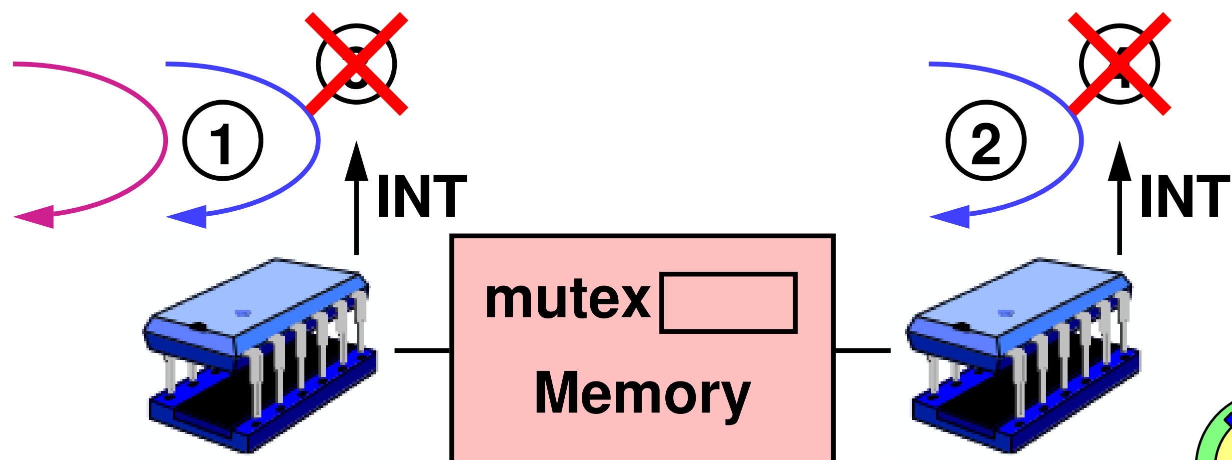
5.1 Threads Implementations

- Strategies
- A Simple Thread Implementation
- ***Multiple Processors***



Straight-threads - Multiple Processors

- `thread_switch()` is no longer sufficient
  - it's meant for uniprocessor

```
void thread_switch( ) {
  thread_t NextThread, OldCurrent;
  NextThread = dequeue(RunQueue);
  OldCurrent = CurrentThread;
  CurrentThread = NextThread;
  swapcontext(&OldCurrent->context,
              &NextThread->context);
}
```

- Simple approach
  - run on each *processor* an *idle thread*

```
void idle_thread() {
  while(1) {
    enqueue(runqueue, CurrentThread)
    thread_switch()
  }
}
```

`yield()`

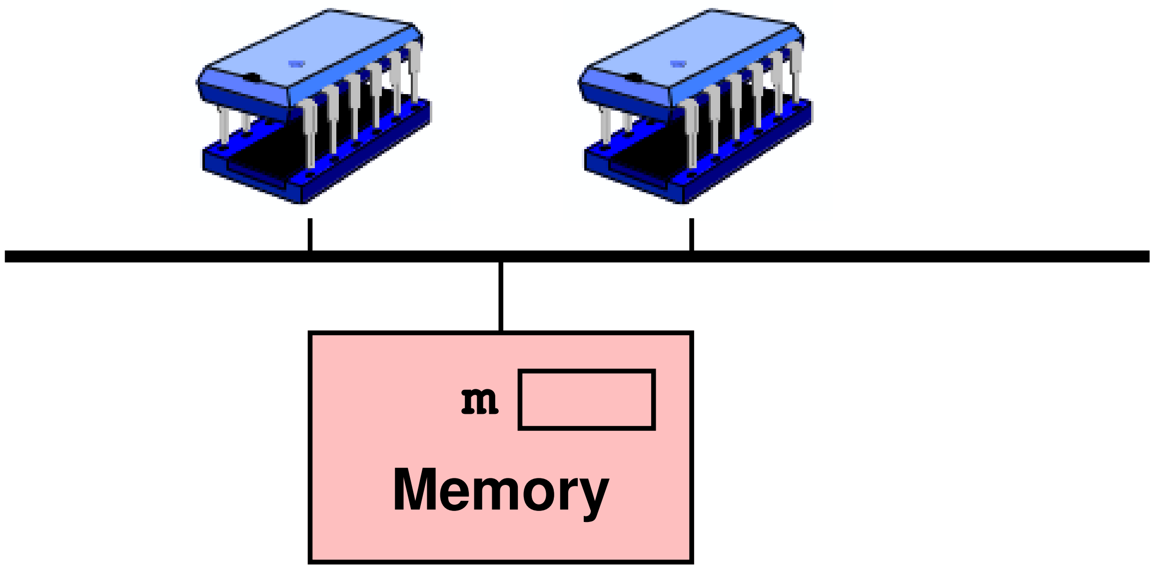
  - this thread never blocks, so there is always something to run to avoid boundary condition *(although this is busy-waiting)*
  - code is incomplete (because `thread_switch()` is incomplete, the way it was presented here)
- normal threads join the RunQueue when ready

Straight-threads - Multiple Processors

- When there are multiple processors, the difficulty lies in locking

```
if (!m->locked) {
  m->locked = 1;
}
```

  - if both threads execute the above code concurrently, in different processors, both threads think they got the lock



- No way to implement this with only software

Hardware Support

- Compare and swap *machine instruction* (pseudo-code in C below)

```
int CAS(int *ptr, int old, int new) {
    int tmp = *ptr;   // get the value of mutex
    if (tmp == old)   // if it equals to old
       *ptr = new;    //    set it to new
    return tmp;       // return old
}
```

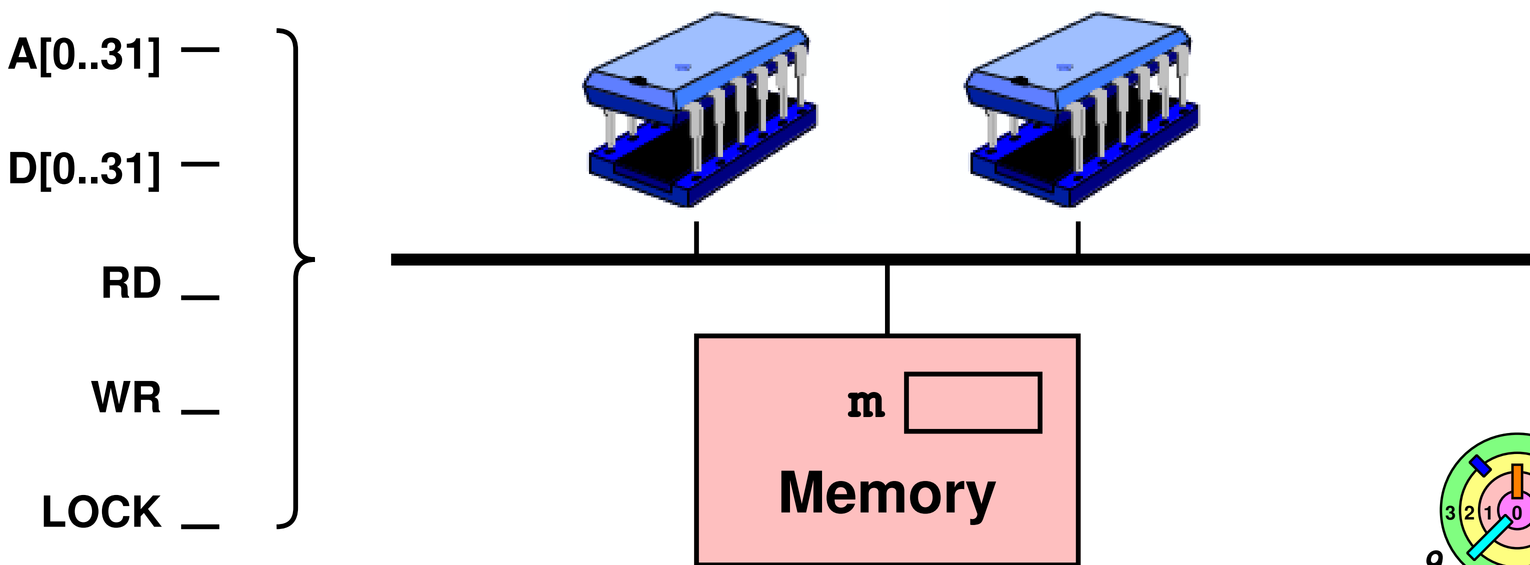
  - often implemented as a machine-level instruction
    - must execute *atomically*
      - how do you guarantee that when there are multiple CPUs?

Hardware Support

- Compare and swap *machine instruction* (pseudo-code in C below)

```
int CAS(int *ptr, int old, int new) {
    int tmp = *ptr;   // get the value of mutex
    if (tmp == old)   // if it equals to old
       *ptr = new;    //    set it to new
    return tmp;       // return old
}
```

  - e.g., assume mutex is *unlocked*, call `CAS(&lock, 0, 1)`
    - mutex is represented as a bit, 0 if unlocked, 1 if locked

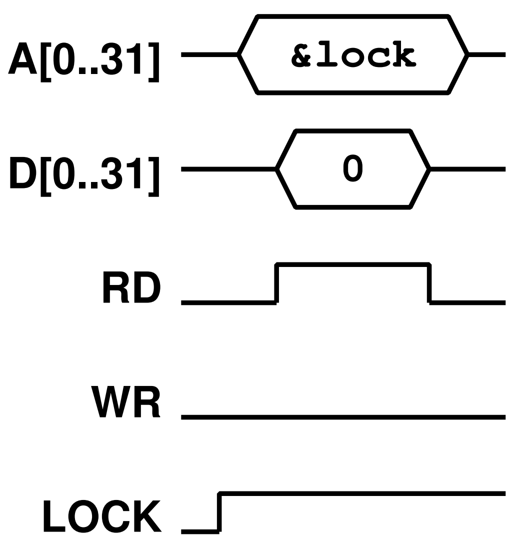


Hardware Support

- Compare and swap *machine instruction* (pseudo-code in C below)

```
int CAS(int *ptr, int old, int new) {
    int tmp = *ptr;   // get the value of mutex
    if (tmp == old)   // if it equals to old
       *ptr = new;    //    set it to new
    return tmp;       // return old
}
```

  - e.g., assume mutex is *unlocked*, call `CAS(&lock, 0, 1)`
    - mutex is represented as a bit, 0 if unlocked, 1 if locked

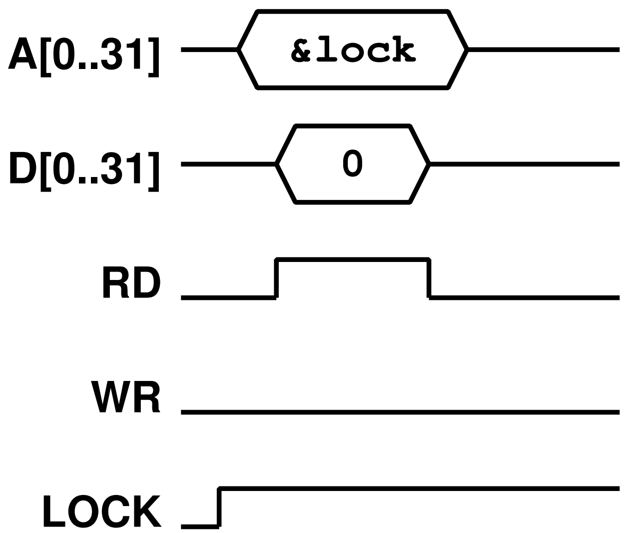


Hardware Support

- Compare and swap *machine instruction* (pseudo-code in C below)

```
int CAS(int *ptr, int old, int new) {
    int tmp = *ptr;   // get the value of mutex
    if (tmp == old)   // if it equals to old
       *ptr = new;    //    set it to new
    return tmp;       // return old
}
```

  - e.g., assume mutex is *unlocked*, call `CAS(&lock, 0, 1)`
    - mutex is represented as a bit, 0 if unlocked, 1 if locked

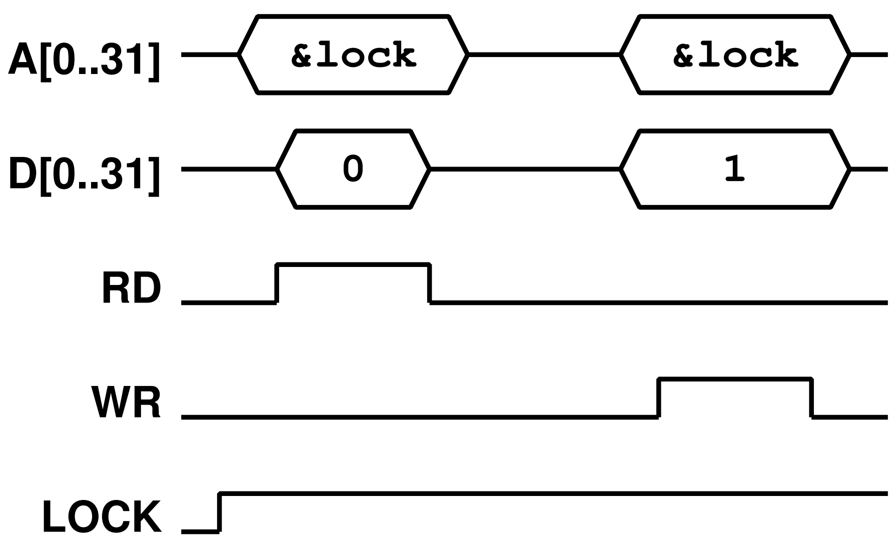


Hardware Support

- Compare and swap *machine instruction* (pseudo-code in C below)

```
int CAS(int *ptr, int old, int new) {
    int tmp = *ptr;   // get the value of mutex
    if (tmp == old)   // if it equals to old
        *ptr = new;   //    set it to new
    return tmp;       // return old
}
```

- e.g., assume mutex is *unlocked*, call `CAS(&lock, 0, 1)`
  - mutex is represented as a bit, 0 if unlocked, 1 if locked

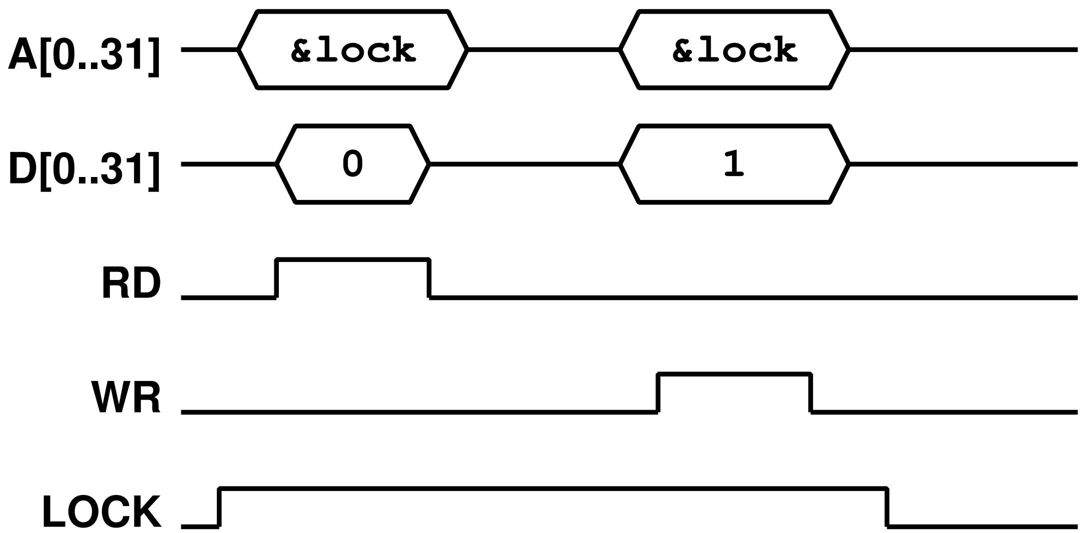


Hardware Support

- Compare and swap *machine instruction* (pseudo-code in C below)

```
int CAS(int *ptr, int old, int new) {
   int tmp = *ptr;   // get the value of mutex
   if (tmp == old)   // if it equals to old
      *ptr = new;    //   set it to new
   return tmp;       // return old
}
```

   - e.g., assume mutex is *unlocked*, call `CAS(&lock, 0, 1)`
      - mutex is represented as a bit, 0 if unlocked, 1 if locked

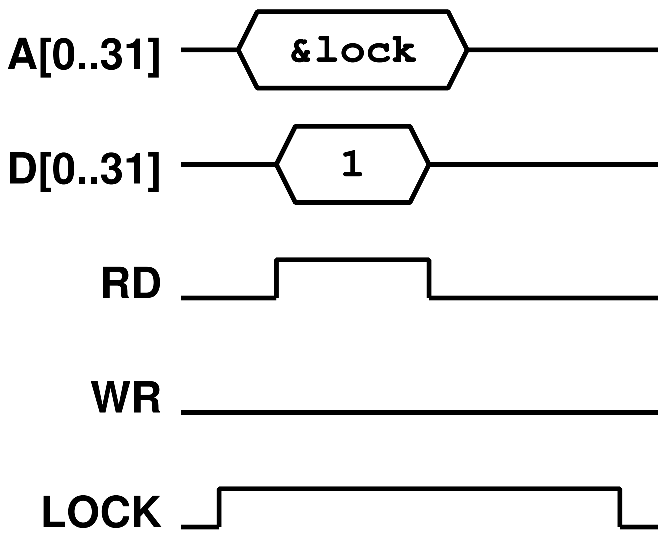


Hardware Support

- Compare and swap *machine instruction* (pseudo-code in C below)

```
int CAS(int *ptr, int old, int new) {
    int tmp = *ptr;   // get the value of mutex
    if (tmp == old)   // if it equals to old
       *ptr = new;    //    set it to new
    return tmp;       // return old
}
```

  - e.g., assume mutex is *locked*, call `CAS(&lock, 0, 1)`
    - mutex is represented as a bit, 0 if unlocked, 1 if locked



Hardware Support

- Compare and swap *machine instruction* (pseudo-code in C below)

```
int CAS(int *ptr, int old, int new) {
   int tmp = *ptr;   // get the value of mutex
   if (tmp == old)   // if it equals to old
      *ptr = new;    //    set it to new
   return tmp;       // return old
}
```

- Can implement *spin lock* using `CAS()`
  - mutex is represented as a bit, 0 if unlocked, 1 if locked

- Naive spin lock

```
void spin_lock(int *mutex) {
   while(CAS(mutex, 0, 1)) // textbook is wrong
      ;
}

void spin_unlock(int *mutex) {
   *mutex = 0;
}
```

Spin Lock

Naive spin lock

```
void spin_lock(int *mutex) {
    while(CAS(mutex, 0, 1)) // textbook is wrong
        ;
}

void spin_unlock(int *mutex) {
    *mutex = 0;
}
```

Better spin lock

```
void spin_lock(int *mutex) {
  while (1) {
    if (*mutex == 0) {
      // the mutex was at least momentarily unlocked
      if (!CAS(mutex, 0, 1))
        break;  // we have locked the mutex
      // some other thread beat us to it, try again
    }
  }
}
```

Blocking Locks

- Spin locks are *wasteful*
  - processor time wasted waiting for the lock to be released
  - barely acceptable if locks are held only briefly
- A better approach is to have a *blocking lock*
  - threads wait by having their execution suspended
  - a thread much yield the processor and join a queue of waiting threads
    - later on, get resumed explicitly

Blocking Locks

```
void blocking_lock(mutex_t *m) {
  if (m->holder != 0) {
    enqueue(m->wait_queue, CurrentThread);
    thread_switch();
  } else
    m->holder = CurrentThread;
}

void blocking_unlock(mutex_t *m) {
  if (queue_empty(m->wait_queue))
    m->holder = 0;
  else {
    m->holder = dequeue(m->wait_queue);
    enqueue(RunQueue, m->holder);
  }
}
```

- This code only works on a uniprocessor

Blocking Lock Failure Scenario (1)

1,2

```
void blocking_lock(mutex_t *m) {
  if (m->holder != 0) {
    enqueue(m->wait_queue, CurrentThread);
    thread_switch();
  } else
    m->holder = CurrentThread;
}

void blocking_unlock(mutex_t *m) {
  if (queue_empty(m->wait_queue))
    m->holder = 0;
  else {
    m->holder = dequeue(m->wait_queue);
    enqueue(RunQueue, m->holder);
  }
}
```

- On a multiprocessor, it may not work (since it has a race condition)
  - threads 1 and 2 can both think they've got the lock

Blocking Lock Failure Scenario (2)

```
  void blocking_lock(mutex_t *m) {
1   if (m->holder != 0) {
      enqueue(m->wait_queue, CurrentThread);
      thread_switch();
    } else
      m->holder = CurrentThread;
  }

  void blocking_unlock(mutex_t *m) {
2   if (queue_empty(m->wait_queue))
      m->holder = 0;
    else {
      m->holder = dequeue(m->wait_queue);
      enqueue(RunQueue, m->holder);
    }
  }
```

- On a multiprocessor, it may not work (since it has a race condition)
  - thread 2 holds the mutex and wait queue is empty and
    thread 1 tries to lock the mutex at the same time
    thread 2 is releasing the mutex
  - thread 1 may wait forever

Blocking Lock Failure Scenario (2)

```
     void blocking_lock(mutex_t *m) {
1 ➡    if (m->holder != 0) {
         enqueue(m->wait_queue, CurrentThread);
         thread_switch();
       } else
         m->holder = CurrentThread;
     }

     void blocking_unlock(mutex_t *m) {
2 ➡    if (queue_empty(m->wait_queue))
         m->holder = 0;
       else {
         m->holder = dequeue(m->wait_queue);
         enqueue(RunQueue, m->holder);
       }
     }
```

- Maybe we can fix both scenarios by making these two functions mutually exclusive (with respect to mutex `m`)
  - maybe we can using a spin lock!

Working Blocking Locks (?)

```
void blocking_lock(mutex_t *m) {
  spin_lock(m->spinlock); // okay to spin here
  if (m->holder != 0) {
    enqueue(m->wait_queue, CurrentThread);
    thread_switch();
  } else {
    m->holder = CurrentThread;
  }
  spin_unlock(m->spinlock);
}

void blocking_unlock(mutex_t *m) {
  spin_lock(m->spinlock); // okay to spin here
  if (queue_empty(m->wait_queue)) {
    m->holder = 0;
  } else {
    m->holder = dequeue(m->wait_queue);
    enqueue(RunQueue, m->holder);
  }
  spin_unlock(m->spinlock);
}
```

Will deadlock because of `thread_switch()`

Working Blocking Locks (?)

```
void blocking_lock(mutex_t *m) {
  spin_lock(m->spinlock);
  if (m->holder != 0) {
    enqueue(m->wait_queue, CurrentThread);
    spin_unlock(m->spinlock);
    thread_switch();
  } else {
    m->holder = CurrentThread;
    spin_unlock(m->spinlock);
  }
}

void blocking_unlock(mutex_t *m) {
  spin_lock(m->spinlock);
  if (queue_empty(m->wait_queue)) {
    m->holder = 0;
  } else {
    m->holder = dequeue(m->wait_queue);
    enqueue(RunQueue, m->holder);
  }
  spin_unlock(m->spinlock);
}
```

⇨ **Has a different problem** *(race condition!)*

Working Blocking Locks (?)

```
void blocking_lock(mutex_t *m) {
  spin_lock(m->spinlock);
  if (m->holder != 0) {
    enqueue(m->wait_queue, CurrentThread);
    spin_unlock(m->spinlock);
1   thread_switch();
  } else {
    m->holder = CurrentThread;
    spin_unlock(m->spinlock);
  }
}

void blocking_unlock(mutex_t *m) {
  spin_lock(m->spinlock);
  if (queue_empty(m->wait_queue)) {
    m->holder = 0;
  } else {
2   m->holder = dequeue(m->wait_queue);
    enqueue(RunQueue, m->holder);
  }
  spin_unlock(m->spinlock);
}
```

Thread 1 may be running in both processors!

Working Blocking Locks (?)

```
void blocking_lock(mutex_t *m) {
  spin_lock(m->spinlock);
  if (m->holder != 0) {
    enqueue(m->wait_queue, CurrentThread);
    spin_unlock(m->spinlock);
    thread_switch();
  } else {
    m->holder = CurrentThread;
    spin_unlock(m->spinlock);
  }
}

void blocking_unlock(mutex_t *m) {
  spin_lock(m->spinlock);
  if (queue_empty(m->wait_queue)) {
    m->holder = 0;
  else {
    m->holder = dequeue(m->wait_queue);
    enqueue(RunQueue, m->holder);
  }
  spin_unlock(m->spinlock);
}
```

- Can you do `spin_unlock()` inside `thread_switch()`?

Futexes

- ***Futex:*** Linux's fast user-space mutex (for 1×1 model)
  - safe, efficient kernel conditional queueing in Linux
    - most of the time when you try to lock a mutex, it's unlocked; so just go ahead and lock it (no system call)
    - if it's locked (by another thread), then a system call is required for this thread to obtain the lock
  - contained in it is an unsigned integer state called `value` and a queue of waiting threads

- Two *system calls* are provided to support futexes

```
futex_wait(futex_t *futex, int val) {
  if (futex->val == val)
    // sleep on the futex queue inside kernel
}

futex_wake(futex_t *futex) {
  // wake up one thread from wait queue if
  //      there is any
  ...
}
```

Ancillary Functions

- Add 1 to `*val`, return its original value

```
unsigned int atomic_inc(unsigned int *val) {
  // performed atomically
  return((*val)++); // textbook is wrong
}
```

  - e.g., x86 has "lock-prefixed instructions" so you can lock any consecutive machine instructions together and make them atomic

- Subtract 1 to `*val`, return its original value

```
unsigned int atomic_dec(unsigned int *val) {
  // performed atomically
  return((*val)--); // textbook is wrong
}
```

- Just like `CAS()`, both functions return the *previous* lock value

Attempt 1

- `futex->val`
  - 0 means unlocked; otherwise, locked

```
void lock(futex_t *futex) {
  unsigned int c;
  while ((c = atomic_inc(&futex->val)) != 0)
    futex_wait(futex, c+1);
}

void unlock(futex_t *futex) {
  futex->val = 0;
  futex_wake(futex);
}
```

- Problem with `unlock()`
  - slow because `futex_wake()` is a system call
- Problem with `lock()`
  - threads run in lock steps in a multiprocessor environment!
  - `futex->val` may wrap-around

Attempt 2

- `futex->val` can only take on values of 0, 1, and 2
  - 0 means unlocked
  - 1 means locked but no waiting thread
  - 2 means locked with the possibility of waiting threads

```
void lock(futex_t *futex) {
  unsigned int c;
  if ((c = CAS(&futex->val, 0, 1) != 0)
    do {
      if (c == 2 || (CAS(&futex->val, 1, 2) != 0))
        futex_wait(futex, 2);
    } while ((c = CAS(&futex->val, 0, 2)) != 0));
}

void unlock(futex_t *futex) {
  if (atomic_dec(&futex->val) != 1) {
    futex->val = 0;
    futex_wake(futex);
  }
}
```

textbook is wrong

Attempt 2

Complications

```
void lock(futex_t *futex) {
  unsigned int c;
  if ((c = CAS(&futex->val, 0, 1)) != 0)
    do {
      if (c == 2 || (CAS(&futex->val, 1, 2) != 0))
        futex_wait(futex, 2);
    } while ((c = CAS(&futex->val, 0, 2)) != 0));
}

void unlock(futex_t *futex) {
  if (atomic_dec(&futex->val) != 1) {
    futex->val = 0;
    futex_wake(futex);
  }
}
```

textbook is wrong

- the implementation of `futex_wait()` and `futex_wake()` must be *atomic* to avoid race conditions
  - https://www.akkadia.org/drepper/futex.pdf

Thread Synchronization Summary

- **Spin locks**
  - used if the duration of waiting is expected to be small
    - as in the case at the beginning of `blocking_lock()`
- **Sleep (or blocking) locks**
  - used if the duration of waiting is expected to be long
- **Futexes**
  - optimized version of blocking locks
- **In your kernel assignmen #1, you need to implement *kernel* threads**
  - very different from *user* threads
    - keep in mind that the `weenix` kernel is *non-preemptive*
    - the kernel is very powerful (and therefore, must be bug free, and therefore, *your code* must be bug free)
  - in kernel assignmen #3, you need to implement *user* threads/processes (well, `MTP=0`, still one thread per process)