EECS 482
Introduction to Operating Systems

Winter 2018

Harsha V. Madhyastha
Recap

- Multi-threaded code with monitors:
  - Locks for mutual exclusion
  - Condition variables for ordering constraints

- Every thread is in one of the following states:
  - Running outside any critical section
  - Waiting on a mutex to enter a critical section
  - Running inside a critical section
  - Waiting on a cv inside a critical section
Semaphores

- Generalized lock/unlock
- Definition:
  - A **non-negative integer** (initialized to user-specified value)
  - `down()`: wait for semaphore value to become positive, then 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)
  - Up() atomically sets value to 1

- **Counting semaphore** (or **general semaphore**)
  - Represents a resource with many units, or a resource that allows concurrent access (e.g., reading)
  - Multiple threads can up/down the semaphore
Benefit of Semaphores

- Mutual exclusion
  
  ```
  semaphore sem(1)
  sem.down()
  critical section
  sem.up()
  ```

- Ordering constraints
  
  - Example: thread A wants to wait for thread B to finish
    ```
    semaphore sem(0)
    ```

  Thread A
  ```
  sem.down()
  continue execution
  ```

  Thread B
  ```
  do task
  sem.up()
  ```
Coke machine with semaphores

- As before, think about shared data, mutual exclusion, and before-after relations
- Assign semaphore for each:
  - **mutex**: for exclusive access to coke machine
  - **fullSlots**: before removing a coke, cokes > 0
    » Counts filled slots in machine
  - **emptySlots**: before adding a coke, cokes < MAX
    » Counts free spaces in the machine
Coke machine with semaphores

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

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

    mutex.down();
    Add coke out of 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();
}
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()
Coke machine with semaphores

- What if there’s 1 full slot, and multiple consumers call `down()` at the same time?
- Why do we need different semaphores for `fullSlots` and `emptySlots`?
- Does the order of `down()` matter?
- Does the order of `up()` matter?
- What if a context switch between `emptySlots.down()` and `mutex.down()`?
- What if `fullSlots.up()` before `mutex.down()`?
Reminders

- Project 1 due on Monday
  - Project 2 will be out the same day

- Comfortable with gdb?

- Work through homework questions about monitors before Friday’s lab section
Readers/Writers with Semaphores

- Use three variables
  - integer `readcount` – number of threads reading
  - Semaphore `mutex` – control access to `readcount`
  - Semaphore `w_or_r` – write-mode or read-mode
Readers/Writers with Semaphores

// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// write-mode or read-mode
Semaphore w_or_r = 1;

writer {
    w_or_r.down();
    Write;
    w_or_r.up();
}

reader {
    mutex.down();
    readcount++;
    if (readcount == 1)
        w_or_r.down();
    mutex.up();
    Read;
    mutex.down();
    readcount--;
    if (readcount == 0)
        w_or_r.up();
    mutex.up();
}
Readers/Writers with Semaphores

- Why don’t writers use mutex?
- If a writer is writing, where will readers be waiting?
- Once a writer exits,
  - Which reader gets to go first?
  - Is it guaranteed that all readers will fall through?
- What if mutex.up() is above “if (readcount == 1)”?
- If read in progress when writer arrives, when can writer get access?
Monitors vs. Semaphores

- Semaphores: 1 mechanism for both mutual exclusion and ordering
  - 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; more flexible</td>
<td>Conditional code in 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; more flexible</td>
<td>Conditional code in semaphore definition</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>

### Example

**Condition variable**

1. T1: `wait()`
2. T2: `signal()`
3. T3: `signal()`
4. T4: `wait()`

**Semaphore**

1. T1: `down()`
2. T2: `up()`
3. T3: `up()`
4. 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
}
```

```java
take coke out of machine
numCokes--
```

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

**Producer**

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

```java
add coke to machine
numCokes++
```

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

**Consumer**
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**
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()
Exercise to try ...

- Convert monitor-style reader/writer lock implementation to use semaphores