Recap: Synchronization

- **Critical sections**
  - Region of code that should execute atomically
- **Avoid race conditions via mutual exclusion**
- **Goal**: Broadly applicable simple solutions
  - Build upon atomic operations provided by hardware

Locks (mutexes)

- **Lock usage**
  - Initialized to free
  - Thread acquires lock before entering critical section (waiting if needed)
  - Thread that has acquired lock should release when done with critical section
- **Problem?**
  - Inefficient: Waiting to acquire lock

```
Peter
milk.lock();
if (noMilk) {
  buy milk
  milk.unlock()
}
```

```
Janet
milk.lock();
if (noMilk) {
  buy milk
  milk.unlock()
}
```

Efficiency

- Use lock to protect posting/looking up of note
  ```
  note.lock()
  if (noNote) {
    leave note
    note.unlock()
    buy milk
  }
  note.lock()
  remove note
  note.unlock()
  } else {
    note.unlock()
  }
  ```

Shared queue

```
struct node {
  int data
  struct node *next
}
```
```
struct queue {
  struct node *head
}
```

```
enqueue(new_element) {
  // find tail of queue
  for (ptr=head; ptr->next != NULL; ptr=ptr->next) {}
  // add new element to tail of queue
  ptr->next = new_element;
}
```
```
dequeue() {
  element = NULL;
  // if something on queue, then remove it
  if (head->next != NULL) {
    element = head->next;
    head->next = head->next->next;
  } else {
    return(element);
  }
```

```
problems if two threads manipulate queue at same time?
```

Shared queue with locks

```
enqueue(new_element) {
  lock:
  // find tail of queue
  for (ptr=queue; ptr->next != NULL; ptr=ptr->next) {}
  // add new element to tail of queue
  ptr->next = new_element;
  unlock();
}
```
```
dequeue() {
  node lock()
  if (element != NULL) {
    element = NULL;
    // if something on queue, then remove it
    if (head->next != NULL) {
      element = head->next;
      head->next = head->next->next;
    }
  }
  unlock();
  return(element);
}
```
Invariants for thread-safe queue

- When can enqueue() unlock?
  - Must restore queue to a stable state
- Stable state is called an invariant
  - Condition that is "always" true for the linked list
  - Example: each node appears exactly one when traversing from head to tail
- Is invariant ever allowed to be false?
  - Hold lock whenever you’re manipulating shared data, i.e., whenever you’re breaking the invariant
  - What if you’re only reading the data?

Don’t break assumptions

enqueue(new_element) {
    lock
    // find tail of queue
    for (ptr=head; ptr->next != NULL; ptr = ptr->next) {}  
    unlock
    lock
    // add new element to tail of queue
    ptr->next = new_element;
    unlock
}

Does this work?

Fine-grained locking

- Instead of one lock for entire queue, use one lock per node
  - Why would you want to do this?
- Lock each node as the queue is traversed, then release as soon as it’s safe, so other threads can also access the queue

What bad thing could occur?

How to fix?

- lock A
- get pointer to B
- lock B
- read B
- unlock B

How to fix?

- lock A
- get pointer to B
- lock B
- unlock A
- read B
- unlock B

- Hand-over-hand locking
  - Lock next node before releasing last node
  - Used in Project 4
  - What assumption are we making here?

Ordering constraints

- What if you wanted dequeue() to wait if the queue is empty?

dqueue() {
    queue.lock();
    // wait for queue to be non-empty
    queue.unlock();
    while(head->next == NULL) {
        queue.lock();
        // remove element
        element = head->next;
        head->next = head->next->next;
        queue.unlock();
        return(element);
    }
}

Does this work?

Ordering constraints

dqueue() {
    queue.lock();
    // wait for queue to be non-empty
    while(head->next == NULL) {
        queue.unlock();
        queue.lock();
    }
    // remove element
    element = head->next;
    head->next = head->next->next;
    queue.unlock();
    return(element);
}

Does this work?
Avoiding busy waiting

- Have waiting dequeuer "go to sleep"
  - Put dequeuer onto a waiting list, then go to sleep
    
    ```
    if (queue is empty) {
      add myself to waiting list
      go to sleep
    }
    
    enqueuer wakes up sleeping dequeuer
    ```

- locker wakes up sleeping dequeuer

Does this work?

Two types of synchronization

- Mutual exclusion
  - Ensures that only one thread is in critical section
  - "Not at the same time"
  - lock/unlock

- Ordering constraints
  - Used when thread must wait for another thread to do something
  - "Before after"
  - E.g., dequeuer must wait for enqueuer to add something to queue

Condition variables

- Enable a thread to sleep inside a critical section, by
  - Releasing lock
  - Putting thread onto waiting list
  - Going to sleep
  - After being woken, call lock()

- Each condition variable has a list of waiting threads
  - These threads are "waiting on that condition"
  - Each condition variable is associated with a lock

Operations on condition variables

- wait()
  - Atomically release lock, add thread to waiting list, go to sleep
  - Thread must hold the lock when calling wait()
  - Should thread re-establish invariant before calling wait()?

- signal()
  - Wake up one thread waiting on this condition variable
  - If no thread is currently waiting, then signal does nothing

- broadcast()
  - Wake up all threads waiting on this condition variable
  - If no thread is currently waiting, then broadcast does nothing

Thread-safe queue with no busy waiting

```java
    enqueuer()
    lock
    find tail of queue
    add new element to tail of queue
    if (dequeuer is waiting) {
      take waiting dequeuer off waiting list
      wake up dequeuer
    }
    unlock
    dequeue()
    ...
    if (queue is empty) {
      add myself to waiting list
      sleep
    }
    ...
```
Thread-safe queue with condition variables

cv queueCV;
enqueue()
queueMutex.lock()
find tail of queue
add new element to tail of queue
queueCV.signal()
queueMutex.unlock();
}
dequeue()
queueMutex.lock()
if (queue is empty) {
queueCV.wait();
}
remove item from queue
queueMutex.unlock()
return removed item

Does this work?

unlock
put thread on wait queue
go to sleep
re-acquire lock

Thread-safe queue with condition variables

cv queueCV;
enqueue()
queueMutex.lock()
find tail of queue
add new element to tail of queue
queueCV.signal()
queueMutex.unlock();
}
dequeue()
queueMutex.lock()
while (queue is empty) {
queueCV.wait();
}
remove item from queue
queueMutex.unlock()
return removed item