COL331 : HW4

Sumit Singh
2016PH10575

A) Assignment : Spin-locking :-

Q: Adding sti() just after acquire() and cli() just before release in 'iderw' function in ide.c.
Why does the kernel panic?

Ans:- The 'iderw' function acquires the idelock and starts executing, thereby disabling the interrupts (using pushcli()). Since we add sti() just after it, the interrupts get enabled. Before the 'iderw' function could release 'idelock', interrupt occurs and the interrupt handler 'ideintr' gets called. 'ideintr' tries to acquire the 'idelock' which is already acquired by 'iderw' and so the kernel panics. ~~The interrupt handler is supposed to be able~~ to ~~acqu~~

B) Q Adding sti() and cli() in filealloc() in file.c.
Why doesn't the kernel panic in this case?

Ans:- The critical section in case of filealloc() is small compared to the ~~durati~~ period of the timer interrupt. Thus it is very unlikely for the interrupt to occur while filealloc function's critical section executes. On the other hand, the critical section in iderw() involves a potential disk access, so it is comparable to the timer interrupt period.

Q. Why does release() clear lk→pcs[0] and lk→cpu before clearing lk→locked? Why not wait until after?

Ans:- There might be a number of threads trying to acquire the lock at the same time.

Let's say lk→locked is cleared before lk→cpu. As soon as lk→locked is cleared, other threads will race to acquire this lock. ~~This~~ The race condition will be b/w the current thread (which is trying to release the lock) and a new thread (which has been spinning until to acquire it). The new threads might thus acquire the lock while the lk→cpu field would still indicate the previous cpu. This leads to an inconsistent state.

Placing lk→cpu and lk→pcs[0] before lk→locked, thus ensures that the current thread has cleared the lock data completely before any other thread can acquire it.

→ Uniprocessor locking:

```
lock(L) {
    cli();
    while(L == 0) continue;
    L=0;
    sti();
}
```

```
unlock(L)
{ L= 1; }
```

Q:) Does this implementation work on a uniprocessor? If not why not?

Ans:- ~~This implementation can lead to deadlock.~~

This implementation can lead to deadlocks.

Say, thread 1 acquires (locks) L and gets preempted before it could unlock L. Thread 2 now gets to run which tries to lock L again. So, it will disable interrupts and spin for the value of L to become 1, which will never happen because interrupts (hence, preemption) are disabled. So thread 2 spins forever.

→
```
lock (L) {
    int acquired = 0;
    while (!acquired) {
        cli();
        if (L = 1) {
            acquired = 1;
            L = 0
        }
        sti();
    }
}

unlock (L) {
    L = 1;
}
```

Ans:- This implementation works on uniprocessor.

There is a chance for the thread which is trying to acquire the lock to get preempted (unlike the previous case where the thread just spun after disabling interrupts).

B.) Assignment: Sleep and Wakeup

Q) Both producer (pcqwrite) and consumer (pcqread) are sleeping on the same channel q. Is this correct? why or why not? Should they sleep on different channels?

Ans:- This implementation is correct.

When the producer calls wakeup (q), all the threads waiting on q (i.e. both producer and consumer threads) are woken up. But the ~~consumer~~ producer threads which are woken up, will check the condition again and will find $q \rightarrow ptr \,!= 0$ and so they will go to sleep again. Among the consumer threads, only one thread will be able to acquire lock and proceed.

Yes, unrelated parts of the code can also wakeup a consumer thread.