EECS 482
Introduction to Operating Systems

Winter 2018

Baris Kasikci

Slides by: Harsha V. Madhyastha
Producer-consumer with monitors

**Consumer**

```java
cokeLock.lock()

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

take coke out of machine
numCokes--

waitingCons&Prod.signal()

cokeLock.unlock()
```

**Producer**

```java
cokeLock.lock()

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

add coke to machine
numCokes++

waitingCons&Prod.signal()

cokeLock.unlock()
```
Producer-consumer with monitors

**Consumer**

```java
cokeLock.lock()
while (numCokes == 0) {
    waitingCons&Prod.wait()
}
take coke out of machine
numCokes--
waitingCons&Prod.signal()
cokeLock.unlock()
```

**Producer**

```java
cokeLock.lock()
while (numCokes == MAX) {
    waitingCons&Prod.wait()
}
add coke to machine
numCokes++
waitingCons&Prod.signal()
cokeLock.unlock()
```

**Time**

- **MAX = 1, numCokes = 0**
  - **consumer-1** → *Waiting (lock released)*
  - **consumer-2** → *Waiting (lock released)*
  - **producer-1** → numCokes = 1 (lock held)
    - Signal
    - Wakes up consumer-1 (can’t grab lock)
    - Return from producer (lock released)
  - **producer-2** → numCokes = 1 (lock held)
    - Can’t add coke (MAX = 1)
    - Waiting (lock released)
  - **consumer-1** → numCokes = 0 (lock held)
    - Signal (goal is to wake up a producer)
    - Wakes up consumer-2
    - Return from consumer (lock released)
  - **consumer-2** → numCokes = 0 (lock held)
    - Waiting (lock released)

Both producer-2 and consumer-2 are waiting!
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++

if (numCokes == 1) {
    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()

**Time**

```
numCokes = 0

consumer-1 → Waiting (lock released)
consumer-2 → Waiting (lock released)
producer-1 → numCokes = 1 (lock held)
            Signal consumer
            Return from producer (lock released)

Consumers do not acquire the lock

producer-2 → numCokes = 2 (lock held)
            No Signal (numCokes == 1 fails)
            Return from producer (lock released)

Only one consumer will wake up
```
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) {
        waitingReaders.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()
}
```
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()

---

**Time**

- numCokes = 0

- consumer-1 \(\rightarrow\) *Waiting (lock released)*
- consumer-2 \(\rightarrow\) *Waiting (lock released)*
- producer-1 \(\rightarrow\) numCokes = 1 (lock held)  
  *Signal consumer*  
  *Return from producer (lock released)*

**Consumers do not acquire the lock**

- producer-2 \(\rightarrow\) numCokes = 2 (lock held)  
  *No Signal (numCokes == 1 fails)*  
  *Return from producer (lock released)*

**Only one consumer will wake up**
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) {
        waitingReaders.broadcast()
        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

```java
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) {
        waitingReaders.broadcast()
        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

- What will happen if a writer finishes and there are several waiting readers and writers?

- How long will a writer wait?

- How to give priority to a waiting writer?
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
    ```java
    do {
      if (value > 0) {
        value--
        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 (e.g., Coke machine), 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

```
Time

Thread A
down()

critical section

continue execution

Thread B

down()

do task

up()
```
Implementing producer-consumer with semaphores

- Semaphore assignments

  - \textit{mutex}: ensures mutual exclusion around code that manipulates coke machine
  
  - \textit{fullSlots}: counts no. of full slots in the coke machine
  
  - \textit{emptySlots}: counts no. of empty slots in machine

- Initialization values?
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();
}

c 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?
- Does the order of down() (e.g., fullSlots, mutex) matter?
- Does the order of up() matter?
- 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()?
Reminders

● Project 1 due on Monday

● Work through discussion questions about monitors before the discussion section
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>
# 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>Semaphore provides shared variable (integer) and thread-safe operations on that variable (down, up)</td>
</tr>
<tr>
<td>User provides shared variable; protects with lock</td>
<td>Remembers past up calls</td>
</tr>
<tr>
<td>No memory of past signals</td>
<td></td>
</tr>
</tbody>
</table>

- **T1**: wait()
- **T2**: signal()
- **T3**: signal()
- **T4**: wait()
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
mutex.down()
while (numCokes == 0) {

go to sleep
}
take coke out of machine
numCokes--

Producer
mutex.down()
while (numCokes == MAX) {

go to sleep
}
adde coke to machine
numCokes++

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

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”?

1. Thou shalt name thy synchronization variables properly.
The third commandment

3. THOU SHALT USE MONITORS INSTEAD OF SEMAPHORES WHENEVER POSSIBLE

We gave you monitors so you don’t have to worship the ancient gods!
The fifth commandment

5. THOU SHALL NOT BUSY-WAIT

This is NOT OK:

```java
while(true) {
    mutex.lock()
    if(condition) {
        mutex.unlock()
        break;
    } else {
        mutex.unlock()
        sleep(200);
    }
}
```
The sixth commandment

6. All shared state must be protected
The seventh commandment

```c
myAtomicFunction() {
    mutex.lock()
    ...
    ...
    ...
    mutex.unlock()
}
```
IRQL\_NOT\_DISPATCH\_LEVEL

If this is the first time you've seen this 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. 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: 0x00000ed (0x80f128d0, 0xc000009c, 0x00000000, 0x00000000)

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


For technical support assistance call : 1-855-596-2695 (USA-Canada)
The seventh commandment

myAtomicFunction() {
  mutex.lock()
  ...
  ...
  ...
  ...
  mutex.unlock()
}

7. Thou shalt grab the monitor lock upon entry to, and release it upon exit from a procedure.