Mutex: Used for co-ordination of multiple processes.

\[ \text{write}(\cdot) \rightarrow \text{BUFFER} \rightarrow \text{read}(\cdot) \]
\text{block when buffer is full}
\text{block when buffer is empty}

Implementation of mutex using spinlock.
buffer(IN), count, IN, OUT

write(data): while count = N
    wait...
    buffer[IN] = data
    \( [IN] = [IN+1] \mod N \)
    count++

read: while count = 0
    wait
    val = buffer[out]
    count--
    \( [out] = [out+1] \mod N \)

Idea of circular buffer

\[ \text{circular buffer} \quad N=10 \]
Counting Semaphore can be used in above case.

Count initialized to N

Two operations:
- `wait()`: decrements count, can't go below 0
- `signal()`: increments count, can't be greater than N

Semaphore initialized to count=1 is same as mutex:

```plaintext
+-------------------+-------------------+
| Thread 1           | Thread 2           |
| s = semaphore(1);  |                   |
| s.wait();          | s.wait();          |
| s.signal();        |                   |
+-------------------+-------------------+
```

```plaintext
free & = semaphore(N)
write(data) : free.wait()
buffer [IN] = data
[IN] = [IN+1] mod N
count++
used.signal();
```
used = semaphore(0)
read : used.wait()
   val = buffer[OUT]
   count --
   [OUT] = [(OUT+1)%modN
   free.signal()

When buffer is full, wait count = 0
The above pipe reads and writes one
data character at a time.

Now, suppose we want to specify length
'len' of data to be read or write, we can say:
write (data,len) and
val = read (len)

For ex:
thread1: write ("abc") \rightarrow abc, 1234
thread2: write ("1234") \rightarrow 1234 abc
for (i = 1 \ldots \ len)
   write1 (data[i])

With this, we can wrong output as
a b c d e f.

For this purpose, another synchronization
primitive called Monitor can be used here.
Monitor = condition variable = synchronized methods
    in POSIX                in JAVA.
Monitor is a data object, consisting of:
→ fields
→ condition variables
→ methods

Monitors also have the property that only one thread can be executed in the monitor at a time.

Two ways to leave monitor:
1. Leaving method 2 when you wait on condition
   method 2
   \[ \rightarrow \text{wait}(c) \]
   \[ \rightarrow \text{signal}(c) \]

Example: buffer[N]

```c
int count, in, out

condition notempty, notfull

read(len): \text{wait}(notempty)
output: len = min(count, len)
get 'output:len' bytes
\text{signal}(notfull)

write(data, len): \text{wait}(notfull)
\text{towrite} = \min(N-count, len)
put 'towrite' bytes
\text{signal}(notempty)
```

```
If we want to write 10 bytes of data and only 5 are available, this write fails.

We modify write as follows:

```plaintext
write (data, len)
while (N-count < len)
    wait (notfull)
    put "len" bytes
    signal (not empty)
```

Monitors give the ability to write your own rule, which is not possible in semaphore.

Consider:

<table>
<thead>
<tr>
<th>thr1</th>
<th>thr2</th>
<th>thr3</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>read(5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>read(5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>write(10)</td>
<td>10</td>
</tr>
</tbody>
</table>

As we write 10 bytes of data, both thr1 and thr2 reading 5 bytes of data should be awak.

But with read(10) on previous page, this unblocking & returning is not possible.

Also, if only 5 bytes were written, thr1 or thr2, which one should be awake?
This is House semantics w/s Mesa Semantics

Mesa semantics: read(len)

while count < len
    wait (notempty)
    remove 'len' bytes
    broadcast (notfull)

In Mesa Semantics, we've 3 operations:
    wait(), signal(), broadcast()
    notify notifyAll

write(len)

    while n: count < len
        wait (notfull)
        put 'len' bytes
        broadcast (notempty)

Monitors give synchronization mechanism that
is easier to reason about as compared to
semaphores.

Mutex $\rightarrow$ Semaphore $\rightarrow$ Monitor
    ($\leq$Semaphore($i$) $\leq$
    = mutex)

Monitor's at least as powerful as a semaphore.

monitor {
    int count
    cond <
    wait() & while count < 0
    wait(count)
}
```c
signal(c)
{
    count++
    signal(c)
}
```

We can implement semaphores using monitor.

Usage of synchronization primitives.

Two environments to consider:

1) POSIX threads

   `<pthread.h>

   In C, pthread_create(......function, arg)
   pthread_mutex_t(pthread_mutex_init,
                    -lock, -unlock, -destroy)

   `<semaphore.h`

   sem_t  (sem_init, sem_wait, -post, -destroy)
   pthread_cond_t(pthread_cond_init, -destroy, -wait,
                   -signal, -broadcast)

   POSIX threads condition variable are similar to
   monitors, but they can be used together with
   mutexes.

   In POSIX thread, semaphore logic is written as:

   int count
   mutex_t m
   cond_ral c
   [semaphore wait]
   mutex_lock (m)
   while count <= 0
   cond_wait (m, c)
count --
mutex_unlock(m)

[semaphore signal]
mutex_lock(m)
count++
signal(c)
mutex_unlock(m)

2. In Java, the 'synchronized' keyword is used for synchronization. The wait method as wait(this).

Topics covered in this class:
- critical section problem
- spinlocks
- mutexes
- semaphores
- monitors.