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.

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
  - It 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().

```
S.Wait()        // wait until semaphore S
                // is available
<critical section>

S.Signal()      // signal to other processes
                // that semaphore S is free
```

- 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.

Binary Semaphores: Example

- Too Much Milk using locks:

```
Thread A                      Thread B

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

- Too Much Milk using semaphores:

```
Thread A                      Thread B

Semaphore.Wait();             Semaphore.Wait();
if (noMilk){                  if (noMilk){
  buy milk;                         buy milk;
}                             }
Semaphore.Signal();           Semaphore.Signal();
```


Implementing Signal and 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

Signal and Wait: Example

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

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

P1: S->Wait();
P2: S->Wait();
P1: S->Wait();
P2: S->Signal();
P1: S->Signal();
P1: S->Signal();

		process state: execute or block	
value	Queue	P1	P2
2	empty	execute	execute

Signal and Wait: Example

P1: S->Wait();
P2: S->Wait();
P1: S->Wait();
P1: S->Signal();
P2: S->Signal();
P1: S->Signal();

value	Queue	P1	P2
2	empty	execute	execute

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;
 // control access to buffers
     Semaphore mutex;
    // count of free slots
     Semaphore empty;
    // count of used slots
     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

	empty	full
initially	● ● ● ●	○ ○ ○ ○
Producer 1 empty->wait();	● ● ● ○	
... full->signal();		● ○ ○ ○
Producer 2 empty->wait();	● ● ○ ○	
... full->signal();		● ● ○ ○
Consumer full->wait();		● ○ ○ ○
... empty->signal();	● ● ● ○	

Summary

- Locks can be implemented by disabling interrupts or busy waiting
- 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.