**Studio 11**

1. **Names of the people:** Sayali Patil
2. **Why is the program occupying all the processors?**

According to my understanding the program occupies all the processors as when we run the program it equally divides the workload to be taken care of on all the processors as we are creating equal number of seconds of work for each processor core.

1. **Declaration statements:**

#define LOCKED 0

#define UNLOCKED 1

volatile int state = UNLOCKED;

1. **Code for lock and unlock functions for spinlock:**

void lock(volatile int \*ptr){

while(!compare(ptr, UNLOCKED, LOCKED));

usleep(1);

printf("Lock acquired by CPU: %d\n", sched\_getcpu());

}

void unlock(volatile int \*ptr){

if(!compare(ptr, LOCKED, UNLOCKED)){

printf("Error: Unlocking failed on CPU: %d\n", sched\_getcpu());

return;

}

printf("Unlocking successful on CPU: %d\n", sched\_getcpu());

}

1. **Why (based on the output you saw) the program only allows one thread at a time into the critical section?**

**Output is:**

pi@raspberrypi\_Sayali\_Patil:~/u $ GOMP\_CPU\_AFFINITY=0,1,2,3 ./spinlock

Lock acquired by CPU: 1

Unlocking successful on CPU: 1

CPU 1 finished!

Lock acquired by CPU: 3

Unlocking successful on CPU: 3

CPU 3 finished!

Lock acquired by CPU: 2

Unlocking successful on CPU: 2

CPU 2 finished!

Lock acquired by CPU: 0

Unlocking successful on CPU: 0

CPU 0 finished!

As critical section of the program is what we are trying to protect by acquiring lock in the beginning of the section and releasing it at the end of the section to avoid thread blocking or data race conditions. As the lock can be acquired by the second thread only when its released by the first one, only one thread can enter the critical section at a time.

1. I did not need to change the define values in my code because for the spinlock code I had initialized lock to 0 and unlock to 1 which was exactly same to the values expected in case of sleeplock.
2. **And**
3. **Code snippet:**

void lock(volatile int \*ptr){

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

while(ret\_val < 0){

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

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

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

}

usleep(1);

printf("Lock acquired by CPU: %d\n", sched\_getcpu());

}

void unlock(volatile int \*ptr){

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

if(ret\_val != 1){

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

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

}

printf("Lock released by CPU: %d\n", sched\_getcpu());

}

1. **Explain why you think only one thread at a time can access the critical section.**

**Output is:**

Lock acquired by CPU: 2

Lock released by CPU: 2

CPU 2 finished!

Lock acquired by CPU: 3

Lock released by CPU: 3

CPU 3 finished!

Lock acquired by CPU: 1

Lock released by CPU: 1

CPU 1 finished!

Lock acquired by CPU: 0

Lock released by CPU: 0

CPU 0 finished!

The reason is the same one as explained in point 5.

1. **Trace:**

Screenshots for both are attached.

It won’t be possible to implement sleeplock in userspace entirely without system calls as linux threads make use of the kernel for the implementation process of sleeplocks.