IT308: Operating Systems

Implementing Locks

Locks and Scheduling

- Threads as entities scheduled by OS
- Locks provide some minimal amount of control over scheduling (back) to programmers
    - e.g., guarantee that no more than a single thread can ever be active within that code (critical section)
- In a previous course, we studied how to use Pthread locks

Locks and Scheduling

- Threads as entities scheduled by OS
- Locks provide some minimal amount of control over scheduling (back) to programmers
  - e.g., guarantee that no more than a single thread can ever be active within that code (critical section)
- In a previous course, we studied how to use Pthread locks
  - But how should we build a lock?

Building a Lock

- Over the years, different hardware primitives have been added to the instruction sets of various computer architectures
- Will study how to use them in order to build a lock
- Will study how OS gets involved to complete the picture and enable us to build a sophisticated locking library

Requirements for Locks

- Correctness
  - Only one thread in critical section at a time

Requirements for Locks

- Correctness
  - Only one thread in critical section at a time
- Fairness
  - Each thread gets a fair chance at acquiring the lock

Requirements for Locks

- Correctness
  - Only one thread in critical section at a time
- Fairness
  - Each thread gets a fair chance at acquiring the lock
- Performance
  - Time overhead for a lock without and with contentions (possibly on multiple CPUs)

Mutual Exclusion and Interrupts

- Mutual exclusion
  - Want to prevent another thread from executing while current thread is in critical section
- Solution: disable interrupts for critical sections

Mutual Exclusion and Interrupts

- Mutual exclusion
  - Want to prevent another thread from executing while current thread is in critical section
- Solution: disable interrupts for critical sections
  - before entering critical section, disable interrupts
  - after exiting critical section, reenable interrupts
  - This enforces mutual exclusion

Lock Implementation 1

```
void lock ()
  {
    DisableInterrupt();   // special hardware instruction
  }

  void unlock()
  {
    EnableInterrupt();    // special hardware instruction
  }
```

- Negatives?

Lock Implementation 1

```
void lock ()
  {
    DisableInterrupt();  // special hardware instruction
  }

  void unlock()
  {
    EnableInterrupt();   // special hardware instruction
  }
```

- Negatives?
  - allow calling thread to perform privileged operations – “trust?”

Lock Implementation 1

```
void lock ()
  {
    DisableInterrupt();   // special hardware instruction
  }

  void unlock()
  {
    EnableInterrupt();    // special hardware instruction
  }
```

- Negatives?
  - allow calling thread to perform privileged operations – "trust?"
  - disables multiprogramming even if another thread is NOT interested in critical section

Lock Implementation 2

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
  mutex->flag = 0; // 0 -> lock is available, 1 -> held
}
void lock(lock_t *mutex) {
  while (mutex->flag == 1) // TEST the flag
    ; // spin-wait (do nothing)
  mutex->flag = 1;         // now SET it!
}
void unlock(lock_t *mutex) {
  mutex->flag = 0;
}
```

What problems does this solution have?

Malicious Scheduler

- Pretend being a malicious scheduler, one that interrupts threads at the most inopportune of times in order to foil synchronization primitives
- Although the exact sequence of interrupts may be improbable, it is possible, and that is all we need to demonstrate that a particular approach does not work

Trace: No Mutual Exclusion

Thread 1	**Thread 2**
call `lock()`	
while (flag == 1) false	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1) false
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Atomic Test-And-Set

- machine instruction used to synchronize threads
  - atomically sets the value of a specified memory location and places that memory location's old value into a register
- A C description of the machine instruction

```
int TestAndSet(int *old_ptr, int new) {
  int old = *old_ptr; // fetch old value at old_ptr
  *old_ptr = new;     // store 'new' into old_ptr
  return old;         // return the old value
}
```

- . . . all this is done atomically

Lock Implementation 3

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *lock) {
  lock->flag = 0; // 0 -> lock is available, 1 -> held
}
void lock(lock_t *lock) {
  while (TestAndSet(&lock->flag, 1) == 1) // TEST the flag
    ; // spin-wait (do nothing)
}
void unlock(lock_t *lock) {
  lock->flag = 0;
}
```

- As long as the lock is held by another thread, TestAndSet() will repeatedly return 1, and thus the calling thread will spin-wait

Drawbacks of Spin Lock

- Starvation is possible
  - a waiting thread may wait (spin) forever under contention
- Spinning can be costly
  - assume thread holding lock is preempted within a critical section
  - subsequently if N-1 waiting threads are scheduled by the OS, then each of those will waste CPU cycles for the duration of a time slice before giving up