CSE 422- Studio 11

1. Selamawit Tegegn(sntegegn@wustl.edu) , Leiquan Pan(lpan22@wustl.edu), yiying Lu([lu.yiying@wustl.edu](mailto:lu.yiying@wustl.edu))
2. Open MP support concurrent execution of functions. See top command result.png

3.

volatile int type=UnlockState;

int expected = LockState;

int desired = UnlockState;

#define LockState 0

#define UnlockState 1

4.

pi@raspberrypi:~ $ ./workload

CPU 3 finished in one critical\_section!

CPU 3 finished!

CPU 0 finished in one critical\_section!

CPU 0 finished!

CPU 2 finished in one critical\_section!

CPU 2 finished!

CPU 1 finished in one critical\_section!

CPU 1 finished!

int lock\_func(volatile int \*type){

int expected = UnlockState;

int desired = LockState;

        while(!\_\_atomic\_compare\_exchange(type, &expected, &desired, FALSE, \_\_ATOMIC\_ACQ\_REL, \_\_ATOMIC\_ACQUIRE)){

                expected = UnlockState;

        }

        return 1;

}

int unlock\_func(volatile int \*type){

int expected = LockState;

int desired = UnlockState;

        if(!\_\_atomic\_compare\_exchange( type, &expected, &desired, FALSE, \_\_ATOMIC\_ACQ\_REL, \_\_ATOMIC\_ACQUIRE)){

                printf("Error: lock is not locked");

expected = LockState;

        }

        return 1;

}

5.

Because the lock function which is a where loop, will make threads wait if they couldn’t get the lock successfully.

6. I don’t think we need to change the define value.

For the lock value which is 0, whenever a thread claims the mutex, it decrements to negative number and then sleep, which is exactly what we want in locked situation.

For the unlock value which is 1, since only one thread at a time can hold the mutex, so when a thread claims the lock, it decrements to 0, and then no one can hold the mutex again unless this thread release it, that’s also what we want for unlocked state.

7.

pi@raspberrypi:~ $ ./workload

CPU 1 finished in one critical\_section!

CPU 1 finished!

CPU 3 finished in one critical\_section!

CPU 3 finished!

CPU 0 finished in one critical\_section!

CPU 0 finished!

CPU 1 finished in one critical\_section!

CPU 1 finished!

See sleep.png spin.png

int lock\_func(volatile int \*type){

int ret\_val;

ret\_val = \_\_atomic\_sub\_fetch(type,1,\_\_ATOMIC\_ACQ\_REL);

while(ret\_val<0){

\_\_atomic\_store\_n( type, -1, \_\_ATOMIC\_RELEASE );

syscall( SYS\_futex, type, FUTEX\_WAIT, -1, NULL );

ret\_val = \_\_atomic\_sub\_fetch(type,1,\_\_ATOMIC\_ACQ\_REL);

}

        return 1;

}

int unlock\_func(volatile int \*type){

int ret\_val;

ret\_val = \_\_atomic\_add\_fetch( type, 1, \_\_ATOMIC\_ACQ\_REL );

        if(ret\_val != 1){

\_\_atomic\_store\_n( type, 1, \_\_ATOMIC\_RELEASE );

syscall( SYS\_futex, type, FUTEX\_WAKE, INT\_MAX );

}

        return 1;

}

8.

Because the lock function use syscall to sleep a thread when the value is negative after decrement the value. Whenever a thread call the unlock function, the value increments and if bigger than zero, the function also use syscall to wake up a process. Most importantly, the value initiate to be 1, so the critical section can be exclusive.

9.

it’s possible, because we can use sleep() function to implement the sleep part, every time the sleep finish, the function can check the value and then determine whether should do the sleep() again.