Locks

Locks: The Basic Idea

- To implement a critical section
- A lock variable must be declared
- A lock variable holds the state of the lock
  - Available (unlocked, free)
  - Acquired (locked, held)
- Exactly one thread holds the lock, owner
- Once the owner of the lock calls unlock(), if there are waiting threads (stuck in lock()), one of them will (eventually) notice (or be informed of) this change of the lock's state, acquire the lock, and enter the critical section.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

Evaluating Locks

- **Mutual exclusion**
  - whether the lock does its basic task
- **Fairness**
  - Does each thread contending for the lock get a fair shot at acquiring it once it is free?
  - No starvation
- **Performance**
  - time overheads added by using the lock
  - In the case of single thread (no contention)
  - In the case of multiple contending threads on a single CPU
  - In the case of multiple threads on multiple CPUs

Controlling Interrupts

- The earliest solutions used to provide mutual exclusion was to disable interrupts for critical sections (single-processor system)

  + Simple

  - Requires to allow any calling thread to perform a privileged operation (interrupt on/off)

  - Does not work on multiprocessors

  - Lost interrupts

  - Inefficient: code that masks or unmasks interrupts tends to be executed slowly by modern CPUs
- OS itself will use interrupt masking to guarantee atomicity when accessing its own data structures

```
void lock() {
        DisableInterrupts();
}
void unlock() {
        EnableInterrupts();
}
```

A Failed Attempt: Just Using Loads/Stores

- Build a simple lock by using a single flag variable
- Correctness problem
- Performance problem
  - spin-waiting
  - high on a uniprocessor

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

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

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;
}
```

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

Successful but Old Attempt (w/o HW support)

- **Dekker's algorithm ('68)**
- **Peterson's algorithm ('81)**
- **Useless now**
  - H/W support for lock and relaxed memory consistency models

P_0 enters CS only if
either flag[1] = 0 or turn = 0

```
int flag[2];
int turn;

void init() {
        flag[0] = flag[1] = 0; // 1->thread wants to grab lock
        turn = 0; // whose turn? (thread 0 or 1?)
}
void lock() {
        flag[self] = 1; // self: thread ID of caller
        turn = 1 - self; // make it other thread's turn
        while ((flag[1-self] == 1) && (turn == 1 - self))
                ; // spin-wait
}
void unlock() {
        flag[self] = 0; // simply undo your intent
}
```

Building Working Spin Locks with test-and-set

- We need hardware support!
- test-and-set instruction (atomic exchange)
  - returns the old value pointed to by the old_ptr (test), and simultaneously updates said value to new (set).
  - this sequence of operations is performed atomically (uninterruptable)

```
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
}
```

Building Working Spin Locks with test-and-set

- As long as the lock is held by another thread, TestAndSet() will repeatedly return 1, and thus this thread will spin and spin until the lock is finally released.
- By making both the **test** and **set** a single atomic operation, we ensure that only one thread acquires the lock.
- To work correctly on a single processor, it requires a **preemptive scheduler**

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

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Evaluating Spin Locks

- **Correctness**? Yes
  - allows a single thread to enter CS at a time
- **Fairness**? Bad
  - A thread spinning may spin forever
- **Performance**?
  - Single CPU case: high overhead
    - If the thread holding the lock is pre-empted within a critical section, other scheduled threads try to acquire the lock.
    - Each of those threads will spin for the duration of a time slice before giving up the CPU, a waste of CPU cycles.
  - Multiple CPUs: work reasonably well
    - Thread A on CPU 1 and Thread B on CPU 2, both contending for a lock.
    - Spinning to wait for a lock held on another processor doesn't waste many cycles in this case, and thus can be effective.

Compare-And-Swap (compare-and-exchange)

- Test whether the value at the address specified by ptr is equal to expected
  - If so, update the memory location pointed to by ptr with the new value.
  - If not, do nothing
- Similar behavior but more powerful instruction than test-and-set
  - Useful for lock-free synchronization

```
int CompareAndSwap(int *ptr, int expected, int new) {
        int actual = *ptr;
        if (actual == expected)
        *ptr = new;
        return actual;
}
```

```
void lock(lock_t *lock) {
        while (CompareAndSwap(&lock->flag, 0, 1) == 1)
                ; // spin
}
```

Load-Linked and Store-Conditional

- load-linked and store-conditional instructions can be used in tandem to build locks and other concurrent structures
- MIPS, Alpha, PowerPC, ARM
- store-conditional only succeeds if no intervening store to the address has taken place

```
int LoadLinked(int *ptr) {
        return *ptr;
}

int StoreConditional(int *ptr, int value) {
        if (no one has updated *ptr since the LoadLinked to this address) {
                *ptr = value;
                return 1; // success!
        } else
                return 0; // failed to update
}
```

Load-Linked and Store-Conditional

```
void lock(lock_t *lock) {
        while (1) {
                while (LoadLinked(&lock->flag) == 1)
                    ; // spin until it's zero
                if (StoreConditional(&lock->flag, 1) == 1)
                    return; // if set-it-to-1 was a success: all done
                // otherwise: try it all over again
        }
}

void unlock(lock_t *lock) {
        lock->flag = 0;
}
```

or

```
void lock(lock_t *lock) {
    while (LoadLinked(&lock->flag)||!StoreConditional(&lock->flag, 1))
                ; // spin
}
```

Fetch-And-Add

```
int FetchAndAdd(int *ptr) {
        int old = *ptr;
        *ptr = old + 1;
        return old;
}
```

	Ticket	myturn	Turn
Initial	0		0
T1 calls lock	1	0 ==	0
T1 enters CS	1		0
T2 calls lock	2	1 ≠	0
T3 calls lock	3	2 ≠	0
T1 calls unlock	3		1
T2 enters CS	3		1
T2 calls unlock	3		2
T3 enters CS	3		2
T3 calls unlock	3		3

```
typedef struct __lock_t {
        int ticket;
        int turn;
} lock_t;

void lock_init(lock_t *lock) {
        lock->ticket = 0;
        lock->turn = 0;
}
void lock(lock_t *lock) {
   int myturn = FetchAndAdd(&lock->ticket);
   while (lock->turn != myturn)
        ; // spin
}

void unlock(lock_t *lock) {
   lock->turn = lock->turn + 1;
}
```

Once a thread is assigned its ticket value, it will be scheduled at some point in the future

Ticket locks

Too Much Spinning

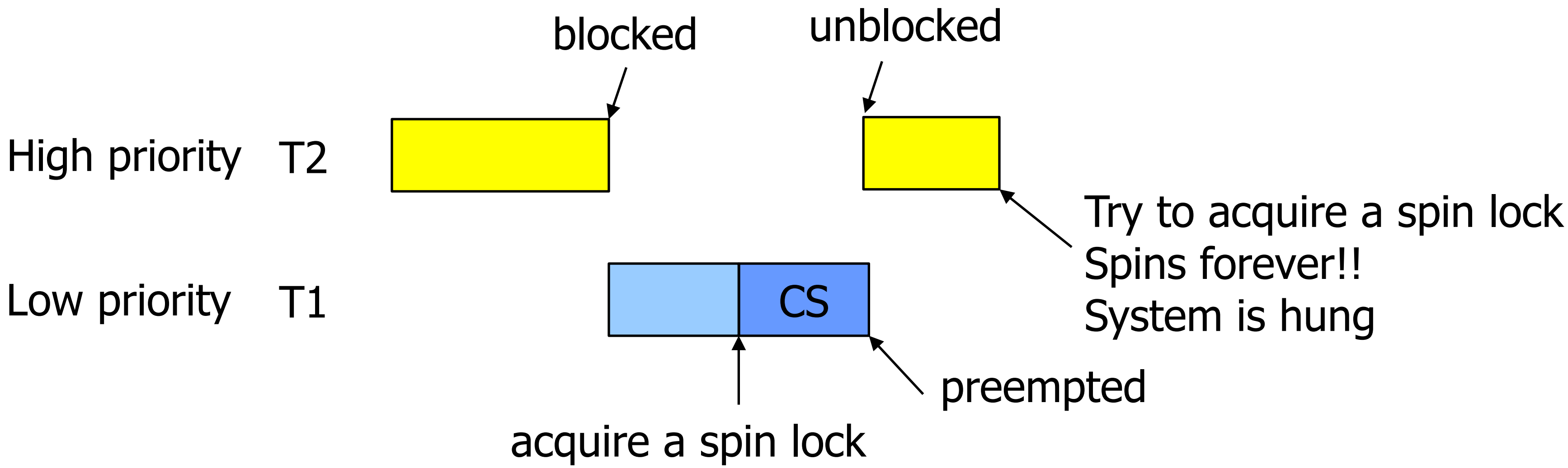
- Hardware-based locks are simple & work
- But, can be quite inefficient
- N threads on a single processor
  - Thread 0 is in a critical section w/ a lock, and unfortunately gets interrupted
  - Thread 1 is scheduled and tries to acquire the lock, and it begins to spin.
  - Thread 2 is scheduled and tries to acquire the lock, and it begins to spin.
  - …
  - N-1 time slices may be wasted

waiting for the interrupted (lock-holding) thread to be run again

- Hardware support alone cannot solve the problem.
- We'll need OS support too!

More Reason to Avoid Spinning

- **Priority-Driven Scheduling**



How about if just avoid the use of spin locks?

A Simple Approach: Just Yield

- **yield** system call moves the caller from the running state to the ready state, deschedules itself
- High context switch cost
  - each thread calling lock() will execute run-and-yield pattern before the thread holding the lock gets to run again
  - High context switch cost
- Starvation problem
  - A thread may get caught in an endless yield loop while other threads repeatedly enter and exit the critical section.

```
void init() {
        flag = 0;
}
void lock() {
        while (TestAndSet(&flag, 1) == 1)
                yield(); // give up the CPU
}
void unlock() {
        flag = 0;
}
```

Spinlock: No Context Switching

Using Queues: Sleeping Instead Of Spinning

- Too much left to chance
  - The schedule determines who runs next; if it makes a bad choice – yield immediately or sleep
  - Let’s get some control over who gets to acquire the lock next
- Need a queue to keep track of which threads are waiting to acquire the lock.
- A lock puts a caller to sleep if it tries to acquire a held lock and wakes it when the lock is free.

Lock With Queues, Test-and-set, Yield, And Wakeup

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
}
```

Guard is a spin-lock around the flag and queue manipulations

the time spent spinning is quite limited

Isn't that a race condition?

Guard lock must be released before park()

put a calling thread to sleep

wake a waiting thread; pass the lock directly to the next thread acquiring it; flag is not set to 0 in-between

wakeup/waiting race

- A thread (T0) will be about to park, assuming that it should sleep until the lock is no longer held.
- A switch at that time to the thread (T1) holding the lock.
- T1 releases the lock.
- The subsequent park by T0 would then sleep forever.

```
queue_add(m->q, gettid());
setpark(); // new code
m->guard = 0;
park();
```

- Solaris solution
  - setpark()
  - a thread can indicate it is *about to* park
  - After this, if the thread is interrupted and another calls unpark before the park is called, parks returns immediately
- Another solution
  - pass the guard into the kernel.
  - kernel could take precautions to atomically release the lock and dequeue the running thread.

Different OS, Different Support

- Linux: **futex** (Fast Userspace MuTEX)
  - each futex has associated with it a specific physical memory location, as well as a per-futex in-kernel queue.
  - Provides atomic compare-and-block operation
  - futex is a lower-level construct
  - Used as building blocks for mutex, condition variables, semaphores
  - futex_wait(address,expected)
    - puts the calling thread to sleep if mem[address] = expected
    - If it is not equal, the call returns immediately.
  - futex_wake(address)
    - wakes one thread that is waiting on the queue.
  - Use a 32-bit integer
    - The leftmost bit (the +/- sign) tracks the lock state
      - 0: free, 1: locked
    - Remaining 31 bits: the number of waiters on the lock

Linux-based Futex Locks

```
void mutex_lock (int *mutex) {
  int v;
  /* Bit 31 was clear, we got the mutex (this is the fastpath)  */
  if (atomic_bit_test_set (mutex, 31) == 0)
    return;
  atomic_increment (mutex);
  while (1) {
      if (atomic_bit_test_set (mutex, 31) == 0) {
          atomic_decrement (mutex);
          return;
      }
      /* We have to wait now. First make sure the futex value
         we are monitoring is truly negative (i.e. locked). */
      v = *mutex;
      if (v >= 0)
        continue;
      futex_wait (mutex, v);
  }
}

void mutex_unlock (int *mutex) {
  /* Adding 0x80000000 to the counter results in 0 if and only if
    there are not other interested threads */
  if (atomic_add_zero (mutex, 0x80000000))
    return;

  /* There are other threads waiting for this mutex,
     wake one of them up.  */
  futex_wake (mutex);
}
```

Two-Phase Locks

- Spinning can be useful, particularly if the lock is about to be released
- **Two-Phase Lock: hybrid between spin-locks and yielding**
- First phase
  - the lock spins for a while, hoping that it can acquire the lock.
  - if the lock is not acquired, a second phase is entered
- Second phase
  - the caller is put to sleep
  - only woken up when the lock becomes free later.