CSCI 350
Ch. 5 – Synchronization

Mark Redekopp

Michael Shindler & Ramesh Govindan

RACE CONDITIONS AND ATOMIC OPERATIONS

Race Condition

- A *race condition* occurs when the behavior of the program depends on the interleaving of operations of different threads.
- Example: Assume x = 2
  - T1: x = x + 5
  - T2: x = x * 5
- Outcomes
  - Case 1: T1 then T2
    - After T1: x = 7
    - After T2: x = 35
  - Case 2: T2 then T1
    - After T2: x = 10
    - After T1: x = 15
  - Case 3: Both read before either writes, T2 Write, T1 Write
    - x = 7
  - Case 4: Both read before either writes, T1 Write, T2 Write
    - x = 10

Interleavings

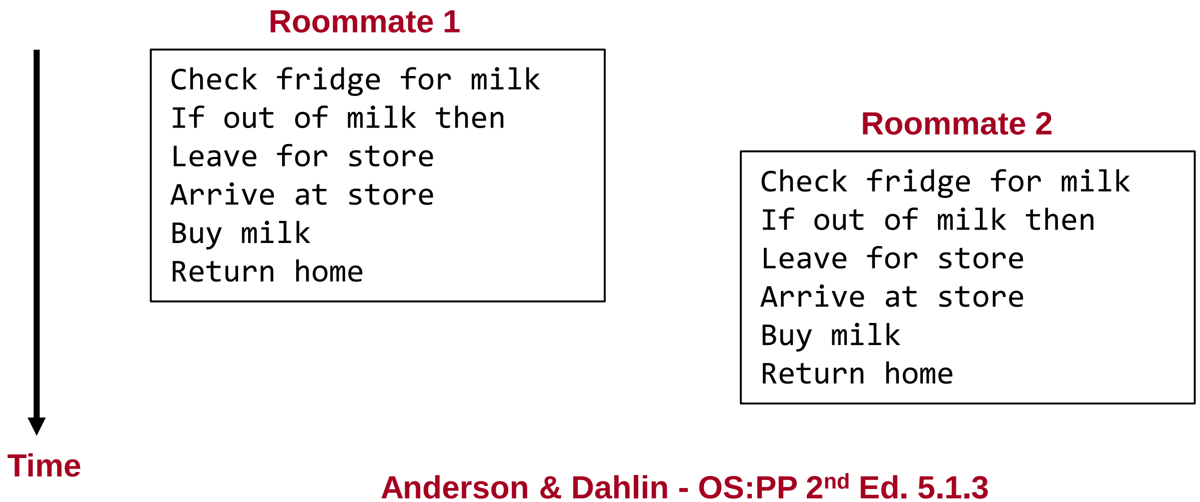
- Code must work under all interleavings
- Just because it works once doesn't mean its bug-free
  - Heisen-"bug" (Heisenberg's uncertainty principle & the observer effect)
    - A bug that cannot be reproduced reliably or changes when debugging instrumentation is added
    - Load-bearing print statement
  - Bohr-"bug"
    - A bug that can be reproduced regardless of debugging instrumentation

Atomic Operations

- An operation that is indivisible (i.e. that cannot be broken into suboperations or whose parts cannot be interleaved)
- Computer hardware generally guarantees:
  - A single memory read is atomic
  - A single memory write is atomic
- Computer hardware does not generally guarantee atomicity across multiple instructions:
  - A Read-Write or Read-Modify-Write cycle
- To guarantee atomic execution of multiple operations we generally need some kind of synchronization variables supported by special HW instruction support

An Example: Got Milk?

- Suppose you and your roommate want to ensure there is always milk available
- **Synchronization should ensure:**
  - Safety: Mutual exclusion (i.e. only 1 person buys milk)
  - Liveness: Someone makes progress (i.e. there is milk)



Roommate 1

```
Check fridge for milk
If out of milk then
Leave for store
Arrive at store
Buy milk
Return home
```

Roommate 2

```
Check fridge for milk
If out of milk then
Leave for store
Arrive at store
Buy milk
Return home
```

This approach ensures liveness but not safety

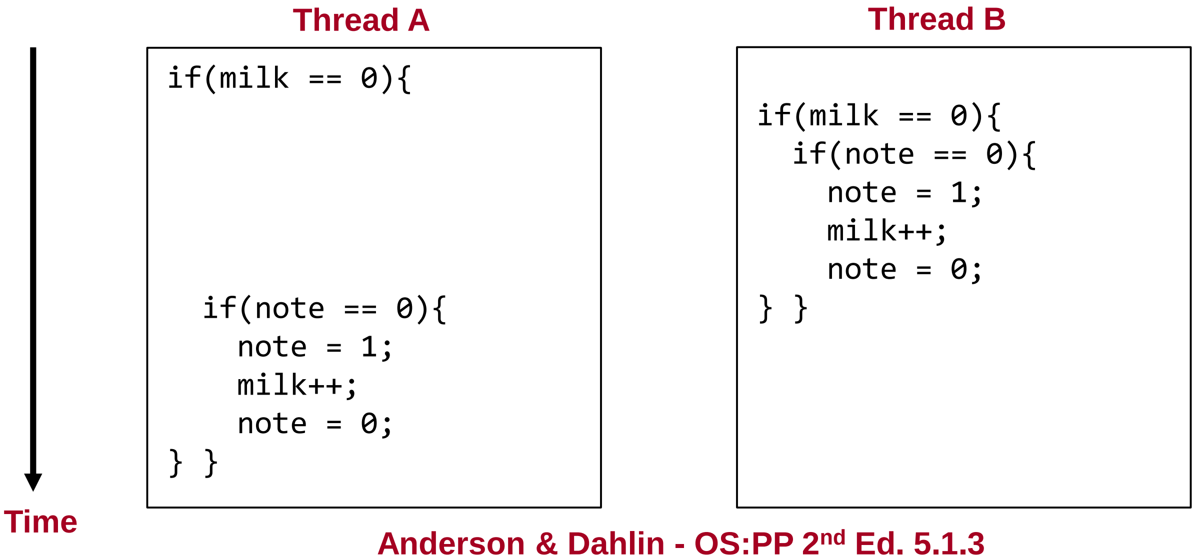
Anderson & Dahlin - OS:PP 2nd Ed. 5.1.3

Got Milk: Option 1

- Suppose you and your roommate want to ensure there is always milk available

Algorithm

```
if(milk == 0){
  if(note == 0){
    note = 1;
    milk++;
    note = 0;
} }
```



Thread A

```
if(milk == 0){






  if(note == 0){
    note = 1;
    milk++;
    note = 0;
} }
```

Thread B

```

if(milk == 0){
  if(note == 0){
    note = 1;
    milk++;
    note = 0;
} }
```

Time

This approach still ensures liveness but not safety

Anderson & Dahlin - OS:PP 2nd Ed. 5.1.3

Got Milk: Option 2

- Post note early: "I will buy milk if needed"
  - Does it ensure safety?

Algorithm

```
noteA = 1;
if(noteB == 0){
  if(milk == 0){
    milk++;
} }
noteA = 0;
```

Thread A

```
noteB = 1;
if(noteA == 0){
  if(milk == 0){
    milk++;
} }
noteB = 0;
```

Thread B

Posting notes ahead of time "ensures" safety.

Important: Actually this may not work on a modern processor with instruction reordering and certain memory consistency models.

Time

?

```
noteB = 1;
if(noteA == 0){
 if(milk == 0){
   milk++;
} }
noteB = 0;
```

noteA	milk	Outcome
0	0	Only B will buy. A hasn't started and won't check since noteB must be 1 now. By the time A can check, B will be done checking/purchasing.
0	>0	A hasn't started. Already milk. B doesn't buy.
1	0	B won't consider buying. A is checking/buying.
1	>0	B won't consider buying.

Anderson & Dahlin - OS:PP 2nd Ed. 5.1.3

Got Milk: Option 2

- Post note early: "I will buy milk if needed"
  - Does it ensure liveness?

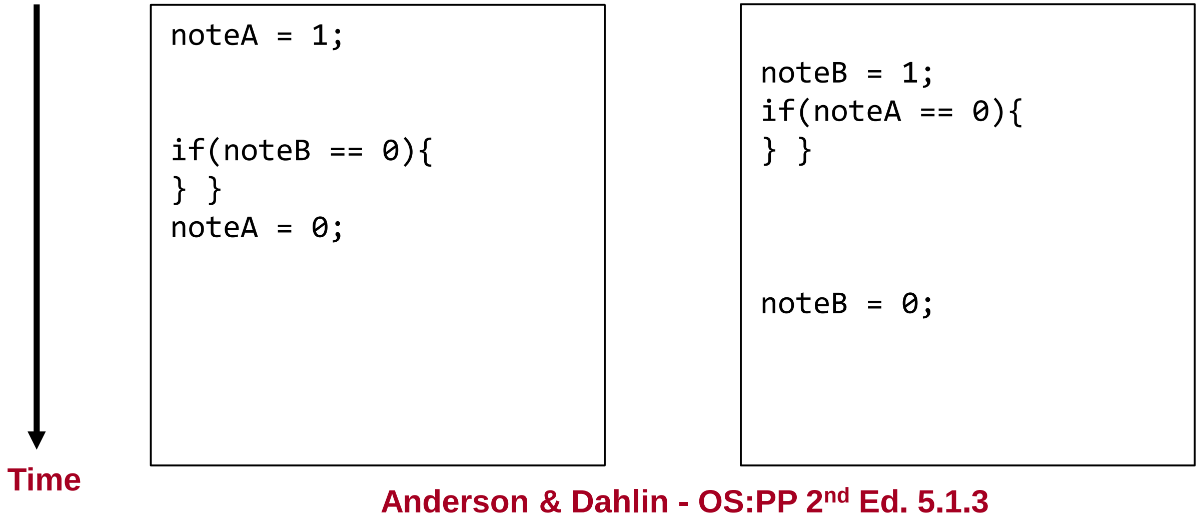
Algorithm

```
noteA = 1;
if(noteB == 0){
  if(milk == 0){
    milk++;
} }
noteA = 0;
```

```
noteB = 1;
if(noteA == 0){
  if(milk == 0){
    milk++;
} }
noteB = 0;
```

Thread A

Thread B



```
noteA = 1;


if(noteB == 0){
} }
noteA = 0;
```

```

noteB = 1;
if(noteA == 0){
} }



noteB = 0;
```

This approach ensures safety but not liveness

Anderson & Dahlin - OS:PP 2nd Ed. 5.1.3

Got Milk: Option 3

- Preferred buyer (i.e. B) if we both arrive at similar times, A will wait until no note from B
  - Notice this requires asymmetric code. What if 3 or more threads?
  - "Spins" in the while loop wasting CPU time (could deschedule the thread)

Algorithm

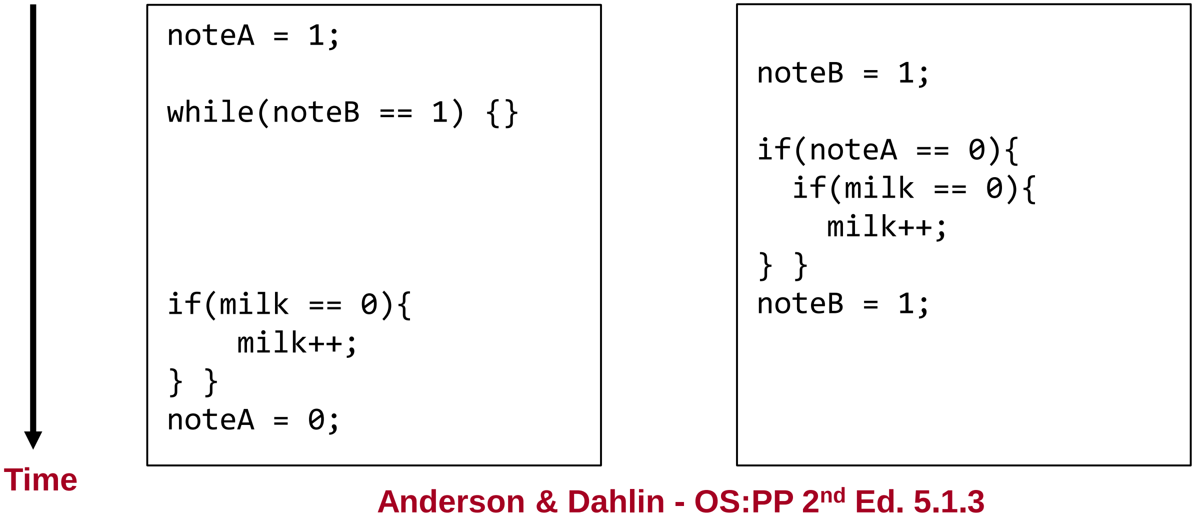
```
noteA = 1;
while(noteB == 1) {}
if(milk == 0){
    milk++;
} }
noteA = 0;
```

Thread A

```
noteB = 1;
if(noteA == 0){
  if(milk == 0){
    milk++;
} }
noteB = 1;
```

Thread B

Posting notes ahead of time ensures safety. Thread A now <u>waits</u> until B is not checking/buying and then checks itself which guarantees someone will buy milk if needed.



```
noteA = 1;

while(noteB == 1) {}
```

```
noteB = 1;

if(noteA == 0){
  if(milk == 0){
    milk++;
} }
noteB = 1;
```

```
if(milk == 0){
    milk++;
} }
noteA = 0;
```

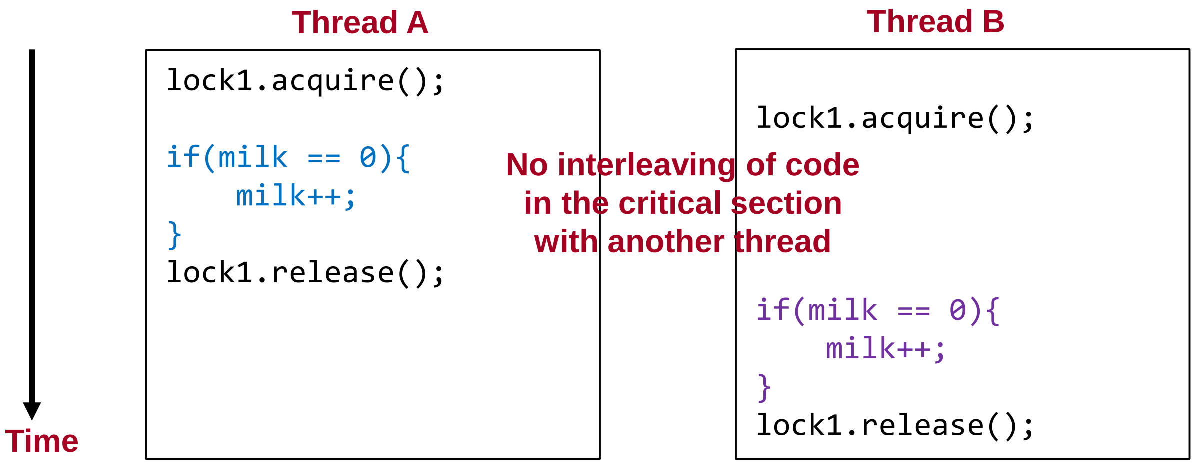
Anderson & Dahlin - OS:PP 2nd Ed. 5.1.3

Locking

- Locking ensures safety (mutual exclusion)
- Provides more succinct code
- This example ensures liveness since threads will wait/block until they can acquire the lock and then check the milk
  - Waiting thread is descheduled

Algorithm

```
lock1.acquire();
if(milk == 0){
    milk++;
}
lock1.release();
```



Thread A

```
lock1.acquire();

if(milk == 0){
    milk++;
}
lock1.release();
```

Thread B

```
lock1.acquire();




if(milk == 0){
    milk++;
}
lock1.release();
```

No interleaving of code in the critical section with another thread

Example: Parallel Processing

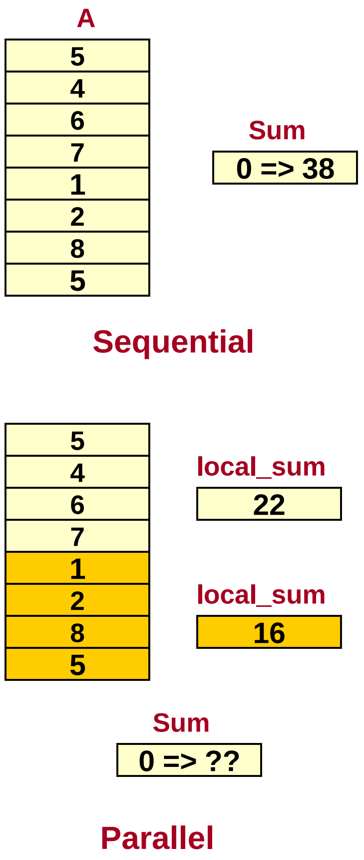
- Sum an array, A, of numbers {5,4,6,7,1,2,8,5}
- Sequential method

```
    for(i=0; i < 7; i++) { sum = sum + A[i]; }
```

- Parallel method (2 threads with ID=0 or 1)

```
for(i=ID*4; i < (ID+1)*4; i++) {
    local_sum = local_sum + A[i]; }
sum = sum + local_sum;
```

- Problem
  - Updating a shared variable (e.g. sum)
  - Both threads read sum=0, perform sum=sum+local_sum, and write their respective values back to sum
  - Any read/modify/write of a shared variable is susceptible
- Solution
  - Atomic updates accomplished via locking or lock-free synchronization



A

5
4
6
7
1
2
8
5

Sum

0 => 38

Sequential

5
4
6
7
1
2
8
5

local_sum

22

local_sum

16

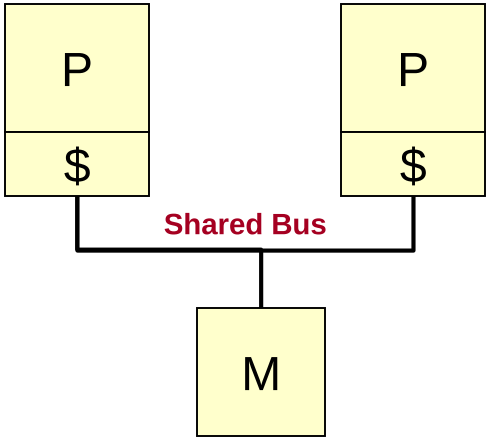
Sum

0 => ??

Parallel

Atomic Operations

- Read/modify/write sequences are usually done with separate instructions
- Possible Sequence:
  - P1 Reads sum (load/read)
  - P1 Modifies sum (add)
  - P2 Reads sum (load/read)
  - P1 Writes sum (store/write)
  - P2 uses old value…
- Partial Solution: Have a separate flag/"lock" variable (0=Lock is free/unlocked, 1 = Locked)
- Lock variable is susceptible to same problem as sum (read/modify/write)
  - if(lock == 0) lock = 1;
- Hardware has to support some kind of instruction to implement atomic operations usually by not releasing bus between read and write



Thread 1:	Thread 2:
Lock L	Lock L
Update sum	Update sum
Unlock L	Unlock L

Locking/Atomic Instructions

- TSL (Test and Set Lock)
  - tsl reg, addr_of_lock_var
  - Atomically stores const. ‘1’ in lock_var value & returns lock_var in reg
    - Atomicity is ensured by HW not releasing the bus during the RMW cycle
- CAS (Compare and Swap)
  - cas addr_to_var, old_val, new_val
  - Atomically performs:
    - if (*addr_to_var != old_val) return false
    - else *addr_to_var = new_val; return true;
  - x86 Implementation
    - old_value always in $eax
    - CMPXCH r2, r/m1
      - if($eax == *r/m1) ZF=1; *r/m1 = r2;
      - else { ZF = 0; $eax = *r/m1; }

```
ACQ:          tsl    (lock_addr), %reg
              cmp    $0,%reg
              jnz    ACQ
              return;

REL:          move  $0,(lock_addr)
```

```
ACQ:          move   $1, %edx
L1:           move   $0, %eax
        lock  cmpxchg %edx, (lock_addr)
              jnz     L1
              ret

REL:          move   $0, (lock_addr)
```

Lockless Atomic Updates

- CAS (Compare and Swap) [x86]
  - x86 Implementation
    - old_value always in $eax
    - CMPXCH r2, r/m1
      - if($eax == *r/m1) ZF=1; *r/m1 = r2;
      - else { ZF = 0; $eax = *r/m1; }
- LL and SC (MIPS & others)
  - Lock-free atomic RMW
  - LL = Load Linked
    - Normal lw operation but tells HW to track any external accesses to addr.
  - SC = Store Conditional
    - Like sw but only stores if no other writes since LL & returns 0 in reg. if failed, 1 if successful

```
// High-level implementation
synchronized {
   sum += local_sum;
}
```

```
// x86 implementation
INC:            move   (sum_addr), %edx
                move   %edx, %eax
                add     (local_sum),%edx
         lock   cmpxchg %edx, (sum_addr)
                jnz     INC
                ret
```

```
// MIPS implementation

                LA      $t1,sum

INC:            LL      $5,0($t1)

                ADD    $5,$5,local_sum

                SC      $5,0($t1)

                BEQ    $5,$zero,UPDATE
```

SYNCHRONIZATION VARIABLES

Lock Properties

- Lock has two states, BUSY and FREE
  - Initially free
  - Acquire waits until free and sets to busy (this step is atomic)
    - Even if multiple threads call acquire at the same instant, one will win and the other(s) will wait
  - Release makes lock free (allowing waiter to proceed)

Lock Properties

- Locks should ensure:
  - Safety (i.e. mutual exclusion)
  - Liveness
    - A holder should release it at some point
    - If the lock is free, caller should acquire it …OR…
    - If it the lock is busy a bounded should exist on the number of times other threads can acquire it before the thread does.
      - A stronger condition might be FIFO ordering

A First Lock: SpinLock

- Uses atomic instructions (e.g. test-and-set-lock or compare-and-swap)
  - Here atomic_swap swaps two variables atomically
- Spins (loops) until the lock is acquired
- Pro: Great when critical section is short (fast lock/unlock)
  - Context switch may be longer than the time to execute a critical section
- Con: Wastes processor resources during spinning
- Any easy way to exploit?

```
class SpinLock
{
  int value;
public:
  SpinLock() : value(FREE), holder(NULL) {}
  ~SpinLock() { /* code */ }
  void acquire()
  {
     while(1){
       int curr = BUSY;
       atomic_swap(curr, value);
       if(curr == FREE) {
           return;
       }
  }  }
  void release()
  {
     value = FREE;
  }
};
```

A First Lock: SpinLock

- May maintain the holder to ensure another thread doesn't unlock mistakenly/maliciously

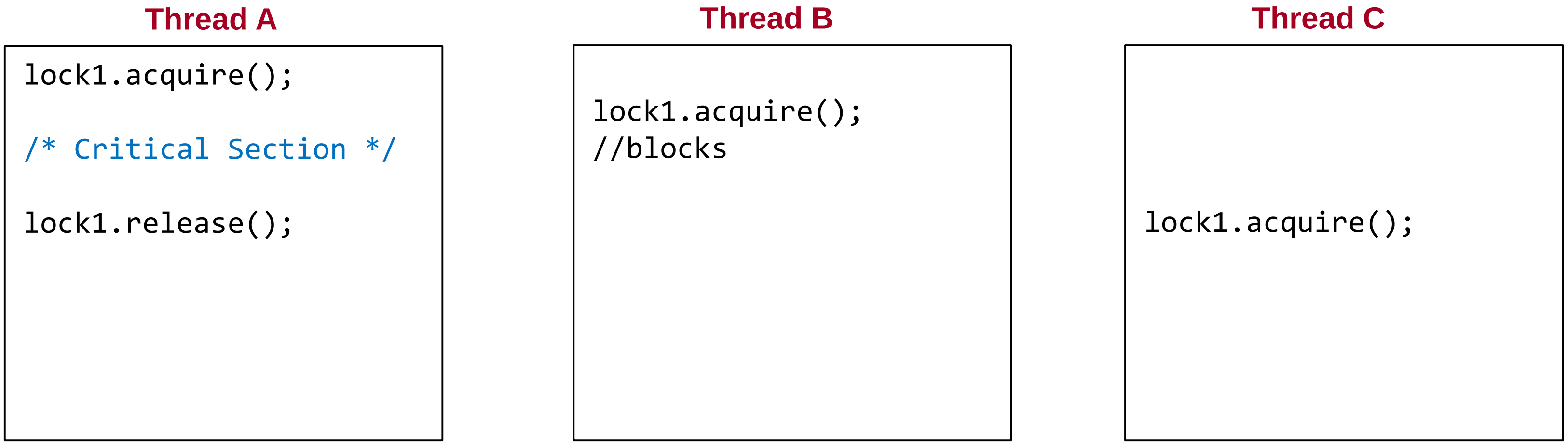
```
class SpinLock
{
  int value;
  Thread* holder;
public:
  SpinLock() : value(FREE), holder(NULL) {}
  ~SpinLock() { /* code */ }
  void acquire()
  {
     while(1){
       int curr = BUSY;
       atomic_swap(curr, value);
       if(curr == FREE) {
           holder = curr_thread(); return;
       }
  }  }
  void release()
  {
     if(curr_thread() == holder)
       value = FREE;
  }
};
```

A Queueing (Blocking) Lock

- We can block threads when they are unable to acquire the lock
- Can you think of a liveness issue that exists?

```
class Lock
{
  int value;        Queue waiters;
  Thread* holder;   SpinLock mutex;
public:
  Lock() : value(FREE), holder(NULL) {}
  ~Lock() { /* code */ }
  void acquire()
  {  mutex.acquire();
     while(1){
       int curr = BUSY;
       atomic_swap(curr, value);
       if(curr == FREE) {
           holder = curr_thread(); break;
       } else {
          waiters.append(self);
          /* context switch */
          thread_block(curr_thread(), &mutex);
       }
     } mutex.release();
  }
  void release()
  {  mutex.acquire();
     if(holder == curr_thread()) {
       if(!waiters.empty())
          thread_unblock(waiters.pop_front());
       value = FREE;
     } mutex.release();
  }
};
```

- Consider the following interleaving
  - Thread B is blocked
  - When thread A releases does our lock implementation guarantee thread B gets the lock?



Thread A

```
lock1.acquire();

/* Critical Section */

lock1.release();
```

Thread B

```

lock1.acquire();
//blocks
```

Thread C

```



lock1.acquire();
```

A Queueing Lock

- On release we can leave the value = BUSY and awaken one waiter
- No new thread can come in and "steal" the lock

```
class Lock
{
  int value;       Queue waiters;
  Thread* holder;  SpinLock mutex;
public:
  Lock() : value(FREE), holder(NULL) {}
  ~Lock() { /* code */ }
  void acquire()
  {
     mutex.acquire();
     int curr = BUSY;
     atomic_swap(curr, value);
     if(curr == FREE) { break; }
     } else {
        waiters.append(self);
        /* ctxt switch & release/reacquires mutex */
        thread_block(curr_thread(), &mutex);
     }
     holder = curr_thread();
     mutex.release();
  }
  void release()
  {  mutex.acquire();
     if(holder == curr_thread()) {
       if(!waiters.empty())
          thread_unblock(waiters.pop_front());
       else { value = FREE; }
     } mutex.release();
  }
};
```

Blocking vs. Non-Blocking Locks

- Acquire (Blocking)
  - If lock == UNLOCKED then lock = LOCKED and return
  - else **block/sleep** until the holder releases the lock giving it to you
    - Need some kind of loop to keep checking everytime you are awakened
    - Only returns once it has acquired the lock
- TryAcquire (Non-blocking)
  - If lock == UNLOCKED then lock = LOCKED and return true;
  - else return false;

Linux Mutex

- See code
  - http://elixir.free-electrons.com/linux/latest/source/kernel/locking/mutex.c line 236
- Combination of "spin" and "queueing" lock
  - Common case: lock is free [Fast Path]
    - First perform an atomic compare-and-swap and check if you got the lock. If so, done!
      - Line 139: __mutex_trylock_fast()
  - Next most common case: Locked but no other waiters [Medium Path]
    - Spin for a little while so we don't have to context switch
      - Line 738: __mutex_lock_common()
  - Block and add yourself to queue [Slow Path]

A Sanity Check Question

- What is wrong with this attempt to synchronize updates to the global variable x?
- Should spinlocks be used on a uniprocessor system?

```
int x = 1;

/* Thread 1 */
void t1(void* arg)
{
  Lock the_lock;
  the_lock.acquire();
  x++;
  the_lock.release();
}

/* Thread 2 */
void t2(void* arg)
{
  Lock the_lock;
  the_lock.acquire();
  x++;
  the_lock.release();
}
```

A Sanity Check Answer

- What is wrong with this attempt to synchronize updates to the global variable x?
  - Different locks (mylock1, mylock2)
  - Should only be 1
- Should spinlocks be used on a uniprocessor system?
  - No, spinning because another thread has the lock which to release will require you to give up the processor (i.e. context switch)…spinning is pointless.

```
int x = 1;
Lock the_lock;

/* Thread 1 */
void t1(void* arg)
{
  ~~Lock the_lock;~~
  the_lock.acquire();
  x++;
  the_lock.release();
}

/* Thread 2 */
void t2(void* arg)
{
  ~~Lock the_lock;~~
  the_lock.acquire();
  x++;
  the_lock.release();
}
```

Shared Objects

- **Shared Object (def.)**: An object that will be accessed by multiple threads
  - Should maintain state/shared data variables and the synchronization variable(s) needed to control access to them
- Methods should lock the object when updating shared state

```
class ObjA
{
  void f1(int newVal);

private:
  /* State vars */
  int sum;
  vector<int> vals;
  /* Synchronization var */
  Lock the_lock;
}

void ObjA::f1(int newVal)
{
  the_lock.acquire();
  vals.push_back(newVal);
  sum += newVal;
  the_lock.release();
}
```

Non-Blocking Bounded Queue

- Examine the Buffer (queue) class to the right
- Assuming multiple threads will be producing and consuming values we will have race conditions
  - Two producers have a race condition on 'tail'
  - Two consumers have a race condition on 'head'
  - All threads have a race condition on 'count'
- Demo: Sample output

```
class Buffer
{
  int data[MAXSIZE];
  int count;
  int head, tail;
public:
  Buffer() : count(0), head(0), tail(0)
    { }
  bool try_produce(int item)
  {
    bool status = false;
    if(count != MAXSIZE) {
      data[tail++] = item; count++;
      if(tail == MAXSIZE) tail = 0;
      status = true;
    }
    return status;
  }
  bool try_consume(int* item)
  {
    bool status = false;
    if(count != 0) {
      *item = data[head++]; count--;
      if(head == MAXSIZE) head = 0;
      status = true;
    }
    return status;
  }
};
```

Non-Blocking Bounded Queue

- By adding a lock we can ensure mutual exclusion
- However, consumers may find the buffer empty or producers may find the buffer full and unable to complete their operation
  - We simply return in this case
- Demo
- By using condition variables we can have the threads block until they will be able to perform their desired task

```
// Consumer code
int val;
while(!buf->try_consume(&val))
 {}
```

```
// Producer code
while(!buf->try_produce(val))
  {}
```

```
class Buffer
{
  int data[MAXSIZE];
  int count;
  int head, tail;
  pthread_mutex_t mutex;
public:
  Buffer() : count(0), head(0), tail(0)
  { pthread_mutex_init(&mutex, NULL); }
  bool try_produce(int item)
  {
    bool status = false;
    pthread_mutex_lock(&mutex);
    if(count != MAXSIZE) {
      data[tail++] = item; count++;
      if(tail == MAXSIZE) tail = 0;
      status = true;
    }
    pthread_mutex_unlock(&mutex);
    return status;
  }
  bool try_consume(int* item)
  {
    bool status = false;
    pthread_mutex_lock(&mutex);
    if(count != 0) {
      *item = data[head++]; count--;
      if(head == MAXSIZE) head = 0;
      status = true;
    }
    pthread_mutex_unlock(&mutex);
    return status;
  }
};
```

Condition Variables

- Condition variables are not really "variables"
  - They don't store any data/value
- CVs assume you have other shared state that you are looking at to determine you can not make progress and allow you to block, waiting for an event
- CVs always are paired with a lock which is guaranteeing exclusive access to the shared state that you are looking at
- CVs provide the following API
  - wait(Lock* mutex): Puts the thread to sleep until signaled
    - The associated lock must be LOCKED on a call to wait, which will unlock it as it puts the thread to sleep and reacquire it once awoken
  - signal(): Wakes one waiting thread
  - broadcast(): Wakes all waiting thread
- CVs are memory-less
  - A signal() when no one is waiting is forgotten

Blocking Bounded Queue

- By using condition variables we can have the threads block until they will be able to perform their desired task
- Producers need to
  - Wait while buffer is full
  - Signal any waiting consumers if the buffer was empty but now will have 1 item
- Consumers need to
  - Wait while buffer is empty
  - Signal any waiting producers if the buffer was full but now has 1 free spot
- Design tip:
  - A good design for a shared object is to have 1 lock and one or more CVs

```
class Buffer
{
  int data[10];
  int count, head, tail;
  pthread_mutex_t mutex;
  pthread_cond_t prodcv, conscv;
public:
  Buffer() : count(0), head(0), tail(0)
  {
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&prodcv, NULL);
    pthread_cond_init(&conscv, NULL);
  }
  void produce(int item)
  {
    pthread_mutex_lock(&mutex);
    while(count == MAXSIZE) {
      pthread_cond_wait(&prodcv, &mutex);
    }
    if(count == 0) pthread_cond_signal(&conscv);
    data[tail++] = item; count++;
    if(tail == MAXSIZE) tail = 0;
    pthread_mutex_unlock(&mutex);
  }
  void consume(int* item)
  {
    pthread_mutex_lock(&mutex);
    while(count == 0){
      pthread_cond_wait(&conscv, &mutex);
    }
    if(count == MAXSIZE)
      pthread_cond_signal(&prodcv);
    *item = data[head++]; count--;
    if(head == MAXSIZE) head = 0;
    pthread_mutex_unlock(&mutex);
  }
};
```

Hansen/Mesa CV Semantics

- Why were the calls to "wait" inside a while loop in the previous bounded buffer code?
- Hansen/Mesa CV Semantics
  - When signal() is called, a waiter is awakened but does not necessarily get the processor or associated lock immediately
  - From our bounded buffer example, say:
    - A producer signals a waiting consumer, C1, that something is available
    - Before C1 is scheduled another consumer thread C2 runs, gets the lock, and consumes an item making the buffer empty again
    - When C1 actually gets the lock, buffer is still empty
  - **Wait should always be in a loop** to ensure the condition you are checking is valid after you awake

```
class Buffer
{
  int data[10];
  int count, head, tail;
  pthread_mutex_t mutex;
  pthread_cond_t prodcv, conscv;
public:
  Buffer() : count(0), head(0), tail(0)
  {
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&prodcv, NULL);
    pthread_cond_init(&conscv, NULL);
  }
  void produce(int item)
  {
    pthread_mutex_lock(&mutex);
    while(count == MAXSIZE) {
      pthread_cond_wait(&prodcv, &mutex);
    }
    if(count == 0) pthread_cond_signal(&conscv);
    data[tail++] = item; count++;
    if(tail == MAXSIZE) tail = 0;
    pthread_mutex_unlock(&mutex);
  }
  void consume(int* item)
  {
    pthread_mutex_lock(&mutex);
    while(count == 0){
      pthread_cond_wait(&conscv, &mutex);
    }
    if(count == MAXSIZE)
      pthread_cond_signal(&prodcv);
    *item = data[head++]; count--;
    if(head == MAXSIZE) head = 0;
    pthread_mutex_unlock(&mutex);
  }
};
```

Hoare CV Semantics

- Hoare Semantics
  - Signaler gives lock and processor to signaled thread ensuring no other thread can modify the state
  - Now signal() must also take the lock as an arg.
- Can make it harder to create a correct implementation
  - In produce() find the red highlighted line, what could go wrong when the producer signals a consumer in the line above?
- tail and count have not been updated but the producer has stopped running and lost the lock
  - Usually, Hoare semantics indicate that the signaler gets the processor and the lock back once the waiter leaves its critical section
  - Requires greater control over scheduling
- Most OSs use Mesa semantics!

```
class Buffer
{
 int data[10];
 int count, head, tail;
 pthread_mutex_t mutex;
 pthread_cond_t prodcv, conscv;
public:
 Buffer() : count(0), head(0), tail(0)
 {
  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&prodcv, NULL);
  pthread_cond_init(&conscv, NULL);
 }
 void produce(int item)
 {
  pthread_mutex_lock(&mutex);
  if(count == MAXSIZE) {
     pthread_cond_wait(&prodcv, &mutex);
  }
  if(count == 0)
     pthread_cond_signal(&conscv, &mutex);
  data[tail++] = item; count++;
  if(tail == MAXSIZE) tail = 0;
  pthread_mutex_unlock(&mutex);
 }
 void consume(int* item)
 {
  pthread_mutex_lock(&mutex);
  if(count == 0){
    pthread_cond_wait(&conscv, &mutex);
  }
  if(count == MAXSIZE)
    pthread_cond_signal(&prodcv, &mutex);
  *item = data[head++]; count--;
  if(head == MAXSIZE) head = 0;
  pthread_mutex_unlock(&mutex);
 }
};
```

What-If 1

- In a normal CV, wait atomically:
  - Unlocks
  - Sleeps
- Do they have to be performed atomically (see red highlighted lines)?
- Yes
  - Could miss a signal if consume() runs between unlock and sleep

```
class Buffer
{
  int data[10];
  int count, head, tail;
  pthread_mutex_t mutex;
  pthread_cond_t prodcv, conscv;
public:
  Buffer() : count(0), head(0), tail(0)
  {
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&prodcv, NULL);
    pthread_cond_init(&conscv, NULL);
  }
  void produce(int item)
  {
    pthread_mutex_lock(&mutex);
    while(count == MAXSIZE) {
      pthread_mutex_unlock(&mutex);
      pthread_cond_wait(&prodcv);
    }
    if(count == 0) pthread_cond_signal(&conscv);
    data[tail++] = item; count++;
    if(tail == MAXSIZE) tail = 0;
    pthread_mutex_unlock(&mutex);
  }
  void consume(int* item)
  {
    pthread_mutex_lock(&mutex);
    while(count == 0){
      pthread_cond_wait(&conscv, &mutex);
    }
    if(count == MAXSIZE)
      pthread_cond_signal(&prodcv);
    *item = data[head++]; count--;
    if(head == MAXSIZE) head = 0;
    pthread_mutex_unlock(&mutex);
  }
};
```

Examples derived from:
http://homes.cs.washington.edu/~arvind/cs422/lectureNotes/l8-6.pdf

What-If 2

- For this question, assume only 1 consumer and RMW of count (just for sake of argument)
- Does the signaler (consumer) need to acquire the lock?
- Yes, again if the consumer runs in between the producer's check of count and wait on count, we might miss a signal

```
class Buffer
{
  int data[10];
  int count, head, tail;
  pthread_mutex_t mutex;
  pthread_cond_t prodcv, conscv;
public:
  Buffer() : count(0), head(0), tail(0)
  {
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&prodcv, NULL);
    pthread_cond_init(&conscv, NULL);
  }
  void produce(int item)
  {
    pthread_mutex_lock(&mutex);
    while(count == MAXSIZE) {
      pthread_cond_wait(&prodcv, &mutex);
    }
    if(count == 0) pthread_cond_signal(&conscv);
    data[tail++] = item; count++;
    if(tail == MAXSIZE) tail = 0;
    pthread_mutex_unlock(&mutex);
  }
  void consume(int* item)
  {
    ~~pthread_mutex_lock(&mutex);~~
    *item = data[head++];
    if(head == MAXSIZE) head = 0;
    pthread_cond_signal(&prodcv);

    ~~pthread_mutex_unlock(&mutex);~~
  }
};
```

Examples derived from:
http://homes.cs.washington.edu/~arvind/cs422/lectureNotes/l8-6.pdf

What-If 3

- What if we update state while we hold the lock but just call signal() after we release the lock.
  - Any problem?
- Not if waiters re-check the condition (i.e. are in a while loop) as they should be
- But realize some difference in operation may occur as a waiter for the mutex/lock will be added to the ready list before the thread waiting on the CV

```
void produce(int item)
 {
  pthread_mutex_lock(&mutex);
  while(count == MAXSIZE) {
   printf("Buffer full...producer waiting\n");
   pthread_cond_wait(&prodcv, &mutex);
  }
  if(count == 0) pthread_cond_signal(&conscv);
  data[tail++] = item; count++;
  if(tail == MAXSIZE) tail = 0;
  pthread_mutex_unlock(&mutex);
 }
 void consume(int* item)
 {
  pthread_mutex_lock(&mutex);
  while(count == 0){
   printf("Buffer empty...consumer waiting\n");
   pthread_cond_wait(&conscv, &mutex);
  }
  *item = data[head++]; count--;
  if(head == MAXSIZE) head = 0;
  if(count == MAXSIZE-1) {
   pthread_mutex_unlock(&mutex);
   pthread_cond_signal(&prodcv);
  }
  else { pthread_mutex_unlock(&mutex); }
 }
```

Examples derived from:
http://homes.cs.washington.edu/~arvind/cs422/lectureNotes/l8-6.pdf

Semaphores

- Semaphores define an integral value and two operations:
  - Down()/P(): Waits until value > 0 then decrements val =>(val is never negative)
  - Up()/V(): Increments val and picks a waiting thread (if any) and unblocks it, allowing it to complete its P operation
- If initial val is 1, then semaphore acts like a lock
  - Down = Acquire
  - Up = Release
- If initial val is 0, then semaphore acts like a CV
  - Down = Wait
  - Up = Signal
- Concern: Semaphore has state (where as CVs were memoryless) so a Up/V when no waiters exist will allow the next wait to immediately proceed
  - Can make reasoning about the value of a semaphore difficult
  - Requires programmer to map shared object state to semaphore count
- Generally prefer locks and CVs over semaphores for shared objects
- However semaphores can be used in specific places (especially in OSs)

Ensuring Mutual Exclusion

- How do we ensure atomic operation when implementing queueing locks, CVs, and semaphores
- Uniprocessor, in-Kernel
  - Can disable interrupts (only source of interleaving of memory access)
- Multiprocessor, in-Kernel
  - Need some kind of atomic locking instruction (i.e. TSL, Compare-and-swap) variable since disabling interrupts only applies to that one processor
    - Often use a spinlock
- Generally use both
  - Lock so that no other concurrent thread can update
  - Turn off interrupts so we quickly complete our code and don't get interrupted or context switched

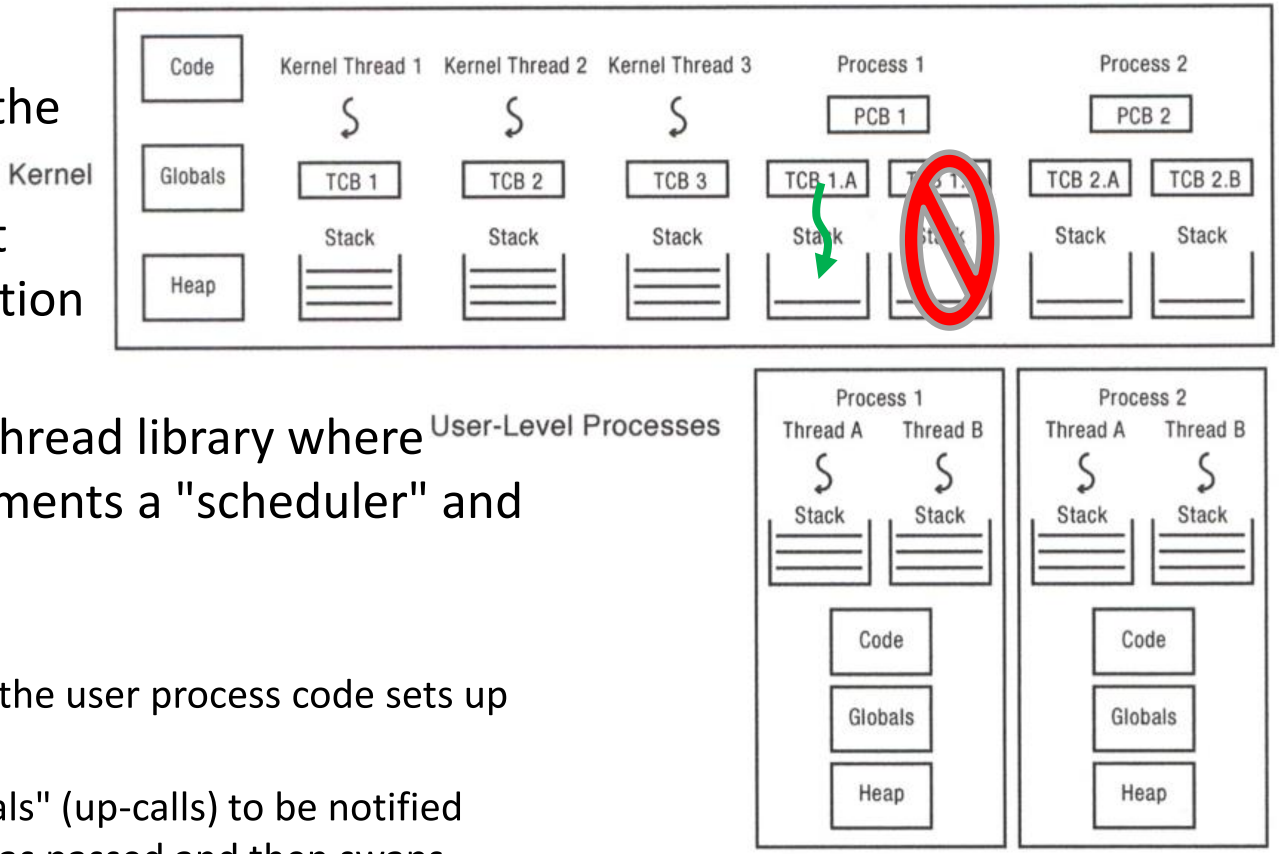
Tour Pintos

- Implements queueing locks and CVs in terms of semaphores
- Since it is uniprocessor, just disable interrupts

USER LEVEL THREAD LIBRARIES

User Level Thread Libraries

- Currently, user threads have to syscall/trap to the OS/kernel mode to perform thread context switch and synchronization
  - This takes time
- Can write a user-level thread library where the user process implements a "scheduler" and thread operations
  - 1 kernel thread
  - Many user threads that the user process code sets up and swaps between
  - User process uses "signals" (up-calls) to be notified when a time quantum has passed and then swaps user threads
  - Problem: When kernel thread gets desceduled all corresponding user threads get descheduled



User-Level Mutual Exclusion

- Can user level code disable interrupts to ensure mutual exclusion?
  - No, that is a privileged operation (only kernel can do that)
- Have to use some kind of atomic instruction (TSL, CAS, etc.)

Best Practices

GENERAL GUILDINES FOR WRITING SHARED OBJECTS

Recall Shared Objects

- **Shared Object (def.)**: An object that will be accessed by multiple threads
  - Should maintain state/shared data variables and the synchronization variable(s) needed to control access to them
- Methods should lock the object when updating shared state

```
class ObjA
{
  void f1(int newVal);

private:
  /* State vars */
  int sum;
  vector<int> vals;
  /* Synchronization var */
  Lock the_lock;
}

void ObjA::f1(int newVal)
{
  the_lock.acquire();
  vals.push_back(newVal);
  sum += newVal;
  the_lock.release();
}
```

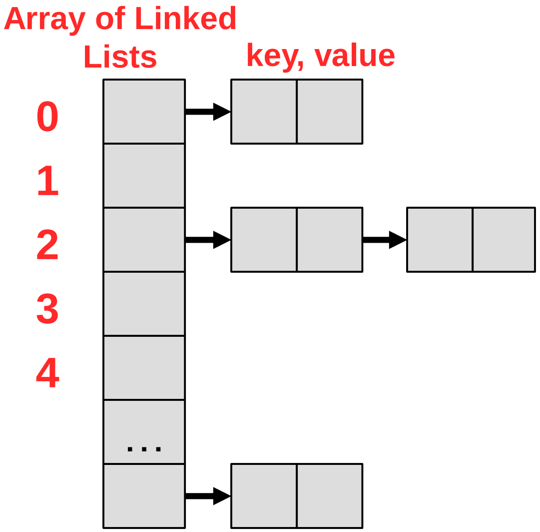
Guidelines For Shared Objects

- Decompose problem into shared objects. For each shared object allocate a lock. Lock when you enter, unlock before returning. Find out what conditions to wait for, an assigned a condition variable for each separate condition. Always use a while loop for the condition variable wait. Safe to always broadcast.
- Best practices:
  - Follow consistent design patterns, do not try to optimize
  - Always synch with locks and condition variables, not semaphores
  - Always acquire at the start of a method & release at the end
  - Condition variable: hold lock before wait, wait in while loop
  - Never use thread_sleep to wait for a condition

OTHER SYNCHRONIZATION PRIMITVIES

Reader/Write Locks

- Consider a shared data-structure like a hashtable (using chaining) supporting insert, remove, and find/lookup
  - We can't lookup while doing an insert or remove since the structures/pointers might be updated
  - Following our guidelines, we'd have a single lock to ensure mutual exclusion and just acquire the lock at the start of each member function (insert, remove, find)
  - Theoretically, can multiple find() operations run in parallel?
    - Yes, but if we lock at the start of find() we will preclude this and lower performance
  - We can safely have many readers, but only 1 writer at a time



Reader Write Locks

- Support many readers but only 1 writer
  - Description below "prioritizes" writers
- Operations:
  - startRead(): Waits if a current writer is active or another writer is already waiting, otherwise proceeds
  - doneRead(): If last reader, signals a waiting writer
  - startWrite(): Waits if a current write is active or 1 or more readers are active, otherwise proceeds
  - doneWrite(): If a waiting writer, signal it; otherwise broadcast/signal all waiting readers
- See OS:PP 2nd Ed. Figure 5.10