execution failed. Lastly, if one thread needs the results of another thread, without waiting process, it is impossible, since current thread will move on without waiting for the target thread to return result.