Monitors
Condition Variables

Otto J. Anshus
University of {Tromsø, Oslo}

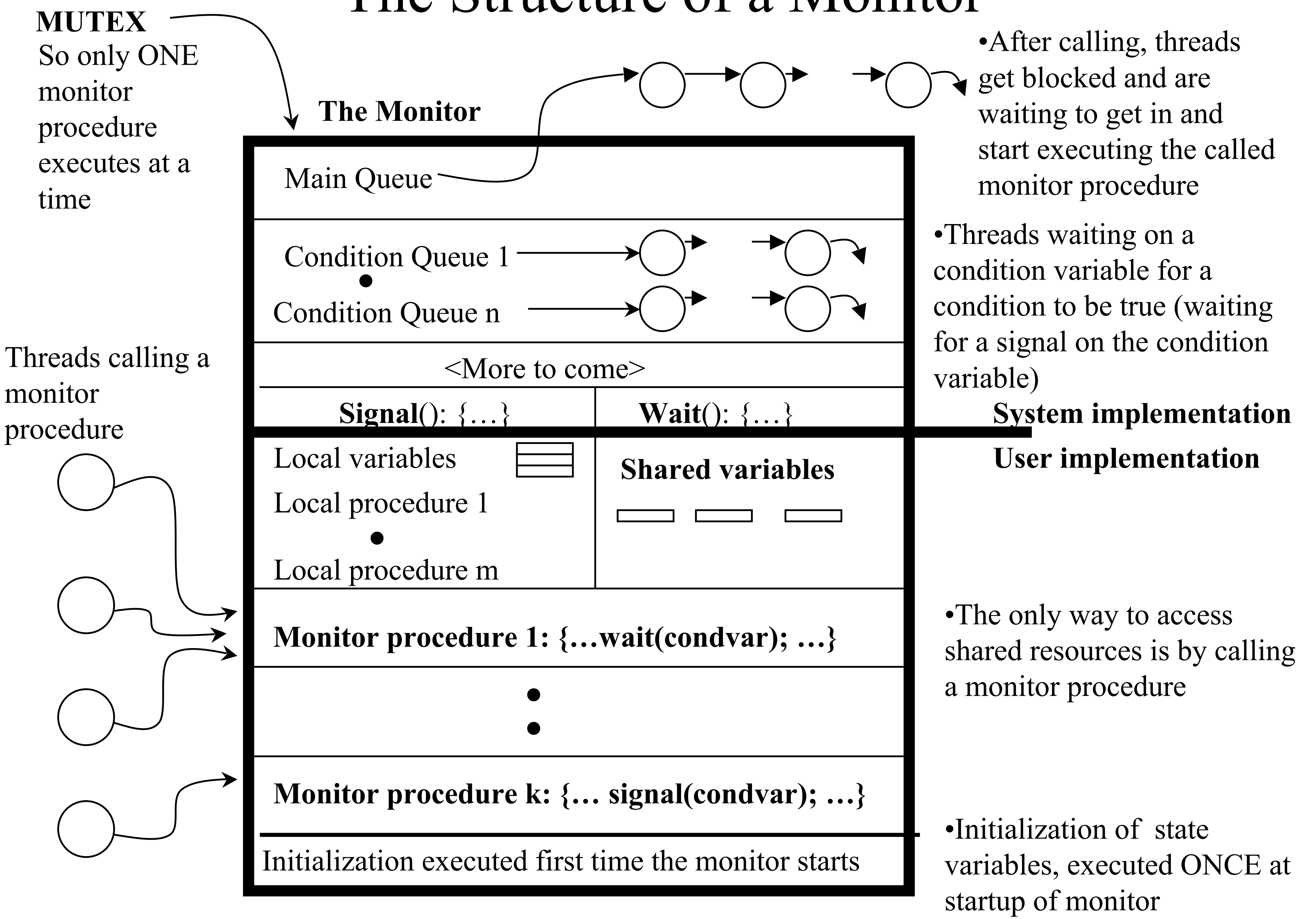
Monitor (Hoare 1974)

- Idea by Brinch-Hansen 1973 in the textbook "Operating System Principles"
    - Structure an OS into a set of modules each implementing a resource scheduler
- Tony Hoare
    - Combine together in each module
        - Mutex
        - Shared data
        - Access methods to shared data
        - Condition synchronization
        - Local code and data

Basic Components

- *Monitor procedures* (called by threads) are meant to be **mutually exclusive**
- Conditon "variables" (declared by user)
- Wait (condition_name) (called by *monitor procedures*)
- Signal (condition_name) (called by *monitor procedures*)

The Structure of a Monitor



Signal and Wait

- Wait (cond)
  - Insert(caller, cond_queue)
  - Block this instance of the monitor procedure
  - open MUTEX by getting next call from Main_Queue
- Signal (cond)
  - Stop monitor procedure calling signal
  - Start first in cond_queue, or just return if empty

Implementation of the Monitor Concept

- As a primitive in a language (Mesa, Java)
- Using semaphores in any language
- As a thread or as a process
  - Need a way to interact with the thread
    - through shared variables to deliver the parameters and name of called monitor procedure
  - Need a way to interact with the process
    - kernel support of shared variables across address spaces
    - using another mechanism like message passing to pass parameters and name of procedure
- At user level,
  - use condition variables (the queues),
  - wait(), signal() implemented by
  - Implementred by
    - the operating system kernel
    - a thread package (Pthreads)

Single Resource Monitor

All threads must follow the pattern:

Reserve;

<use shared resource>

Release;

Observe

- the shared variable
- the naming of the condition variable
- the wait and signal calls
- implements a binary semaphore (s=0,1)

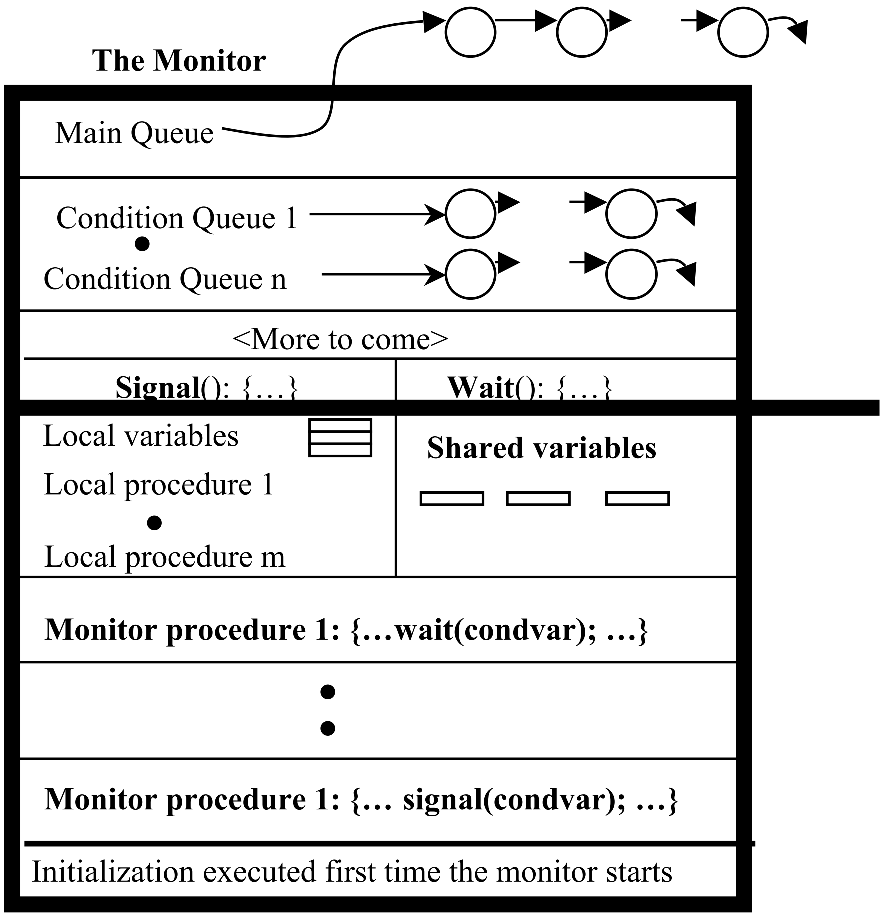
```
/*Local  functions, variables*/
<none needed>
/*Shared variable*/
Boolean busy;
/*Condition variable*/
Condition nonbusy;

Reserve:
{
   if (busy) wait (nonbusy);
   busy:=TRUE;
}

Release:
{
   busy:=FALSE;
   signal (nonbusy);
}

/* Initialization code*/
busy:=FALSE;
nonbusy:=EMPTY;
```

What is a Condition Variable?



- No “value”
- Waiting queue
- Used to represent a condition we need to wait for to be TRUE
- Initial “non-value” is EMPTY :-)

Semaphore vs. Monitor

Semaphore

P(s) means WAIT if s=0
And s--

V(s) means start a waiting thread and REMEMBER that a V call was made: s++

Assume s=0 when V(s) is called: If there is no thread to start this time, the next thread to call P(s) will get through P(s)

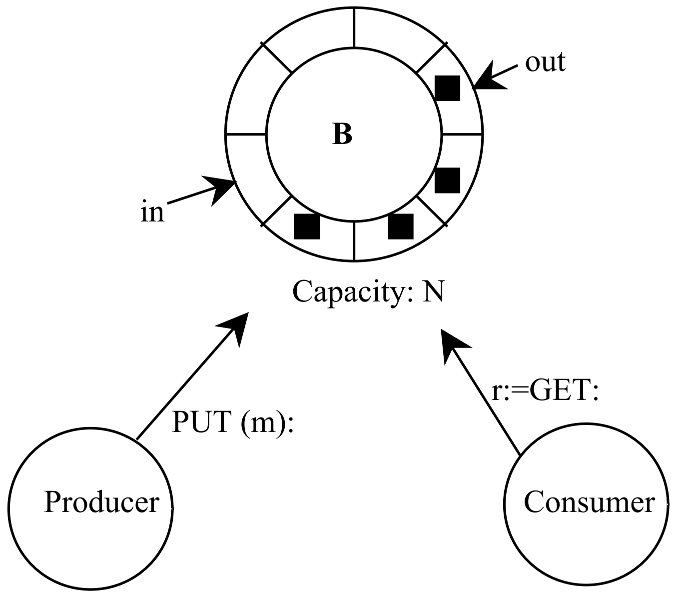
Monitor

Wait(cond) means unconditional WAIT

Signal(cond) means start a waiting thread. But no memory!

Assume that the condition queue is empty when signal() is called. The next thread to call Wait(cond) (by executing a monitor procedure!) will block because the signal() operation did not leave any trace of the fact that it was executed on an empty condition waiting queue.

Bounded Buffer Monitor



Rules for the buffer B:

- No Get when empty
- No Put when full
- B shared, so must have mutex between Put and Get

One condition variable for each condition:

- nonempty
- nonfull
- MUTEX is already provided by the monitor

/*Local functions, variables*/
int in, out;
/*Shared variable*/
int **B(0..n-1), count**;
/*Condition variable*/
Condition **nonfull, nonempty**;

Put (int m):
{ if (count=n) **wait** (nonfull);
B(in):=m;
in:=in+1 MOD n; /* MOD is % */
count++;
signal (nonempty); }

int Get:
{ if (count=0) **wait** (nonempty);
Get:=B(out);
out:=out+1 MOD n;
count--;
signal (nonfull); }

/* Initialization code*/
in:=out:=count:=0;
nonfull, nonempty:=EMPTY;

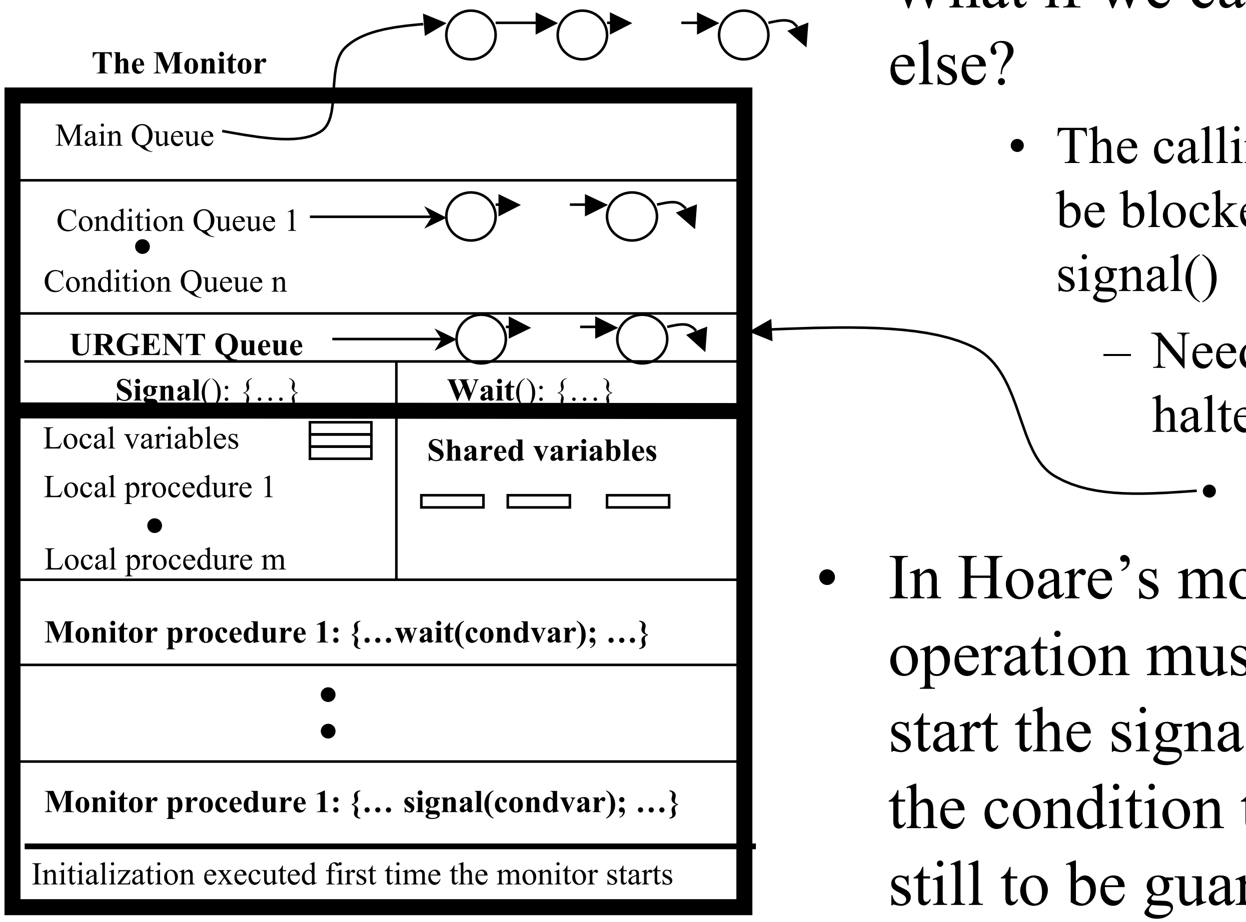
What will happen when a signal() is executed?

- Assume we have threads in Main_Queue and in a condition queue
- Main_Queue has lower “priority” than the signaled condition queue:
    - signal() => Take first from condition queue one and start it from its next instruction after the wait() which blocked it
    - The signaled thread now executes
        - … until a wait(): block it, and take new from Main_Queue
        - … until a signal():
        - … until finished: take new from Main_Queue

Options of the Signaler

- Run the signaled monitor procedure (or thread) *immediately* (must suspend the current one right away) (**Hoare**)
  - If the signaler has other work to do, life gets complex
  - It is difficult to make sure there is nothing more to do because the signal implementation is not aware how it is used (where it is called)
  - It is easy (well, easier) to prove things
- Exit the monitor
  - Just let signal be the last statement before return from a monitor procedure
- Continues its execution
  - Easy to implement
  - But, the condition may not be true when the awaken process actually gets a chance to run

Where to allow a call to signal()?



- Look at the two monitors we have analyzed! Where is the signal() operation?
- What if we called signal somewhere else?
  - The calling function instance must be blocked, awaiting return from signal()
    - Need a queue for the temporary halted thread
      - URGENT QUEUE
- In Hoare's monitors the signal operation must IMMEDIATELY start the signaled thread in order for the condition that it signals about still to be guaranteed true when the thread starts

Mutex between monitor procedures?

- Hoare: Yes
- But not needed if we have no shared variables
  - But **signal** and **wait** must be atomic because they can access the same condition variable
    - So no gain?
      - Finer granularity (is good)
      - Makes life harder (is bad)
- Should be possible to Put and Get at each end of a buffer?
  - Try it

Performance problems of Monitors?

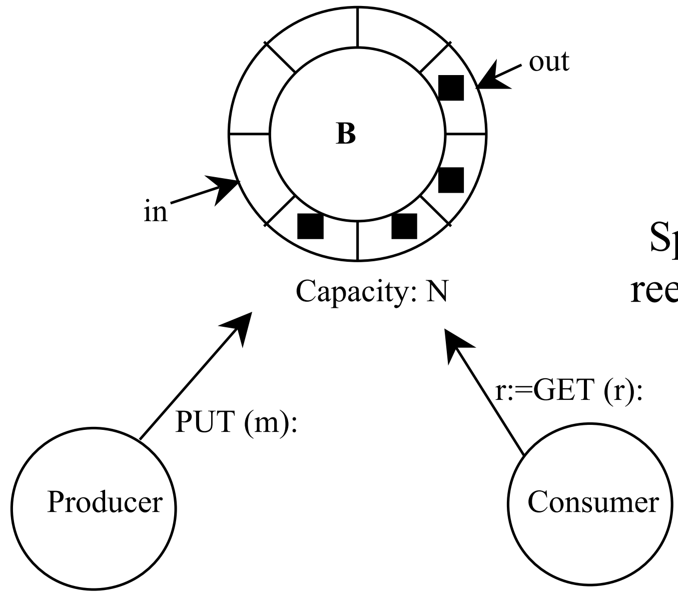
- Getting in through Main_Queue
    - Many can be in Main_Queue and in a condition queue waiting for a thread to execute a monitor procedure calling a signal.
        - Can take a long time before the signaler gets in
    - Need one Wait_Main_Queue and one Signal_Main_Queue?
        - But difficult when all procedures call both wait and signal
- The monitor is a potential bottleneck ("Bottleneck OS"??)
    - Use several to avoid hot spots
- Signal must start the signaled thread immediately, so must switch thread context and save our own
    - Can have nested calls
    - Even worse for process context switches
    - Solution?
        - Avoid starting the signaled thread immediately
        - But then race conditions can happen

Mesa Style "Monitor" (Birrell's Paper)

- Condition variables are associated with a mutex
- Wait(lock, condition)
  - Atomically unlock the mutex and enqueue on the condition variable (block the thread)
  - Re-lock the lock when it is awaken
- Signal(condition)
  - No-op if there is no thread blocked on the condition variable
  - Wake up at **some** convenient time **at least one** (if there are threads blocked)
- Broadcast(condition)
  - Wake up **all** threads waiting on the condition

Is really a NOTIFY or a HINT

Bounded Buffer Mesa Monitors



Rules for the buffer B:

- No Get when empty → nonempty
- No Put when full → nonfull
- B shared, so must have mutex between Put and Get → MUTEX is locked by LOCK and unlocked by Wait

One condition for each condition:

- nonempty
- nonfull
- MUTEX is locked by LOCK and unlocked by Wait

Spins to reevaluate

Wait will UNLOCK

```
/*Local  functions, variables*/
int in, out, count;
/*Shared variable*/
int B(0..n-1);
/* Mutex */
mutex_t bb_mutex;
/*Condition variable*/
Condition nonfull, nonempty;
```

```
Put (int m):
LOCK bb_mutex {
   {  while (count=n) wait (bb_mutex, nonfull);
      B(in):=m;
      in:=in+1 MOD n;
      count++;
      signal (nonempty);   }
}
```

```
int Get:
LOCK bb_mutex {
   {  while (count=0) wait (bb_mutex, nonempty);
      Get:=B(out);
      out:=out+1 MOD n;
      count--;
      signal (nonfull);  }
}
```

```
/* Initialization code*/
in:=out:=count:=0;
nonfull, nonempty:=EMPTY;
```

Instead of LOCK and UNLOCK…

```
static count = 0;
static Cond full, empty;
static Mutex lock;

Enter(Item item) {
  Acquire(lock);
  while (count==N)
    Wait(lock, full);
  insert item into buffer
  count++;
  if (count==1)
    Signal(empty);
  Release(lock);
}

Remove(Item item) {
  Acquire(lock);
  while (!count)
    Wait(lock, empty);
  remove item from buffer
  count--;
  if (count==N-1)
    Signal(full);
  Release(lock);
}
```

Can we replace “while” with “if?”

Think about the performance benefit of this solution

Programming Idiom

- Waiting for a resource

```
Acquire(mutex);
while (no resource)
  wait(mutex, cond);
  use the resource
Release(mutex);
```

- Make resource available

```
Acquire(mutex);
  make resource
Signal(cond);
Release(mutex);
```

Implementing Semaphores with Mesa-Monitors

```
P( s )
{
   Acquire( s.mutex );
   --s.value;
   if (s.value < 0 )
     wait( s.mutex, s.cond );
   Release( s.mutex);
}
```

```
V( s )
{
   Acquire( s.mutex );
   ++s.value;
   if (s.value >= 0 )
     signal( s.cond );
   Release( s.mutex);
}
```

Assume that Signal wakes up exactly one awaiting thread.

Mesa-Style vs. Hoare-Style Monitor

- Mesa-style
  - Signaler keeps lock and CPU
  - Waiter simply put on ready queue, with no special priority
    - Must then spin and reevaluate!
  - No costly context switches immediately
  - No constraints on when the waiting thread/process must run after a "signal"
  - Simple to introduce a broadcast: wake up all
    - Good when one thread frees resources, but does not know which other thread can use them ("who can use j bytes of memory?")
  - Can easily introduce a watch dog timer: if timeout then insert waiter in Ready_Queue and let waiter reevaluate
    - Will guard a little against bugs in other signaling processes/threads causing starvation because of a "lost" signal
- Hoare-style
  - Signaler gives up lock and waiter runs immediately
  - Waiter (now executing) gives lock and CPU back to signaler when it exits critical section or if it waits again

Equivalence

- Semaphores
  - Good for signaling
  - Not good for mutex because it is easy to introduce a bug
- Monitors
  - Good for scheduling and mutex
  - Too (maybe?) costly for simple signaling