CSE 347
Real Time System Architectures

Homework - 2

Muharrem Ozan Yeşiller
171044033

Priority Ceiling Protocol Definition

Our problem is to prevent the use of mutexes as a synchronization technique, but to prevent deadlocks from being inevitable in bad designs. For example, let's consider a system, if the lower one of two threads with high and low priority receives a thread and waits for that mutex with high priority, deadlock is inevitable because the only way to do this is if the thread with high priority yields itself. In real-time non-preemptive systems, this is called priority inversion. To solve this problem, the thread that receives the mutex should have a higher priority than the highest thread. In other words, the priority of the high priority thread decreases, the priority of the low priority thread increases. In this project, we developed a mutex class following the Priority Ceiling Protocol principles. This class extends from the std::mutex class and the starting thread must register to this class in order to use the mutex of this class. After this register, priority management of threads is done by this class.

```
namespace gtu
{
   enum LockState { _LOCKED, _NOT_LOCKED };
   class mutex : public std::mutex
   {
       public:
           mutex();
           ~mutex();
           void Register(std::thread&, int);
           void lock();
           void unlock();
       private:
           #if defined(__linux__)
               cpu_set_t cpuset;
           #endif
           static std::forward_list<gtu::mutex *> LockedList;
```

```
            struct MyThreadInfo
            {
                std::thread::native_handle_type ThisThread;
                int Prio;
                MyThreadInfo(std::thread::native_handle_type, int);
                MyThreadInfo(const gtu::mutex::MyThreadInfo&);
            };

            LockState IsLock;
            std::thread::native_handle_type LockMaker;
            std::set<gtu::mutex::MyThreadInfo> Nominees;
            int Ceil;
            void SetSchedThread();
            void SetYieldThread();
            std::thread::native_handle_type GetSelf();
    };
}
```

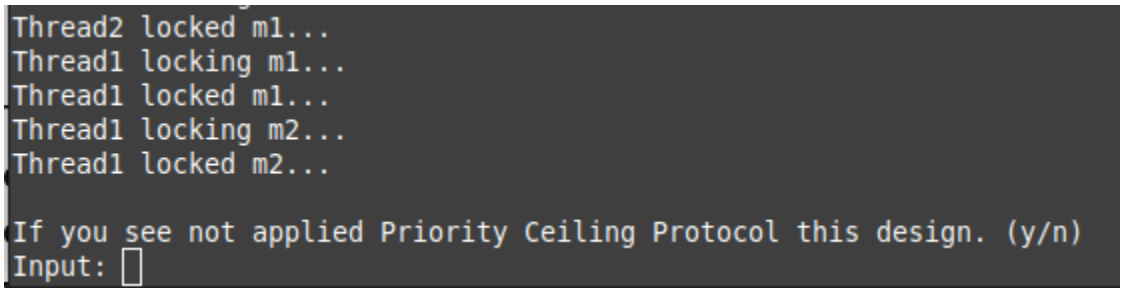
There is no cross way to set the priority issue per operating system. Therefore, this problem has been solved with separate calls as Unix based and Windows based systems.

```
void gtu::mutex::SetSchedThread()
{
    int Zero = 0;
    #if defined(_WIN32)
        SetThreadPriority(GetCurrentThread(),
Nominees.find({GetCurrentThreadId(), Zero}) -> Prio);
    #elif defined(__linux__)
        sched_param Prio;
        Prio.__sched_priority = Nominees.find({pthread_self(), Zero})->Prio;
        if (pthread_setschedparam(pthread_self(), SCHED_FIFO, &Prio)) {
            throw std::system_error(std::error_code(errno,
std::system_category()), "Failed to set priority to orj while unlock!");
        }
    #endif
}
```

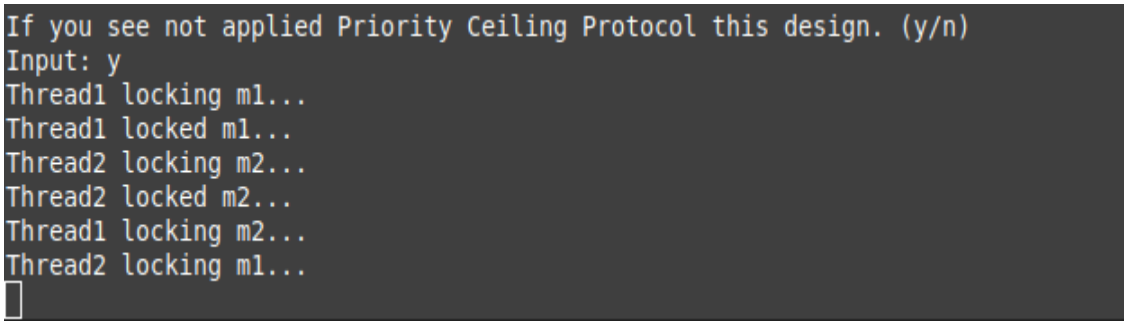
Test Cases

As a test, threads receive mutex m1 and m2. The first thread first takes the m1 mutex, yields and tries to get the m2 mutex. The second thread takes the m2 mutex, yields and tries to get the m1 mutex, in this case deadlock is inevitable. However, when the Priority Ceiling Protocol is applied, there is no deadlock in this case. My program includes both options. These two threads repeat these processes for 50 000 steps and a yes no question is applied to show the user what will happen without the Priority Ceiling Protocol being applied.

No deadlock after 50 000 iteration



If the user type y and enter, user will see deadlock



How to build?

If you are on a unix-based system, you can use Makefile. It is in the project file. If it's Windows, you can use Visual Studio. The important point is that if you are on unix the build code should be SuperUserDO (sudo). Because calls that change priority need this.