Reminders

- Project 1 due on Tuesday
- Work through discussion questions about monitors before tomorrow’s discussion section

Recap

- Multi-threaded code must synchronize access to shared data
- High-level synchronization primitives:
  - Locks
  - Condition variables
  - Monitors
- 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-- 
          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 pass 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
    - Thread A: `down()`
    - Thread B: `do task`
    - Thread A: `continue execution`
    - Thread A: `up()`
Implementing producer-consumer with semaphores

- Semaphore assignments
  - `mutex`: ensures mutual exclusion around code that manipulates coke machine
  - `fullSlots`: counts the number of full slots in the coke machine
  - `emptySlots`: counts the number of empty slots in the coke machine
- Initialization values?

```c
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();
}
```

Implementing producer-consumer with semaphores

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

Readers/Writers with Semaphores

- Use three variables
  - integer `readcount` – number of threads reading
  - Semaphore `mutex` – control access to `readcount`
  - Semaphore `w_or_r` – exclusive writing or reading

```c
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
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();
    Read;
    w_or_r.up();
  }
  mutex.up();
}```
Readers/Writers with Semaphores

- 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?
- If readers and writers are waiting, and a writer exits, who goes first?
- What if mutex.up() is above "if (readcount == 1)?
- If read in progress when writer arrives, when can writer get access?

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>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>

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

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 semaphores (monitor style)

Consumer

mrules.down()
while (numCokes == 0) {
  semaphore s = 0
  waitingConsumers.push(&s)
mrules.up()
s.down()
  mutex.lock()
  s.down()
  mutex.unlock()
}
take coke out of machine
numCokes--
waitingProducers.signal()

Producer

mrules.down()
while (numCokes == MAX) {
  semaphore s = 0
  waitingProducers.push(&s)
mrules.up()
s.down()
  mutex.lock()
  s.down()
  mutex.unlock()
}
add coke to machine
numCokes++
waitingConsumers.push(&numCokes)

if (!waitingProducers.empty()) {
  waitingProducers.pop()
}
mrules.up()