Hardware Support for Synchronization — Last Time

	Concurrent Programs			
Low-level atomic operations (hardware)	load/store	interrupt disable	test&set	
High-level atomic operations (API)	locks	semaphores	monitors	send&receive

Semaphores — Today

- 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.

Two types of Semaphores

A Binary (or Mutex) Semaphore: is the same as a lock. It guarantees mutually exclusive access to a resource (only one process is in the critical section at a time). It is initialized to free (value $= 1$).

A Counted Semaphore: represents a resource with many units available. The initial count to which the semaphore is initialized is usually the number of resources. A counted semaphore lets a process continue as long as more instances are available.

Semaphores

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, it continues executing. If semaphore S is not free, the OS puts the process on the wait queue for semaphore S.
- S->Signal() unblocks *one* process on semaphore S's wait queue.

Simple 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();
```

Counted Semaphores

A Counted Semaphore: represents a resource with many units available. The initial count to which the semaphore is initialized is usually the number of resources (value=n). A counted semaphore lets processes continue as long as more instances are available.

```
S->Wait()             // Decrement value.
                      // If value < 0, wait on queue.
<critical section>

S->Signal()           // Increment value.
                      // If there is a waiter on the
                      // queue, wake it up.
```

Example: For each thread, the OS keeps track of the open files. In order to make the PCB a fixed size, the OS only allows each process to open a maximum of say 16 files. If the OS wants to support a file protocol that forces the process to wait when the maximum is exceeded, we can write:

```
Semaphore->value = 16;   // initialization

Kernel::OpenFile(...) {
  Semaphore->Wait();
  ...
}
Kernel::CloseFile(...) {
  ...
  Semaphore->Signal();
}
```


Implementing Signal and Wait

```
class Semaphore {
  public:
    void Wait(Process P);
    void Signal();
  private:
   int value;
   Queue Q;    // queue of processes;
}
Semaphore::Semaphore(int val) {
   value = val;  // This ini-
tial value depends
   Q = empty;    // on the problem.
}
Semaphore::Wait(Process P) {
   value = value - 1;
   if (value < 0) {
      add P to Q;
      P->block();
}  }
Semaphore::Signal() {
   value = value + 1;
   if (value <= 0){
      remove P from Q;
      wakeup(P);
}  }
```

$\Longrightarrow$ Signal and Wait of course must be atomic!

Signal and Wait

Example:

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

	value	Queue	process state: execute or block P1	P2
	2	empty	execute	execute
P1: S->Wait();				
P2: S->Wait();				
P1: S->Wait();				
P1: S->Signal();				
P2: S->Signal();				
P1: S->Signal();				

	value	Queue	P1	P2
	1	empty	execute	execute
P1: S->Wait();				
P2: S->Wait();				
P1: S->Wait();				
P1: S->Signal();				
P2: S->Signal();				
P1: S->Signal();				

Using Semaphores

1. **Mutual Exclusion** - as in the too much milk problem, the semaphore has an initial value of 1, and S->Wait() is called before the critical section, and S->Signal is called after the critical section.

2. **Scheduling Constraints** - Semaphores can be 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
    S->Wait();

Thread::Finish
    S->Signal();
```

Multiple Consumers and Producers

```
class BoundedBuffer {
   public:
     void Producer();  void Consumer();
   private:
     Items *buffer;
     Semaphore mutex; // control access to shared buffer
     Semaphore empty; // count of empty slots
     Semaphore full;  // count of full slots
}
BoundedBuffer::BoundedBuffer(int N) {
     mutex->value = 1;
     empty->value = N;
     full->value  = 0;
     new buffer[N];
}
BoundedBuffer::Producer() {
   <produce item>
   empty->Wait();     // one fewer buffer, or wait
   mutex->Wait();     // get access to buffer pointers
   <add item to buffer>
   mutex->Signal();   // release pointers
   full->Signal();    // note one more item in buffer
}
BoundedBuffer::Consumer() {
   full->Wait();      // wait until there's an item
   mutex->Wait()      // get access to buffer pointers
   <remove item from buffer>
   mutex->Signal();   // release pointers
   empty->Signal();   // count down
   <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

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.

Next Time

More on how to use Semaphores.