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

What are semaphores?

Implementing locks using test/and busy waiting

More on hardware support for synchronization

Today: Synchronization: Locks and Semaphores

<table>
<thead>
<tr>
<th>Concurrent Programs</th>
<th>Monitors</th>
<th>(Software)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Locks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Send &amp; Receive</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Semaphores</td>
<td></td>
</tr>
</tbody>
</table>
continues to loop until a Release executes.  

- If lock is busy (value = 1), Test/Release reads I, sets value to 0, and returns 1. The while 
  now busy: the test in the while fails, and acquire is complete.
- If lock is free (value = 0), Test/Release reads 0, sets value to 1, and returns 0. The lock is 
  
  ```
  {
  if (value = 0)
  {
  value = 0;
  }
  }
  ```

  lock::lock ()
  
  ```
  {?
  value = 0;
  }
  ```

  lock::Release ()
  
  ```
  {?
  void Release ()
  }
  ```

  with (test/Release (value) == 1) {
  if busy do nothing
  }

  } //

  ```
  lock::Acquire ()
  ```

  class lock {
  
  ```
  {?
  }
  ```

  Implementing locks with Test/Release

  Examples

  - *compare_exchange* (8800) read value, if value matches register value, exchange 

  - *exchange* (x86) swaps value between register and memory

  - *Test/Release* (most architectures) read a value, write I, back to memory

  The multiprocessor must support some type of cache coherence.

  - Invalidate any copies of the value the other processors may have in their cache.

  - On a multiprocessor, the processor issuing the instruction must also be able to 
  invalidate any copies of the value the other processors may have in their cache.

  - Atomic read-modify-write instructions atomically read a value from 
  memory into a register and write a new value.

  Atomic read-modify-write instructions
```java
{ 
  guard = 0;
  value = FREE;
  { 
    while (guard)  
      if (guard == 1)  
        busy.wait();
        // lock::guard();
  } 
}
```

lock value and giving up the CPU if the lock is busy

No, but we can minimize busy-waiting time by atomically checking the

disabling interrupts?

Can we implement locks with test&set without any busy-waiting or

Locks using Test&Set with minimal busy-waiting

```java
{ 
  guard = 0;
  value = BUSY;
  } 
}
```

RELEASEING thread can execute?

How can we get the waiting thread to give up the processor, so the

What could happen to threads with different priorities?

What is the CPU doing?

What's wrong with the above implementation?

```java
{ 
  while (test&set(value) == 1)  
    if (busy)  
      do nothing;
        // lock::acquire();
```
A process can acquire access so long as at least one unit of the resource is available.

- The initial count to which the semaphore is initialized is usually the number of
  unused when multiple units of a resource are available.

- The exclusive access to a resource (only one process is in the critical
  section at a time).

- Counting Semaphore:
  - It is initialized to free (value = 1).
  - Can vary from 0 to 1.
  - Useful when multiple units of a resource are available.

- Binary (or Mutex) Semaphore: (same as a lock)
  - Special atomic instructions.

Synchronization:
- semaphore S
  - A $\rightarrow$ signal() unblocks one process on semaphore S's wait queue.
  - semaphore S
    - If semaphore S is not free, the OS puts the process on the wait queue for
      section (e.g., to buy milk).
    - Each semaphore supports a queue of processes that are waiting to access the critical
      section.

Semaphore: Key Concepts

- S->signal() // that semaphore S is free
- $\rightarrow$block to other processes //

<critical section>
- S->available //
- wait until semaphore S //

Semaphore->Signal() and Semaphore->Wait() and Semaphore->Signal().
Signal and Wait of course must be atomic:

```java
{ 

    if (empty)
        remove p from q;
    if (value == 0)
        Semaphore::semaphore(mutex)
    else
        value = value - 1;
    Semaphore::semaphore(mutex)
    p->block();
    add p to q;
    if (value > 0)
        Semaphore::semaphore(mutex)
    else
        Semaphore::semaphore(mutex)
    value = value - 1;
    Semaphore::semaphore(mutex)
        }
}

```

**Implementing Signal and Wait**

---

**Binary Semaphores: Example**

```
Semaphore::semaphore(mutex)
{ 
    if (buy milk)
        Semaphore::semaphore(mutex)
    if (milk)
        Semaphore::semaphore(mutex)
    lock->acquire();
}

Thread A

Too Much Milk using semaphores:

```
Semaphore::semaphore(mutex)
{ 
    if (buy milk)
        Semaphore::semaphore(mutex)
    if (milk)
        Semaphore::semaphore(mutex)
    lock->acquire();
}

Thread B

Too Much Milk using locks:

```
Semaphore::semaphore(mutex)
{ 
    if (buy milk)
        Semaphore::semaphore(mutex)
    if (milk)
        Semaphore::semaphore(mutex)
    lock->acquire();
}

Thread B

```
Signal and Wait: Example

<table>
<thead>
<tr>
<th>Process State</th>
<th>Empty</th>
<th>Execute</th>
<th>Queue</th>
<th>Value</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Signal and Wait: Example

<table>
<thead>
<tr>
<th>Process State</th>
<th>Empty</th>
<th>Execute</th>
<th>Queue</th>
<th>Value</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Signal and Wait: Example

<table>
<thead>
<tr>
<th>Process State</th>
<th>Empty</th>
<th>Execute</th>
<th>Queue</th>
<th>Value</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multiple Consumers and Producers

Semaphore

Example: You can implement thread.join() for the Unix system call waitpid().

- The initial value of the semaphore is usually 0 in this case.
- When threads must wait for some circumstance.

Scheduling constraints: used to express general scheduling constraints

- Critical section
- Mutual exclusion: used to guard critical sections

Using Semaphores
Semaphores can be used for three purposes:

- To cause one thread to wait for a specific action to be signaled from another thread.
- To control access to a shared pool of resources (using a counting semaphore).
- To ensure mutually exclusive execution of a critical section (as locks do).

Semaphores are a generalization of locks.

Locks can be implemented by disabling interrupts or busy waiting.

Summary

Multiple Consumers and Producers Problem