EECS 482
Introduction to Operating Systems

Winter 2018

Harsha V. Madhyastha

Recap

- Two types of synchronization
  - Mutual exclusion → Locks
  - Ordering constraints → Condition variables

- Condition variables: Enable a thread to sleep inside a critical section
  - Release lock
  - Put thread onto waiting list
  - Go to sleep
  - After being woken, call lock()

atomic (applies to: Release lock; Put thread onto waiting list; Go to 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()
    while (queue is empty) {
        queueCV.wait();
    }
    remove item from queue
    queueMutex.unlock()
    return removed item
}
```

Monitors

- Combine two types of synchronization
  - A lock + condition variables associated with that lock

```
lock
while (!condition) {
    wait
}
do stuff
signal about the stuff you did
unlock
```

Producer-consumer (bounded buffer)

- Producers fill a shared buffer; consumers empty it
- Need to synchronize actions of producers and consumers

- Used in many situations
  - Unix pipes (grep “keyword” foo.txt | wc -l)
  - Project 1!
  - Coke machine
- Why use a shared buffer?
  - Lets producers and consumers operate somewhat independently

Coke machine with monitors

- Step 1: Identify shared state
  - State of coke machine
  - *numCokes*
  - What about MAX = capacity of coke machine?
- Step 2: Assign locks
  - One lock (*cokeLock*) to protect all shared data

Coke machine with monitors

- Step 3: Identify before-after conditions
  - Before coke purchase, at least 1 coke in machine
  - Before adding 1 coke, at least 1 empty slot
- Step 4: Assign condition variables
  - Consumer waits on *waitingConsumers* if all slots are empty
  - Producer waits on *waitingProducers* if all slots are full

Coke machine with monitors

Consumer

```
cokeLock.lock()

while (numCokes == 0) {
   waitingConsumers.wait()
}

take coke out of machine
numCokes--

waitingProducers.signal()

cokeLock.unlock()
```

Producer

```
cokeLock.lock()

while (numCokes == MAX) {
    waitingProducers.wait()
}

add coke to machine
numCokes++

waitingConsumers.signal()

cokeLock.unlock()
```

Wait-signal pairing

Consumer

```
cokeLock.lock()

while (numCokes == 0) {
   waitingConsumers.wait()
}

take coke out of machine
numCokes--

waitingProducers.signal()

cokeLock.unlock()
```

Producer

```
cokeLock.lock()

while (numCokes == MAX) {
    waitingProducers.wait()
}

add coke to machine
numCokes++

waitingConsumers.signal()

cokeLock.unlock()
```

Looping while holding lock

<u>Consumer</u>

```
cokeLock.lock()

while (numCokes == 0) {
   waitingConsumers.wait()
}

take coke out of machine
numCokes--

waitingProducers.signal()

cokeLock.unlock()
```

<u>Producer</u>

```
cokeLock.lock()

while (1) {
   sleep(1 hour)
   while (numCokes == MAX) {
      waitingProducers.wait()
   }

   add coke to machine
   numCokes++

   waitingConsumers.signal()
}

cokeLock.unlock()
```

Reducing number of signals

Consumer

```
cokeLock.lock()

while (numCokes == 0) {
   waitingConsumers.wait()
}

take coke out of machine
numCokes--

waitingProducers.signal()

cokeLock.unlock()
```

Producer

```
cokeLock.lock()

while (numCokes == MAX) {
    waitingProducers.wait()
}

add coke to machine
numCokes++

if (numCokes == 1) {
    waitingConsumers.signal()
}

cokeLock.unlock()
```

Reducing number of signals

- numCokes = 0
- C1 and C2 waiting on waitingConsumers
- P1 increments numCokes to 1 and signals
- P2 increments numCokes to 2, but does not signal
- Only one of C1 and C2 may wake up!

Reducing condition variables

Consumer

```
cokeLock.lock()

while (numCokes == 0) {
    waitingCons&Prod.wait()
}

take coke out of machine
numCokes--

waitingCons&Prod.signal()

cokeLock.unlock()
```

Producer

```
cokeLock.lock()

while (numCokes == MAX) {
    waitingCons&Prod.wait()
}

add coke to machine
numCokes++

waitingCons&Prod.signal()

cokeLock.unlock()
```

Reducing condition variables

- Say MAX = 1, and numCokes = 0
- C1 and C2 wait

- P1 increments numCokes to 1 and signals
  - Wakes up C1
- P2 waits because numCokes = MAX

- C1 decrements numCokes to 0 and signals
  - May wake up C2!

Need broadcast due to condition variable reuse

Consumer

```
cokeLock.lock()

while (numCokes == 0) {
   waitingCons&Prod.wait()
}

take coke out of machine
numCokes--

waitingCons&Prod.broadcast()

cokeLock.unlock()
```

Producer

```
cokeLock.lock()

while (numCokes == MAX) {
    waitingCons&Prod.wait()
}

add coke to machine
numCokes++

waitingCons&Prod.broadcast()

cokeLock.unlock()
```

Announcements

- Group declaration due today

- Started with Project 1?
  - Due in a week

- Be aware of object lifetimes
- Avoid using the following:
  - broadcast()
  - signal() inside while loop around wait()

```
while (!condition) {
    cv.signal()
    cv.wait()
}
```

Reader-writer locks

- Recall: Threads need to lock to read shared data
  - Implication: No concurrent reads!
- How to safely allow more concurrency?

- Problem definition:
  - Shared data will be read and written by multiple threads
  - Allow multiple readers, if no threads are writing data
  - A thread can write only when no other thread is reading or writing

Need for reader-writer locks

- Use of normal mutex locks limits concurrency

Reader:
```
lock()
print catalog
unlock()
```

Writer:
```
lock()
change catalog
unlock()
```

Reader-writer locks

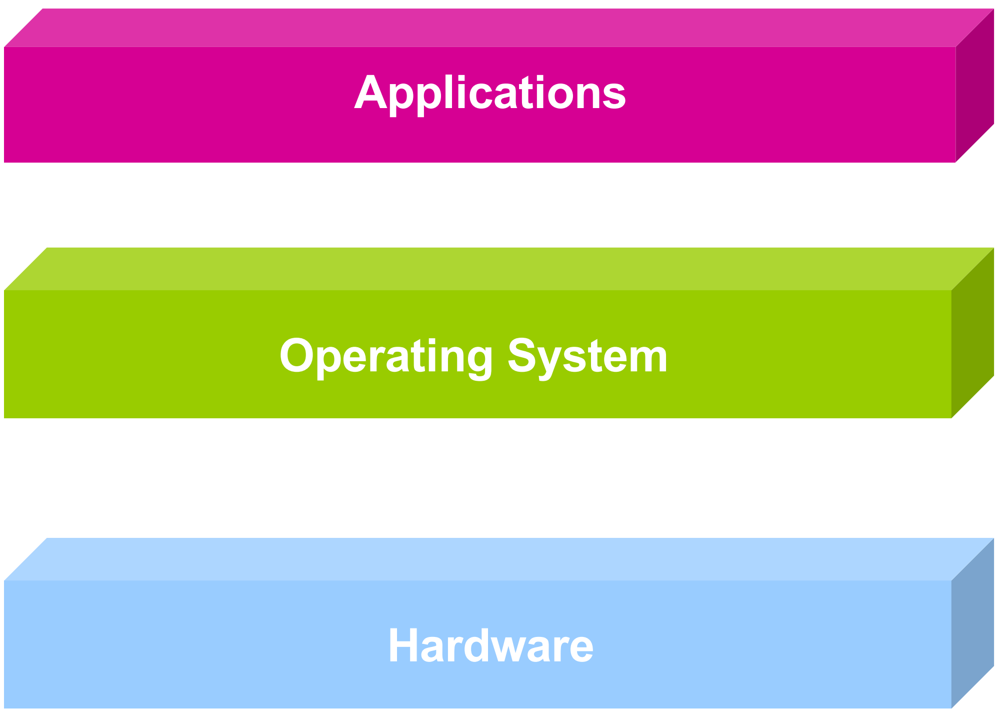
- Implement set of functions that a program can use to follow *“multiple-reader, single-writer”* constraint
  - readerStart()
  - readerFinish()
  - writerStart()
  - writerFinish()

```
Reader:
readerStart()
print catalog
readerFinish()

Writer:
writerStart()
change catalog
writerFinish()
```

- Pros and cons compared to normal mutex locks?

Another level of abstraction



Even higher-level synchronization primitives

Concurrent programs

(readerStart, readerFinish, writerStart, writerFinish)

Higher-level synchronization primitives

(lock, monitor, semaphore)

Atomic operations

(load/store, interrupt enable/disable, test&set)

R/W lock with monitors

- Step 1: What state is shared?
  - *numReaders*
  - *numWriters*
- Step 2: Assign locks to shared state
  - *rwLock*
- Step 3: What are the before-after conditions?
  - readers must wait if thread is writing
  - writers must wait if thread is reading or writing
- Step 4: Assign condition variables
  - *waitingReaders, waitingWriters*

R/W lock with monitors

```
readerStart () {
    rwLock.lock()
    while (numWriters > 0) {
        waitingReaders.wait()
    }
    numReaders++
    rwLock.unlock()
}
readerFinish() {
    rwLock.lock()
    numReaders--
    waitingWriters.signal()
    rwLock.unlock()
}
```

```
writerStart() {
    rwLock.lock()
    while (numReaders > 0 || numWriters > 0) {
        waitingWriters.wait()
    }
    numWriters++
    rwLock.unlock()
}
writerFinish() {
    rwLock.lock()
    numWriters--
    waitingReaders.broadcast()
    waitingWriters.signal()
    rwLock.unlock()
}
```

R/W lock with monitors

```
readerStart () {
    rwLock.lock()
    while (numWriters > 0) {
        waitingReaders.wait()
    }
    numReaders++
    rwLock.unlock()
}
readerFinish() {
    rwLock.lock()
    numReaders--
    if (numReaders == 0) {
        waitingWriters.signal()
    }
    rwLock.unlock()
}
```

```
writerStart() {
    rwLock.lock()
    while (numReaders > 0 || numWriters > 0) {
        waitingWriters.wait()
    }
    numWriters++
    rwLock.unlock()
}
writerFinish() {
    rwLock.lock()
    numWriters--
    waitingReaders.broadcast()
    waitingWriters.signal()
    rwLock.unlock()
}
```

R/W lock with monitors

```
readerStart () {
    rwLock.lock()
    while (numWriters > 0) {
        waitingReaders.wait()
    }
    numReaders++
    rwLock.unlock()
}
readerFinish() {
    rwLock.lock()
    if (numReaders == 1) {
        waitingWriters.signal()
    }
    numReaders--
    rwLock.unlock()
}
```

```
writerStart() {
    rwLock.lock()
    while (numReaders > 0 || numWriters > 0) {
        waitingWriters.wait()
    }
    numWriters++
    rwLock.unlock()
}
writerFinish() {
    rwLock.lock()
    numWriters--
    waitingReaders.broadcast()
    waitingWriters.signal()
    rwLock.unlock()
}
```

R/W lock with monitors

- What will happen if a writer finishes and there are several waiting readers and writers?
  - Will writerStart return, or will 1 readerStart return, or will all readerStart return?
- How long will a writer wait?
- How to give priority to a waiting writer?

Prioritizing waiting writers

```
readerStart () {
    rwLock.lock()
    while (numWriters > 0 ||
        numWaitingWriters > 0) {
        waitingReaders.wait()
    }
    numReaders++
    rwLock.unlock()
}
```

```
writerStart() {
    rwLock.lock()
    numWaitingWriters++
    while (numReaders + numWriters > 0) {
        waitingWriters.wait()
    }
    numWaitingWriters--
    numWriters++
    rwLock.unlock()
}
```

Programming with monitors

- Make sure to try homework questions for this Friday's lab section

- Key challenges in monitor programming:
  - Adding more locks (deadlock!)
  - Enforcing ordering/preventing starvation