Spinlock is a hardware based approach to implement ME.

- Disadvantage: CPU stays in a loop - poor performance, waste of resources.

A solution to this problem is to have the waiting thread to be idle (not looping), and to be woken up when the lock is released.

```
lock spinlock()
if (!locked)
  lock = 1
  release()
  return
else
  add to wait Q
  release
  schedule()
unlock spinlock
if @ != q
  t = dequeue
  append t to ready
else
  lock = 0
  unlock
```
In terms of behavior:
mutex m:
lock()
unlock()

If \( L \) is \# times returned from lock, and \( U \) is \# of calls to unlock, then \( L - U \) is either 0 or 1

**Producers-Consumers Problem (aka bounded buffer)**

Two methods: `put()` and `get()` and want to bound the number of items. We already saw a circular buffer. Another approach is doubly-linked lists

```
head: next, prev
count
```

`put(i)`
```
i -> next = head -> next;
    -> prev = head; h -> next = i;
head = next = i;
count++;
```

`get()` is similar

This data structure is not intrinsically safe, so we would use a mutex on put, locking on start and unlocking at end.

But if the buffer is full/empty we want the threads to wait until the buffer is ready.
\[ x \cdot V = \text{verhogen (increase)} \]

We can do this with a counting semaphore.

Semaphore \( s(N) \) \n\[
\text{p/lock/} \quad \text{wait()} \quad \text{returns from wait}
\]
\[
\text{v/unlock} \quad \text{signal()} \quad s \leftarrow s + 1 \quad \text{calls to signal}
\]
\[
w - s \leq N
\]

Semaphore space(max)

items(0)

put: space.wait

items.signal

get: items.wait

space.signal

A mutex is just a binary semaphore.

**Recursive mutex**

What happens if:

\[
f() \quad \text{lock(m)} \quad \rightarrow \quad g() \quad \text{lock(m)}
\]

\[
\text{unlock()}
\]

**DEADLOCK**
Semaphore extensions

semp. count () → Retrieve # of items
semp. wait (N) → Decrement or increment by N ≠ 1
semp. signal

In this case not all requests are equivalent.

Monitor: A user-defined class, provides synchronization capabilities. Contains variables, conditions and methods.
Conditions are pure synchronization objects.

 fronts
rear
writing threads

fronts
rear
conditions { wait()

methods

1) Mutual exclusion: only 1 thread in the monitor at any given time.

To enter a monitor: call a method or return from unit
To leave a monitor: by returning or waiting
method() → wait()

monitor
wait()
if signal() return

signaling() wakes one of the waiting threads.
monitor $S(N)$:

```java
int count = N
condition C
sem_wait:
    count --
    if count < 0
        wait (c)
sem_signal:
    count ++
    if count < 0
        signal (c)
    count ++
```

In Java, the 'synchronized' keyword, wait, notify and notifyAll (broadcast) are monitor-implemented.

In Posix threads, mutexes, conditions, locking and waiting are available.