Last Class: Synchronization

- Synchronization
  - Mutual exclusion
  - Critical sections
- Locks

- Synchronization primitives are required to ensure that only one thread executes in a critical section at a time.
Clicker Question #1

What will data be at the end of these two threads? (assume data=0 and is on the heap or a global)

(A) 0

(B) 1

(C) -1

(D) Any of the above

(E) None of the above

THREAD 1
lock();
a = data;
a++;
data = a;
unlock();

THREAD 2
lock();
b = data;
b--;
data = b;
unlock();
Answer on Next Slide
Today: Semaphores

• What are semaphores?
  
  • Semaphores are basically generalized locks.
  
  • Like locks, semaphores are a special type of variable that supports two atomic operations and offers elegant solutions to synchronization problems.
  
  • They were invented by Dijkstra in 1965.
Semaphores

- **Semaphore**: an integer variable that can be updated only using two special atomic instructions.

- **Binary (or Mutex) Semaphore**: (same as a lock)
  - Guarantees mutually exclusive access to a resource (only one process is in the critical section at a time).
  - Can vary from 0 to 1, is initialized to free (value = 1)

- **Counting Semaphore**:
  - Useful when multiple units of a resource are available
  - The initial count to which the semaphore is initialized is usually the number of resources.
  - A process can acquire access so long as at least one unit of the resource is available
Semaphores: Key Concepts

- Like locks, a semaphore supports two atomic operations, Semaphore.Wait() and Semaphore.Signal().

- Each semaphore supports a queue of processes that are waiting to access the critical section (e.g., to buy milk).

- If a process executes `S.Wait()` and semaphore S is free (non-zero), it continues executing. If semaphore S is not free, the OS puts the process on the wait queue for semaphore S.

- A `S.Signal()` unblocks one process on semaphore S's wait queue.

```c
S.Wait(); // wait until semaphore S
// is available
<critical section>
S.Signal();
// signal to other processes
// that semaphore S is free
```
Semaphores: Example

Locks:
Lock.Acquire();
if (noMilk){
    buy milk;
}
Lock.Release();

Semaphores:
Semaphore.Wait();
if (noMilk){
    buy milk;
}
Semaphore.Signal();
Implementing Signal + Wait

class Semaphore {
    public:
        void Wait(Process P);
        void Signal();
    private:
        int value;
        Queue Q;   // queue of processes;
}
Semaphore(int val) {
    value = val;
    Q = empty;
}

Wait(Process P) {
    value = value - 1;
    if (value < 0) {
        add P to Q;
        P->block();
    }
}

Signal() {
    value = value + 1;
    if (value <= 0){
        remove P from Q;
        wakeup(P);
    }
}

=> Signal and Wait of course must be atomic!

- Use interrupts or test&set to ensure atomicity
Counting Semaphore

P1:
S.Wait();
S.Wait();
S.Wait();
S.Signal();
S.Signal();

P2:
S.Wait();
S.Signal();

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>empty</td>
<td>execute</td>
<td>execute</td>
</tr>
</tbody>
</table>

CS377: Operating Systems
Using Semaphores

• **Mutual Exclusion**: used to guard critical sections
  
  - the semaphore has an initial value of 1
  
  - `S->Wait()` is called before the critical section, and `S->Signal()` is called after the critical section.

• **Scheduling Constraints**: used to express general scheduling constraints where threads must wait for some circumstance.
  
  - The initial value of the semaphore is usually 0 in this case.
  
  - **Example**: You can implement thread join (or the Unix system call `waitpid(PID)`) with semaphores:

    ```
    Semaphore S;
    S.value = 0; // semaphore initialization
    Thread.Join:       Thread.Finish:
    S.Wait();          S.Signal();
    ```
Multiple Consumers and Producers

class BoundedBuffer {
public:
    void Producer();
    void Consumer();
private:
    Items buffer;
    Semaphore mutex;
    Semaphore empty;
    Semaphore full;
}
BoundedBuffer::BoundedBuffer(int N){
    mutex.value = 1;
    empty.value = N;
    full.value = 0;
    new buffer[N];
}
BoundedBuffer::Producer(){
    <produce item>
    empty.Wait(); // one fewer slot, or wait
    mutex.Wait(); // get access to buffers
    <add item to buffer>
    mutex.Signal(); // release buffers
    full.Signal(); // one more used slot
}
BoundedBuffer::Consumer(){
    full.Wait(); // wait until there's an item
    mutex.Wait(); // get access to buffers
    <remove item from buffer>
    mutex.Signal(); // release buffers
    empty.Signal(); // one more free slot
    <use item> }

Multiple Consumers and Producers Problem

<table>
<thead>
<tr>
<th></th>
<th>empty</th>
<th>full</th>
</tr>
</thead>
<tbody>
<tr>
<td>initially</td>
<td>⬤⬤⬤⬤</td>
<td>○○○○</td>
</tr>
<tr>
<td><strong>Producer 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty-&gt;wait();</td>
<td>⬤⬤⬤〇</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>full-&gt;signal();</td>
<td></td>
<td>⬤〇〇〇</td>
</tr>
<tr>
<td><strong>Producer 2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty-&gt;wait();</td>
<td>⬤⬤〇〇</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>full-&gt;signal();</td>
<td>⬤⬤〇〇</td>
<td></td>
</tr>
<tr>
<td><strong>Consumer</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>full-&gt;wait();</td>
<td></td>
<td>⬤〇〇〇</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty-&gt;signal();</td>
<td>⬤⬤⬤〇</td>
<td></td>
</tr>
</tbody>
</table>

BoundedBuffer::Producer()

  <produce item>
  empty.Wait();
  mutex.Wait();
  <add item to buffer>
  mutex.Signal();
  full.Signal();

BoundedBuffer::Consumer()

  full.Wait();
  mutex.Wait();
  <remove item from buffer>
  mutex.Signal();
  empty.Signal();
  <use item> }
Clicker Question #2

What happens if we only use one semaphore for full/empty?

(A) Works fine

(B) All threads stop running

(C) Items get lost

(D) Items get duplicated

(E) None of the above

BoundedBuffer::Producer()

<produce item>
full.Wait();
mutex.Wait();
<add item to buffer>
mutex.Signal();
full.Signal();
}

BoundedBuffer::Consumer()

full.Wait();
mutex.Wait();
<remove item from buffer>
mutex.Signal();
full.Signal();
<use item>  

What's wrong w/ Semaphores?

• Semaphores have the following drawbacks.
  – They are essentially shared global variables.
  – There is no linguistic connection between the semaphore and the data to which the semaphore controls access.
  – Access to semaphores can come from anywhere in a program.
  – They serve two purposes, mutual exclusion and scheduling constraints.
  – There is no control or guarantee of proper usage.

• **Solution**: use a higher level primitive called monitors
Summary

• Semaphores are a generalization of locks

• Semaphores can be used for three purposes:
  
  – To ensure mutually exclusive execution of a critical section (as locks do).
  
  – To control access to a shared pool of resources (using a counting semaphore).
  
  – To cause one thread to wait for a specific action to be signaled from another thread.