EECS 482
Introduction to Operating Systems

Winter 2018

Baris Kasikci

Slides by: 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 by
  - Releasing lock
  - Putting thread onto waiting list
  - Going to sleep
  - After being woken, call lock()

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
}
```

Operations on condition variables

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

- signal()
  - Wake up one thread waiting on this condition variable

- broadcast()
  - Wake up all threads waiting on this condition variable
  - When is this useful?

Thread-safe queue with condition variables

```
cv queueCV;
enqueue(set of elements)
    queueMutex.lock()
    find tail of queue
    add new elements to tail of queue
    queueCV.broadcast()
    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
  - Locks for mutual exclusion
  - Condition variables for ordering constraints

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

Producer-consumer (bounded buffer)

- Producers put things into a shared buffer; consumers take them out
- Need to synchronize actions of producers and consumers

- Why use a shared buffer?
  - Lets producers and consumers operate somewhat independently
- Used in many situations
  - Unix pipes
  - Project 1!
  - Coke vending machine

Producer-consumer with monitors

- Shared variables
  - State of coke machine slots
    - » *numCokes* (assume coke machine can hold at most MAX cokes)
  - One lock (*cokeLock*) to protect this data
- When must a thread wait?
  - Mutual exclusion (when acquiring a lock)
  - Consumer must wait if all slots are empty
    - » Use condition variable *waitingConsumers*
  - Producer must wait if all slots are full
    - » Use condition variable *waitingProducers*

Producer-consumer 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()
```

Producer-consumer with monitors

Consumer

```
cokeLock.lock()

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

take coke out of machine
numCokes--

waitingProducers.signal()

cokeLock.unlock()
```

Producer

```
cokeLock.lock()

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

   add coke to machine
   numCokes++

   waitingConsumers.signal()
}

cokeLock.unlock()
```

Producer-consumer 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++

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

cokeLock.unlock()
```

Producer-consumer with monitors

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()
```

Producer-consumer with monitors

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()
```

Producer-consumer 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()
```

Announcements

- Started with Project 1?
  - Due in a week
- Group declaration due today
  - Project 2 will be posted on the Sunday 29th of Jan

Reader-writer locks

- Recall: Threads need to acquire lock even to read shared data
  - This prevents other threads from accessing the data
- Can we allow more concurrency without risking reading unstable data?
- 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

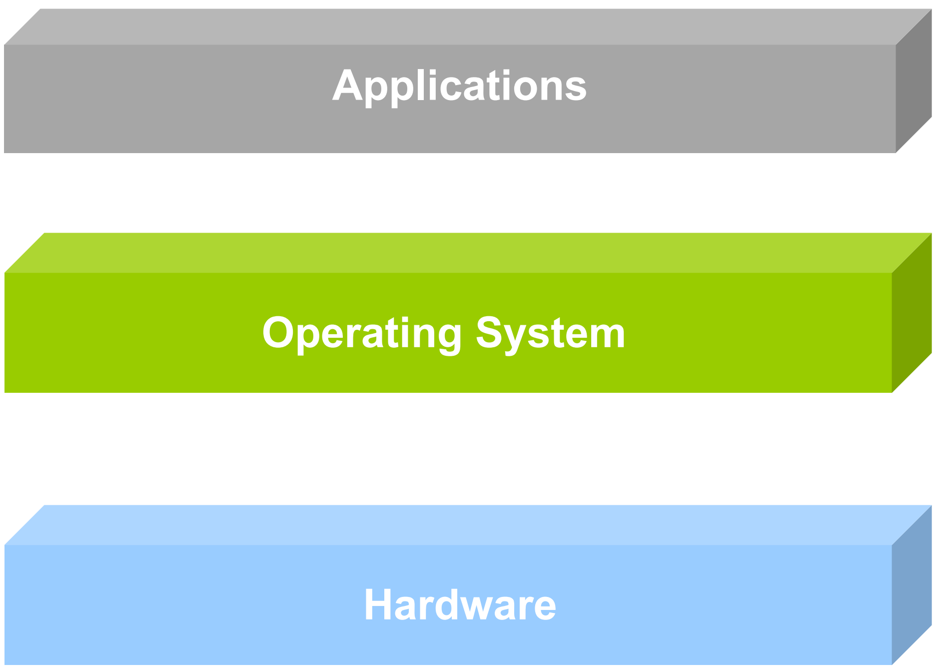
- Implement set of functions that a program can use to follow *"multiple-reader, single-writer"* paradigm
  - readerStart()
  - readerFinish()
  - writerStart()
  - writerFinish()

```
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)

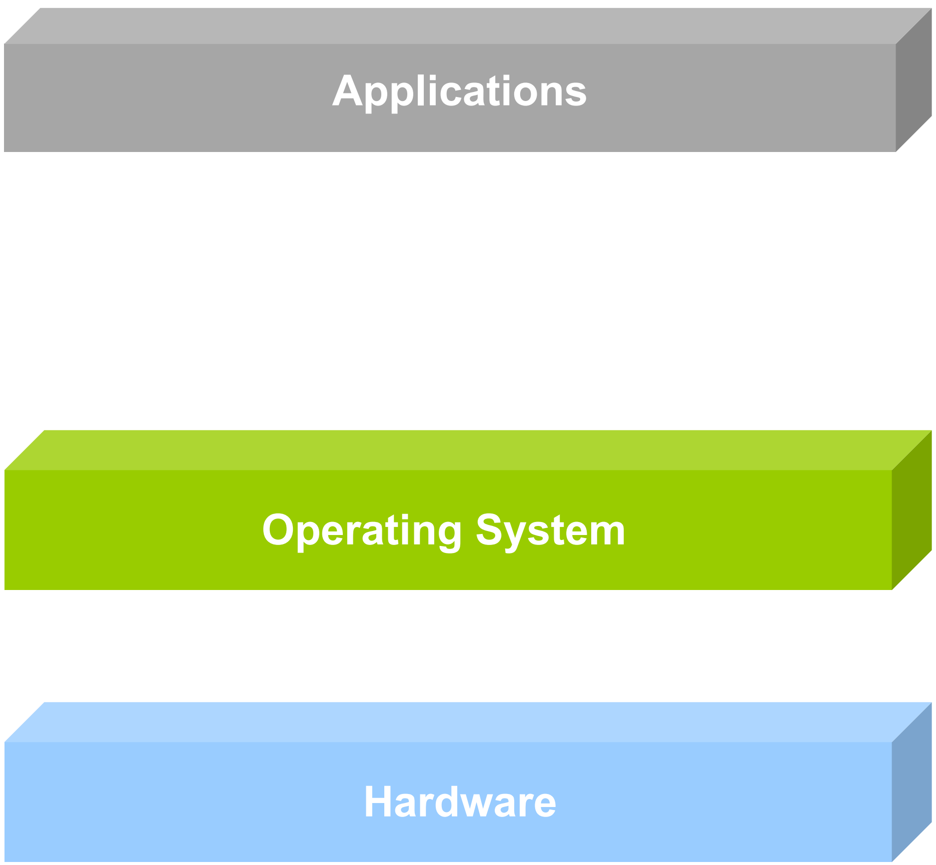
Higher-level synchronization primitives

(lock, monitor, semaphore)

Atomic operations

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

Another level of abstraction



Concurrent programs

Even higher-level synchronization primitives
(readerStart, readerFinish, writerStart, writerFinish)

Higher-level synchronization primitives
(lock, monitor, semaphore)

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

Implementing reader-writer locks with monitors

- Shared data needed to implement readerStart, readerFinish, writerStart, writerFinish?
  - *numReaders*
  - *numWriters*
- Use one lock (*sLock*)
- Condition variables?
  - *waitingReaders*: readers must wait if there are writers
  - *waitingWriters*: writers must wait if there are readers or writers

Implementing reader-writer locks with monitors

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

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