EECS 482
Introduction to Operating Systems

Winter 2019

Manos Kapritsos

Thanks to Harsha Madhyastha and Peter Chen for the slides and notes
Implementing reader-writer locks with monitors

- Shared data needed to implement readerStart, readerFinish, writerStart, writerFinish?
  - `numReaders`
  - `numWriters`
- Use one lock (`rwLock`)?
- 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

```java
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()
}
```
Implementing reader-writer locks 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()
}
Implementing reader-writer locks with monitors

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

readerFinish() {
    rwLock.lock()
    numReaders--
    if (numReaders == 0) {
        waitingWriters.signal()
        waitingReaders.broadcast()
    }
    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()
}
```
Implementing reader-writer locks with monitors

- What will happen if a writer finishes and there are several waiting readers and writers?
  - Either 1 writer will start, or all readers will start

- How long will a writer wait?
  - Potentially forever (starvation)
Recap

● Multi-threaded code must synchronize access to shared data

● High-level synchronization primitives:
  ◆ **Locks**: Mutual exclusion
  ◆ **Condition variables**: Ordering constraints
  ◆ **Monitors**: Lock + condition variables

● Today: **Semaphores**
Semaphores

- Generalized lock/unlock
- Definition:
  - A non-negative integer (initialized to user-specified value)
  - down(): wait for semaphore value to become positive, then atomically decrement semaphore value by 1
    
    ```
    do {
      if (value > 0) {
        value--; // Atomic
        break
      }
    } while (1)
    ```
  
  - up(): increment semaphore value by 1
Two types of semaphores

- **Mutex semaphore (or binary semaphore)**
  - Represents single resource (critical section)
  - Guarantees mutual exclusion

- **Counting semaphore (or general semaphore)**
  - Represents a resource with many units, or a resource that allows concurrent access (e.g., reading)
  - Multiple threads can “hold” the semaphore
    - Number determined by the semaphore “count”
Benefit of Semaphores

- Mutual exclusion
  - Initial value is 1
  down()
critical section
up()

- Ordering constraints
  - Usually, initial value is 0
  - Example: thread A wants to wait for thread B to finish

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>down()</td>
<td>do task</td>
</tr>
<tr>
<td>continue execution</td>
<td>up()</td>
</tr>
</tbody>
</table>
Implementing producer-consumer with semaphores

- Semaphore assignments
  - *mutex*: ensures mutual exclusion around code that manipulates coke machine
  - *fullSlots*: counts no. of full slots in the coke machine
  - *emptySlots*: counts no. of empty slots in machine

- Initialization values?
  - mutex=1, fullSlots=0, emptySlots=N
Implementing producer-consumer with semaphores

Semaphore mutex = 1; // mutual exclusion to shared set of slots
Semaphore emptySlots = N; // count of empty slots (all empty to start)
Semaphore fullSlots = 0;  // count of full slots (none full to start)

**producer** {
    // wait for empty slot
    emptySlots.down();

    mutex.down();
    *Add coke to the machine*
    mutex.up();

    // note a full slot
    fullSlots.up();
}

**consumer** {
    // wait for full slot
    fullSlots.down();

    mutex.down();
    *Take coke out of machine*
    mutex.up();

    // note an empty slot
    emptySlots.up();
}
Implementing producer-consumer with semaphores

- Why do we need different semaphores for fullSlots and emptySlots? **Semaphores wait at 0.**
- Does the order of down() matter? **Yes**
- Does the order of up() matter? **No**
- What if there’s 1 full slot, and multiple consumers call down() at the same time?
- What if a context switch happens between emptySlots.down() and mutex.down()?
- What if fullSlots.up() before mutex.down()?
Implementing producer-consumer with semaphores

Semaphore mutex = 1; // mutual exclusion to shared set of slots
Semaphore emptySlots = N; // count of empty slots (all empty to start)
Semaphore fullSlots = 0; // count of full slots (none full to start)

```c
producer {
    // wait for empty slot
    emptySlots.down();

    mutex.down();
    Add coke to the machine
    mutex.up();
}

consumer {
    // wait for full slot
    fullSlots.down();

    mutex.down();
    Take coke out of machine
    mutex.up();
}
```

// note a full slot
fullSlots.up();

// note an empty slot
emptySlots.up();
Reminders

- Project 1 due today
- Project 2 out! Read the spec **carefully**!
- A few stragglers
- Semaphore lab moved to 2/8
Comparing monitors and semaphores

- Semaphores provide 1 mechanism that can accomplish both mutual exclusion and ordering (monitors use different mechanisms for each)
  - Elegant
  - Can be difficult to use

- Monitor lock = binary semaphore (initialized to 1)
  - lock() = down()
  - unlock() = up()
## Condition variable versus semaphore

<table>
<thead>
<tr>
<th>Condition variable</th>
<th>Semaphore</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(!cond) {wait();}</td>
<td>down()</td>
</tr>
<tr>
<td>Can safely handle spurious wakeups</td>
<td>No spurious wakeups</td>
</tr>
<tr>
<td>Conditional code in user program</td>
<td>Conditional code in semaphore definition</td>
</tr>
<tr>
<td>User writes customized condition; more flexible</td>
<td>Condition specified by semaphore definition (wait if value == 0)</td>
</tr>
<tr>
<td>User provides shared variable; protects with lock</td>
<td>Semaphore provides shared variable (integer) and thread-safe operations on that variable (down, up)</td>
</tr>
<tr>
<td>No memory of past signals</td>
<td>Remembers past up calls</td>
</tr>
</tbody>
</table>

T1: wait()  
T2: signal()  
T3: signal()  
T4: wait()  

T1: down()  
T2: up()  
T3: up()  
T4: down()
Implementing custom waiting condition with semaphores

- Semaphores work best if the shared integer and waiting condition (value==0) map naturally to problem domain

- How to implement custom waiting condition with semaphores?
Producer-consumer with monitors

**Consumer**

```java
cokeLock.lock()

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

take coke out of machine
numCokes--

waitingProducers.signal()

cokeLock.unlock()
```

**Producer**

```java
cokeLock.lock()

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

add coke to machine
numCokes++

waitingConsumers.signal()

cokeLock.unlock()
```
Producer-consumer with semaphores (monitor style)

**Consumer**

```java
mutex.down()
while (numCokes == 0) {
    go to sleep
}
take coke out of machine
numCokes--

wake up waiting producer, if any
mutex.up()
```

**Producer**

```java
mutex.down()
while (numCokes == MAX) {
    go to sleep
}
add coke to machine
numCokes++

wake up waiting consumer, if any
mutex.up()
```
Producer-consumer with semaphores (monitor style)

**Consumer**

```c
mutex.down()
while (numCokes == 0) {
    semaphore s = 0
    waitingConsumers.push(&s)

    s.down()
}
take coke out of machine
numCokes--
if (!waitingProducers.empty()) {
    waitingProducers.front()->up()
    waitingProducers.pop()
}
mutex.up()
```

**Producer**

```c
mutex.down()
while (numCokes == MAX) {
    semaphore s = 0
    waitingProducers.push(&s)

    s.down()
}
add coke to machine
numCokes++
if (!waitingConsumers.empty()) {
    waitingConsumers.front()->up()
    waitingConsumers.pop()
}
mutex.up()
```
Producer-consumer with semaphores (monitor style)

**Consumer**

```c
mutex.down()
while (numCokes == 0) {
  semaphore s = 0
  waitingConsumers.push(&s)
  mutex.up()
  s.down()
  mutex.down()
}
take coke out of machine
numCokes--
if (!waitingProducers.empty()) {
  waitingProducers.front()->up()
  waitingProducers.pop()
}
mutex.up()
```

**Producer**

```c
mutex.down()
while (numCokes == MAX) {
  semaphore s = 0
  waitingProducers.push(&s)
  mutex.up()
  s.down()
  mutex.down()
}
add coke to machine
numCokes++
if (!waitingConsumers.empty()) {
  waitingConsumers.front()->up()
  waitingConsumers.pop()
}
mutex.up()
```
The first commandment

Would you name your kid “Kid”? Or “KidA”? Or “MyKid”? Or “k”? 
The third commandment

We gave you monitors so you don’t have to worship the ancient gods!
This is NOT OK:
while(true) {
    mutex.lock()
    if(!condition) {
        mutex.unlock()
        break;
    } else {
        mutex.unlock()
        sleep(200);
    }
}
The sixth commandment

6. All shared state must be protected
myAtomicFunction() {
  mutex.lock()
  ...
  ...
  ...
  mutex.unlock()
}