COMSW4118 - 11-1 10/10/2017

- From last time:

```
read_lock(L) {
    lock(x);
    count++;
    if (count == 1)
        lock(L);
    unlock(x)
}
```

grab L when first reader shows up

```
read_unlock(L) {
    lock(x);
    count--;
    if (count == 0)
        unlock(L);
    unlock(x);
}
```

Last reader release L

```
write_lock(L) {
    lock(L);
}
```

```
write_unlock(L) {
    unlock(L);
}
```

NOTE: thread that locked L as reader does not have to be the same one that locked it.

- Problem: if you have lots of readers they will still be serialized in read_lock. this is a performance bottleneck.

▶ **Read/Copy/Update (RCU)** can be used in some cases to avoid reader lock contention.

- Requirement is that the writer update needs to be atomic - ex. if you update a structure, rather than going field-by-field, just swap pointer from old to new copy.
- Readers may only see the old value = before update
  Then only some readers may see the new value = update happens
  Finally, all readers see new value = update is complete
  I.e. it does not guarantee ordering.
- Note that this does not resolve race conditions for the final value, but the values that readers see are consistent (either old or new, but correct)

Example: (slides link on course web page for RCU)

```
typedef struct Elements{
    int key;
    int value;
    struct Elements *next;
} Element;
```

```
class RCUList {
  private:
    RCULock rcuLock;
    Element *head;
  public:
    bool search(int key, int *value);
    void insert(Element *item, value);
    bool remove(int key);
};
```

```
bool RCUList::search(int key, int *valuep) {
    bool result = FALSE;
    Element *current;
    rcuLock.readLock();
    current = head;
    for(current = head; current != NULL; current = current->next) {
        if(current->key == key) {
            *valuep = current->value; result = TRUE; break;
        }
    }
    rcuLock.readUnlock();
    return result;
}
```

RCU read lock has the same semantics as RWLock read lock.

```
void RCUList::Insert(int key, int value) {
    Element *item;
    rcuLock.writeLock()  <- one writer at a time
    item = (Element *) malloc(sizeof(Element));
    item->key = key; item->value = value; item->next = head;
    rcuLock.publish(&head, item))  <- this is the atomic update (after this a search may or may not contain the new item)
    rcuLock.writeUnlock();  <- allow other writes to proceed
    rcuLock.synchronize();  <- wait until no reader has old version (any search will find new item)
};

bool RCUList::remove(int key) {
    bool found = FALSE;
    Element *prev, *current;
    rcuLock.WriteLock();  <- one writer at a time
    for(prev=NULL; current=head; current!=NULL; prev=current, current=current->next)
        if (current->key == key) {
            found = TRUE;
            if (prev == NULL)
                rcuLock.publish(&head, current->next)  <- publish update to readers
            else rcuLock.publish(&(prev->next), current->next);
            break;
        }
    rcuLock.writeUnlock();  <- allow other writers to proceed
    rcuLock.synchronize();  <- wait until no reader has old version to free!
    if(found) {
        rcuLock.synchronize();
        free(current);
    }
    return found;
};
```

RCU can only be used if you can publish your updates as one change like: rcuLock.publish()

RCU Lock implementation:

FOR MULTI-PROCESSORS!

NOTE: No locking anymore, just disable/enable interrupts.

```
void RCULock::readLock() {
    disableInterrupts();
}

void RCULock::readUnlock() {
    enableInterrupts();
}
```

<- disable/enable is only for the current CPU

<- PRIVILEGED!

So, you should not sleep b/n rculock.readLock and rculock.readUnlock because you will be sleeping with interrupts disabled and that is bad!

```
void RCULock::writeLock() {
    writerSpin.acquire();
}

void RCULock::writeUnlock() {
    writerSpin.release();
}
```

← Plain lock

NOTE: In the Linux kernel, there is no writeLock implemented because any lock would be OK to use to synchronize writers.

```
void RCULock::publish(void **pp1, void *p2) {
    memory_barrier();
    *pp1 = p2;
    memory_barrier();
}
```

to make sure any instructions before the section between need to be executed before and any instructions after need to happen after. (to resolve instruction reordering issues).

```
void RCULock::synchronize() {
    int p, c;
    globalSpin.acquire();
    c = ++globalCounter;
    globalSpin.release();
    FOREACH_PROCESSOR(p) {
        while ((PER_PROC_VAR(quiescentCount, p) - c) < 0) {
            sleep(10); ← release CPU for 10 ms.
        }
    }
}
```

(meaning replaced all threads that were previously on each CPU)

Once the OS has switched something else on all CPUs, then all the readers would've been done reading (because they disable interrupts while reading, so they can't be switched and when they are switched, this means they are done!

Called by scheduler

```
void RCULock::QuiescentState() {
    memory_barrier();
    PER_PROC_VAR(quiescentCount) = globalCounter;
    memory_barrier();
}
```

NOTE: This in essence counts how many CPUs got to executing synchronize() - if all CPUs got to there, then all would have done reading!

Scheduler calls QuiescentState() to get an always increasing timestamp (as an int counter) when a process is scheduled per CPU. Synchronize gets/increments that counter when it is called, then waits for all CPUs to get past it, thus making sure scheduler was called on each CPU!!!

COMSW4118-11-4 10/10/2017

RCU avoids the reader contention at the expense of some added complexity

Linux has locks throughout the kernel, since it is designed to be a multi-processor and multi-thread kernel:

- **Linux kernel is reentrant** = you can have multiple threads/processes running in the kernel code

If you are not re-entrant, you can't execute kernel code in parallel on two different CPUs.

Reentrant kernel (2 types)
- **Non-Preemptive** = a thread or process in the kernel needs to voluntarily yield to give the CPU
- **Preemptive** = a thread/process can be forcefully preempted while executing kernel code

Non-preemptive kernels have the luxury of cleaning up/leaving the kernel in consistent state before yielding.

Preemptive kernels are more difficult to program since a thread can be preempted at any time certain portions of the code can be executed w/ preemption disabled.

(!) Example: lock(); ..... unlock(); in the kernel are usually meant to bracket messy code, so a common approach is to disable preemption in that section of the code.
Also, this is why you don't want to ~~voluntarily~~ sleep while holding a spinlock in Linux - you would have disabled preemption and then voluntarily preempted yourself (with the preemption still disabled).

- **Linux Synchronization primitives:**
  - Atomic Types = set of functions/types; include/asm-generic/atomic.h
  - Spinlocks = spin_lock(), spin_unlock(); example in kernel/timer.c
    spin_lock_init(), spinlock_t, spin_lock_irq ← grab lock & disable interrupts
    spin_lock_irqsave & spin_lock_irqrestore (use spin_unlock_irq to release & enable) to save/restore interrupts consistently (i.e. restore the status before, not just enable)
  - Reader/Writer Lock
  - RCU
  - Semaphores

Spinlock can be done in 2 ways:
- Regular, as described so far
- Interrupted - if you have a lock and are interrupted and the handler tries to get the same lock (it will be stuck forever), then use spin_lock_irq()

(!) Basically, any time you are manipulating something that can be modified by an interrupt.
→ to find out, manually check source