EECS 482
Introduction to Operating Systems

Winter 2018

Baris Kasikci

Slides by: Harsha V. Madhyastha

Recap: Synchronization

- Critical sections
  - Region of code that should execute atomically
- Avoid data races 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
- Solution?

Baris

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

Tia

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

Efficiency

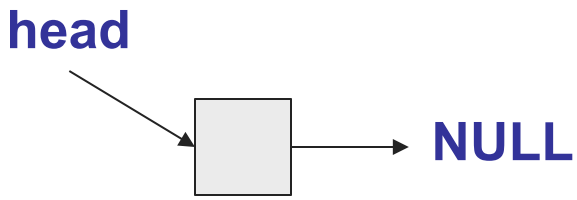
- Use lock to protect posting/looking up of note

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

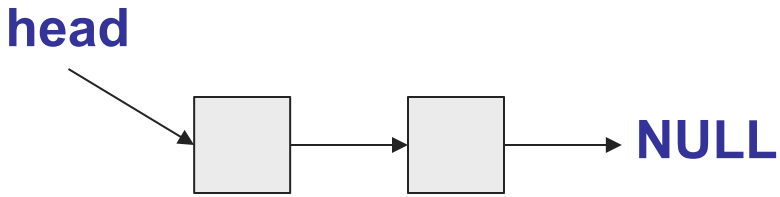
Shared queue

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

struct queue {
     struct node *head
}
```

- Empty list

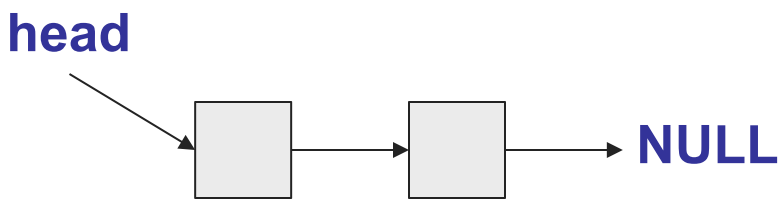


- List with one node



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;
}

struct node 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);
}
```

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: in a list traversal, nodes appears exactly once, and in the order they were inserted
- 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?

Hold the lock!

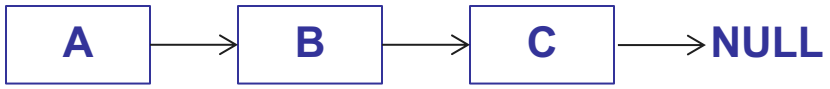
```
enqueue(new_element) {
    lock
    // find tail of queue
    for (ptr=head; ptr->next != NULL; ptr = ptr->next) {}
    unlock

    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. read A, get pointer to B
3. unlock A
4. lock B
5. read B, get pointer to C
6. unlock B

What could go wrong?
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

- Group declaration due in 5 days (Jan 22nd)

- Read handout for Project 1?
  - 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
    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()
    lock
    if (queue is empty) {
        add myself to waiting list
        sleep
    }
    unlock
```

Does this work?

Avoiding busy waiting

```
enqueue()
    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()
    lock
    if (queue is empty) {
        unlock
        add myself to waiting list
        sleep
    }
    unlock
```

Does this work?

Avoiding busy waiting

```
enqueue()
    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()
    lock
    if (queue is empty) {
        add myself to waiting list
        unlock
        sleep
    }
    unlock
```

Does this work?

Two types of synchronization

- Mutual exclusion ← Locks
  - Ensures that only one thread is in critical section
  - *"Not at the same time"* relations between threads
  - lock/unlock
- Ordering constraints ← Condition variables
  - Used when thread must wait for another thread to do something
  - *"Happens-before"* relations between threads
  - 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

  atomic
  - 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

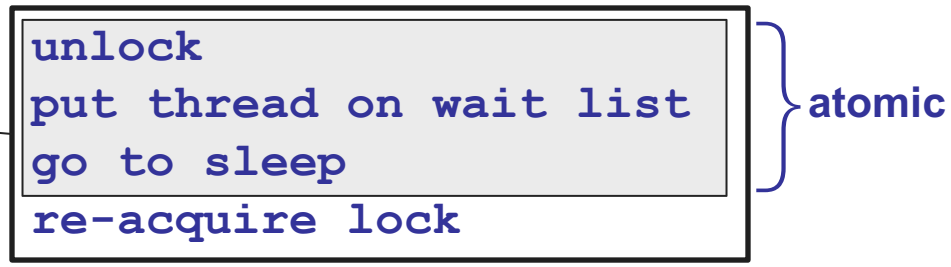
Operations on condition variables

- wait()
  - Atomically release lock, add thread to waiting list, 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 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?



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
}
```

When can you use if?

!!!Never!!! due to spurious wakeups!

The prophet
Emin Gün Sirer

Archangel
Michael Dahlin

In the beginning, there was hardware…

And the hardware was formless and empty,
And darkness was over the surface of silicon.

And the creator said “let there be
operating systems” and there were OSes.
And the creator saw that OSes were good.

And the creator said “let there be processes
and threads.” And OSes were teeming with

In the beginning, there was hardware…

And the hardware was formless and empty,
And darkness was over the surface of silicon.

And the creator said “let there be operating systems” and there were OSes.
And the creator saw that OSes were good.

And the creator said “let there be processes and threads.” And OSes were teeming with

A problem has been detected and Windows has been shut down to prevent damage to your computer.

PFN_LIST_CORRUPT

If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

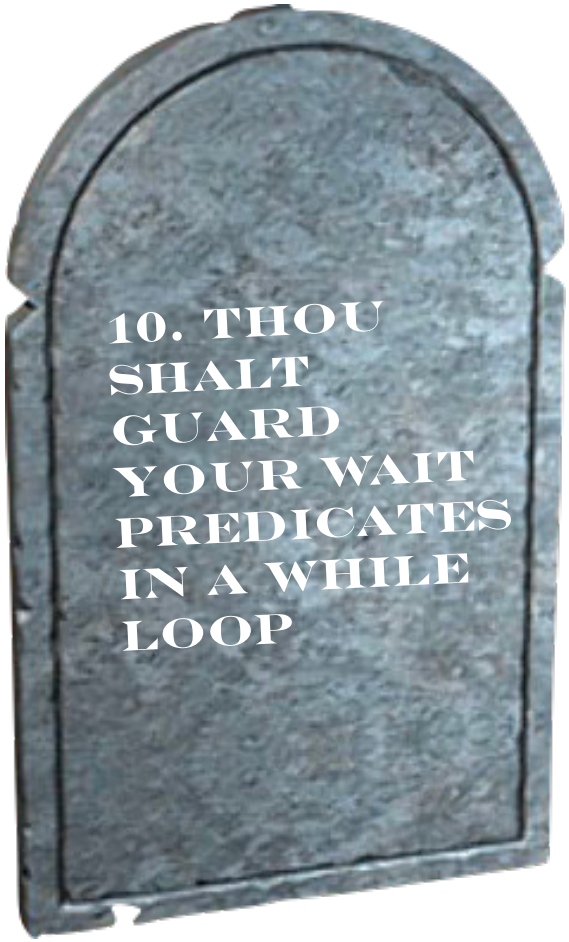
Technical information:
*** STOP: 0x0000004e (0x00000099, 0x00900009, 0x00000900, 0x00000900)

Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further assistance.

And the creator said “let there be processes and threads.” And OSes were teeming with processes and threads carrying out different tasks.
Then the creator said “let the processes and threads synchronize with each other.” For this task, the creator appointed human-kind. But humans were fallible, and weak, and they failed to get synchronization correct, and the fallen angel BSOD, ruled the day with great evil.

So the creator sent us the 12 commandments of synchronization.

The tenth commandment



Conditional variables eliminate busy waiting

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

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

Project 1

- Now, you should know everything you need to know to do project 1

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 up, it competes for the lock

We (and most OSes) use Mesa monitors

Waiter is solely responsible for ensuring condition is met

  - 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

How to program with monitors

- List the shared data needed for the problem
- Assign locks to each group of shared data
- Each thread tries to go as fast as possible, without worrying about other threads, except for two reasons

- Mutual exclusion: Enforce with lock/unlock
- Ordering conditions
  - » Can't proceed because condition of shared state isn't satisfactory
  - » Some other thread must do something
  - » Assign a condition variable for each situation
    - Belongs to lock that protects the shared data used to evaluate the condition
  - » Use "while(!condition) { wait }"
  - » Call signal() or broadcast() when a thread changes something that another thread might be waiting for

Typical way to program with monitors

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

do stuff

signal about the stuff you did
unlock
```