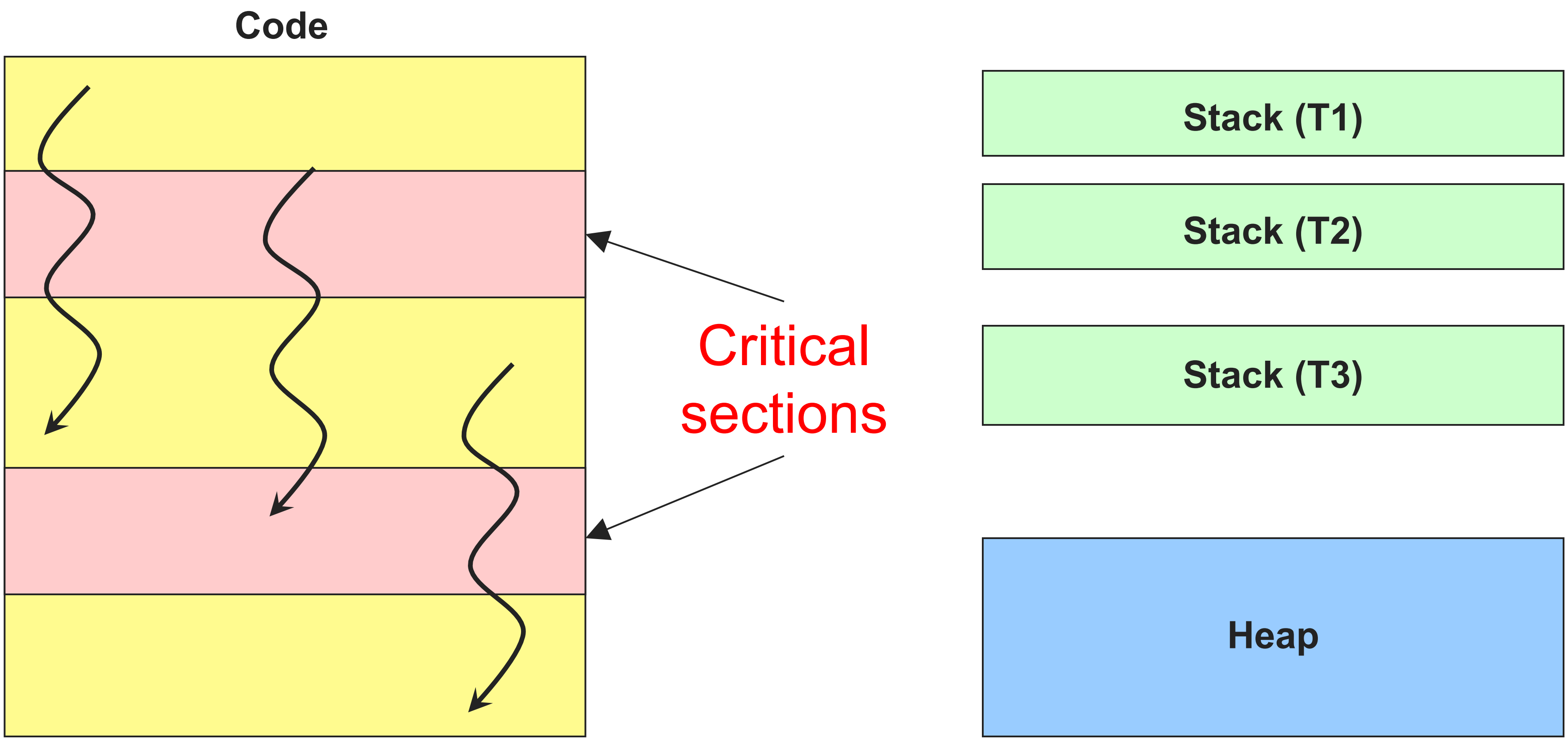
EECS 482
Introduction to Operating Systems

Winter 2018

Harsha V. Madhyastha

Recap: Synchronization

Avoid race conditions via mutual exclusion



“Too much milk” Takeaways

- Feasible to synchronize via loads/stores, but …
  - Complex
  - Busy waiting

Want simple to use and efficient solutions that are broadly applicable

Bob

```
leave noteBob
while (noteAlice) {
    do nothing
}
if (noMilk) {
    buy milk
}
remove noteBob
```

Alice

```
leave noteAlice

if (no noteBob) {
    if (noMilk) {
        buy milk
    }
}
remove noteAlice
```

"Too much milk" Takeaways

- Locks simplify mutual exclusion
  - Acquire lock at start of critical section
  - Release lock at end of critical section
- No busy waiting in user-level code

Bob

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

Alice

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

"Too much milk" Takeaways

- For efficiency, minimize size of critical section

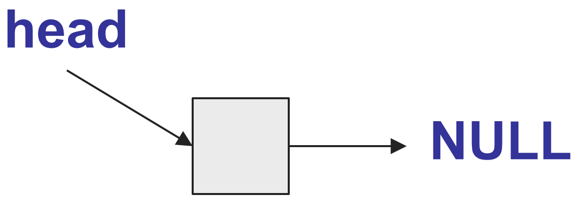
```
note.lock()
if (noNote) {
    leave note
    note.unlock()
    if (noMilk) {
        buy milk
    }
    note.lock()
    remove note
    note.unlock()
}
else {
    note.unlock()
}
```

Shared queue

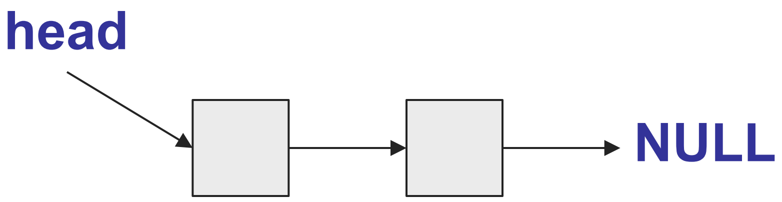
```
struct node {
    int data
    struct node *next
}

struct queue {
    struct node *head
}
```

- Empty list



- List with one node

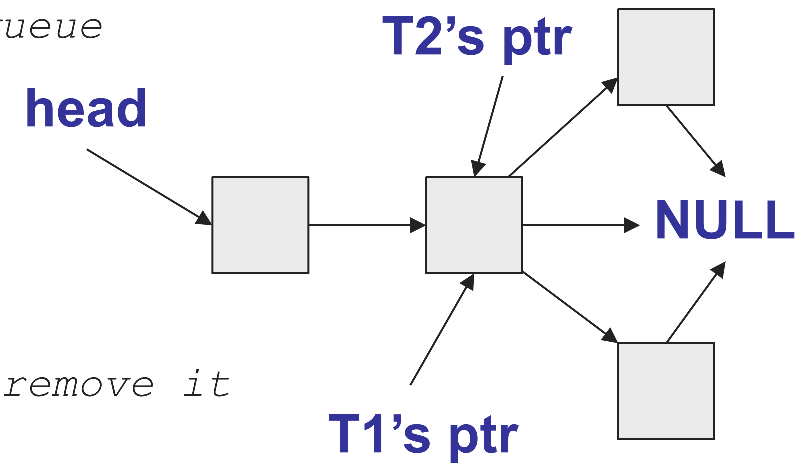


Shared queue

```
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;
    new_element->next = NULL;
}

dequeue() {
    element = NULL;
    // if something on queue, then remove it
    if (head->next != NULL) {
        element = head->next;
        head->next = head->next->next;
    }
    return(element);
}
```



Problems if two threads manipulate queue at same time?

Shared queue with locks

```
enqueue(new_element) {
    qmutex.lock();
    // find tail of queue
    for (ptr=head; ptr->next != NULL; ptr = ptr->next) {}

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

Thread-safety invariants

- 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 once
- 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?

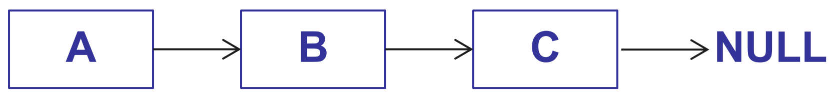
Don’t break assumptions

```
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;
    new_element->next = NULL;
    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



1. lock A
2. get pointer to B
3. unlock A
4. lock B
5. read B
6. unlock B

Another thread could lock A and dequeue all nodes

What problems could occur?
How to fix?

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

Announcements

- Attempt homework questions before discussion section on Friday
- Group declaration due on Monday (Jan 22)
- Read handout for Project 1 and tried autograder?
  - After today's lecture, we'll have covered all material to do the project

Ordering constraints

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

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

    qmutex.unlock();
    return(element);
}
```

Does this work?

Ordering constraints

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

Avoiding busy waiting

```
enqueue()
    lock
    add new item to tail of queue
    if (dequeuer is waiting) {
        take waiting dequeuer off waiting list
        wake up dequeuer
    }
    unlock
```

Does this work?

```
dequeue()
    lock
    if (queue is empty) {
  -->   add myself to waiting list
  -->   sleep
    }
    remove item from queue
    unlock
```

Two types of synchronization

- Mutual exclusion ⟵ Locks
  - Ensures that only one thread is in critical section
  - *"Not at the same time"*
  - lock/unlock
- Ordering constraints ⟵ Condition variables
  - One thread waits for another to do something
  - *"Before after"*
  - E.g., dequeuer must wait for enqueuer to add something to queue

Condition variables

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

  atomic

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

Condition variables interface

- wait(mutex)
  - Atomically release lock, add thread to waiting list, 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
- broadcast()
  - Wake up all threads waiting on this condition variable
- If no thread is waiting, signal/broadcast does nothing

Thread-safe queue with no busy waiting

```
enqueue()
    lock
    add new element to tail of queue
    if (dequeuer is waiting) {
        take waiting dequeuer off waiting list
        wake up deqeuer
    }
    unlock
dequeue()
    lock
    if (queue is empty) {
        add myself to waiting list
        sleep
    }
    remove item from queue
    unlock
    return removed item
```

Thread-safe queue with condition variables

```
cv queueCV;
enqueue() {
    queueMutex.lock()
    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
}
```

This is broken!

queueCV.wait():

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

(unlock, put thread on wait queue, go to sleep: atomic)

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
}
```

Always use while!

Conditional variables eliminate busy waiting

```
lock
. . .
while (queue is empty) {
    unlock
    lock
}
. . .
unlock
```

```
lock
. . .
while (queue is empty) {
    cv.wait
}
. . .
unlock
```

Monitors

- Combine two types of synchronization
  - Locks for mutual exclusion
  - Condition variables for ordering constraints

- A monitor = a lock + the condition variables associated with that lock

Mesa vs. Hoare monitors

- Mesa monitors
  - When waiter is woken, it must contend for the lock
  - So it must re-check the condition it was waiting for
- What would be required to ensure condition is met when waiter starts running again?
- Hoare monitors
  - Special priority to woken-up waiter
  - Signaling thread immediately gives up lock
  - Signaling thread reacquires lock after waiter unlocks

Mesa vs. Hoare monitors

- Mesa monitors
  - When waiter is woken, it must contend for the lock

We (and most OSes) use Mesa monitors

Waiter is solely responsible for ensuring condition is met

- Hoare monitors
  - Special priority to woken-up waiter
  - Signaling thread immediately gives up lock
  - Signaling thread reacquires lock after waiter unlocks

Programming with monitors

1. List the shared data needed for the problem
2. Assign locks to each group of shared data
   - » Tradeoff between complexity and concurrency
3. List the before-after conditions for the problem
4. Assign condition variable to each condition
   - » Associate with lock protecting data used for condition
5. Add lock/unlock around all accesses to shared data
   - » Remember invariant
6. Add while (!cond) { wait } where condition must hold
7. Add signal/broadcast after possibly making cond true

Typical monitor code

```
lock
while (!condition) {
    wait
}

do stuff

signal about the stuff you did
unlock
```

Project 1

- Now, you should know everything you need to know to do project 1
- Due in 12 days (on Jan 29th)