Last Class: Synchronization

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

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

Today: Synchronization: Locks and Semaphores

- More on hardware support for synchronization
- Implementing locks using test&set and busy waiting
- 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.

Atomic read-modify-write Instructions

- Atomic read-modify-write instructions *atomically* read a value from memory into a register and write a new value.
  - Straightforward to implement simply by adding a new instruction on a uniprocessor.
  - On a multiprocessor, the processor issuing the instruction must also be able to *invalidate* any copies of the value the other processes may have in their cache, i.e., the multiprocessor must support some type of *cache coherence*.
- **Examples:**
  - **Test&Set:** (most architectures) read a value, write '1' back to memory.
  - **Exchange:** (x86) swaps value between register and memory.
  - **Compare&Swap:** (68000) read value, if value matches register value r1, exchange register r2 and value.

Implementing Locks with Test&Set

- **Test&Set:** reads a value, writes '1' to memory, and returns the old value.

```
class Lock {                    Lock::Acquire() {
  public:                         // if busy do nothing
    void Acquire();               while (test&set(value) == 1);
    void Release();              }
  private:                      Lock::Release() {
    int value;                     value = 0;
}                                }
Lock::Lock {
  value = 0;
}
```

- If lock is free (value = 0), test&set reads 0, sets value to 1, and returns 0. The Lock is now busy: the test in the while fails, and Acquire is complete.
- If lock is busy (value = 1), test&set reads 1, sets value to 1, and returns 1. The while continues to loop until a Release executes.

Busy Waiting

```
Lock::Acquire(){
   //if Busy, do nothing
  while (test&set(value) == 1);
}
```

- What's wrong with the above implementation?
  - What is the CPU doing?
  - What could happen to threads with different priorities?
- How can we get the waiting thread to give up the processor, so the releasing thread can execute?

Locks using Test&Set with minimal busy-waiting

- Can we implement locks with test&set without any busy-waiting or disabling interrupts?
- No, but we can minimize busy-waiting time by atomically checking the lock value and giving up the CPU if the lock is busy

```
class Lock {
  // same declarations as earlier
   private int guard;
}
Lock::Acquire(T:Thread) {
  while (test&set(guard) == 1)  ;
  if (value != FREE) {
     put T on Q;
     T->Sleep() & set guard = 0;
  } else {
     value = BUSY;
     guard = 0;
  } }
```

```
Lock::Release() {
  // busy wait
  while (test&set(guard) == 1) ;
  if Q is not empty {
     take T off Q;
     put T on ready queue;
  } else {
     value = FREE;
  }
  guard = 0;
}
```

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::Semaphore(int val) {
   value = val;
   Q = empty;
}
```

```
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);
} }
```

⇒Signal and Wait of course must be atomic!

Signal and Wait: Example

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