CIS520 – Operating Systems
Handout 5
Implementing Synchronization Operations

- How do we implement synchronization operations like locks? Can build synchronization operations out of reads and writes - see the bakery algorithm in Section 6.2.2 in OSC. But, this is slow and cumbersome to use. So, most machines have hardware support for synchronization - they provide synchronization instructions.
- On a uniprocessor, the only thing that will make multiple instruction sequences not atomic is interrupts. So, if want to do a critical section, turn off interrupts before the critical section and turn on interrupts after the critical section. Guaranteed atomicity. It is also fairly efficient. Early versions of Unix did this.
- Why not just use turning off interrupts? Two main disadvantages: can't use in a multiprocessor, and can't use directly from user program for synchronization.
- Test-And-Set. The test and set instruction atomically checks if a memory location is zero, and if so, sets the memory location to 1. If the memory location is 1, it does nothing. It returns the old value of the memory location. You can use test and set to implement locks as follows:
  - The lock state is implemented by a memory location. The location is 0 if the lock is unlocked and 1 if the lock is locked.
  - The lock operation is implemented as:

    ```
    while (test-and-set(l) == 1);
    ```

  - The unlock operation is implemented as: *l = 0;

  The problem with this implementation is busy-waiting. What if one thread already has the lock, and another thread wants to acquire the lock? The acquiring thread will spin until the thread that already has the lock unlocks it.
- What if the threads are running on a uniprocessor? How long will the acquiring thread spin? Until it expires its quantum and thread that will unlock the lock runs. So on a uniprocessor, if can't get the thread the first time, should just suspend. So, lock acquisition looks like this:

  ```
  while (test-and-set(l) == 1) {
    currentThread->Yield();
  }
  ```

  Can make it even better by having a queue lock that queues up the waiting threads and gives the lock to the first thread in the queue. So, threads never try to acquire lock more than once.
- On a multiprocessor, it is less clear. Process that will unlock the lock may be running on another processor. Maybe should spin just a little while, in hopes that other process will release lock. To evaluate spinning and suspending strategies, need to come up with a cost for each suspension algorithm. The cost is the amount of CPU time the algorithm uses to acquire a lock.
- There are three components of the cost: spinning, suspending and resuming. What is the cost of spinning? Waste the CPU for the spin time. What is cost of suspending and resuming? Amount of CPU time it takes to suspend the thread and restart it when the thread acquires the lock.

- Each lock acquisition algorithm spins for a while, then suspends if it didn't get the lock. The optimal algorithm is as follows:
  - If the lock will be free in less than the suspend and resume time, spin until acquire the lock.
  - If the lock will be free in more than the suspend and resume time, suspend immediately.

  Obviously, cannot implement this algorithm - it requires knowledge of the future, which we do not in general have.

- How do we evaluate practical algorithms - algorithms that spin for a while, then suspend. Well, we compare them with the optimal algorithm in the worst case for the practical algorithm. What is the worst case for any practical algorithm relative to the optimal algorithm? When the lock become free just after the practical algorithm stops spinning.

- What is worst-case cost of algorithm that spins for the suspend and resume time, then suspends? (Will call this the SR algorithm). Two times the suspend and resume time. The worst case is when the lock is unlocked just after the thread starts the suspend. The optimal algorithm just spins until the lock is unlocked, taking the suspend and resume time to acquire the lock. The SR algorithm costs twice the suspend and resume time -it first spins for the suspend and resume time, then suspends, then gets the lock, then resumes.

- What about other algorithms that spin for a different fixed amount of time then block? Are all worse than the SR algorithm.
  - If spin for less than suspend and resume time then suspend (call this the LT-SR algorithm), worst case is when lock becomes free just after start the suspend. In this case the the algorithm will cost spinning time plus suspend and resume time. The SR algorithm will just cost the spinning time.
  - If spin for greater than suspend and resume time then suspend (call this the GR-SR algorithm), worst case is again when lock becomes free just after start the suspend. In this case the SR algorithm will also suspend and resume, but it will spin for less time than the GT-SR algorithm

  Of course, in practice locks may not exhibit worst case behavior, so best algorithm depends on locking and unlocking patterns actually observed.

- Here is the SR algorithm. Again, can be improved with use of queueing locks.

```
notDone = test-and-set(l);
if (!notDone) return;
start = readClock();
while (notDone) {
  stop = readClock();
  if (stop - start >= suspendAndResumeTime) {
    currentThread->Yield();
    start = readClock();
  }
  notDone = test-and-set(l);
}
```

- There is an orthogonal issue. test-and-set instruction typically consumes bus resources every time. But a load instruction caches the data. Subsequent loads come out of cache and never hit the bus. So, can do something like this for inital algorithm:

```
while (1) {
  if !test-and-set(l) break;
  while (*l == 1);
}
```

- Are other instructions that can be used to implement spin locks - swap instruction, for example.

- On modern RISC machines, test-and-set and swap may cause implementation headaches. Would rather do something that fits into load/store nature of architecture. So, have a non-blocking abstraction: Load Linked(LL)/Store Conditional(SC).
- Semantics of LL: Load memory location into register and mark it as loaded by this processor. A memory location can be marked as loaded by more than one processor.
- Semantics of SC: if the memory location is marked as loaded by this processor, store the new value and remove all marks from the memory location. Otherwise, don't perform the store. Return whether or not the store succeeded.
- Here is how to use LL/SC to implement the lock operation:

```
while (1) {
  LL  r1, lock
  if (r1 == 0) {
    LI r2, 1
    if (SC r2, lock) break;
  }
}
```

  Unlock operation is the same as before.

- Can also use LL/SC to implement some operations (like increment) directly. People have built up a whole bunch of theory dealing with the difference in power between stuff like LL/SC and test-and-set.

```
while (1) {
  LL   r1, lock
  ADDI r1, 1, r1
  if (SC r2, lock) break;
}
```

- Note that the increment operation is non-blocking. If two threads start to perform the increment at the same time, neither will block - both will complete the add and only one will successfully perform the SC. The other will retry. So, it eliminates problems with locking like: one thread acquires locks and dies, or one thread acquires locks and is suspended for a long time, preventing other threads that need to acquire the lock from proceeding.