Monitors
Condition Variables

Otto J. Anshus

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*** (are **mutually exclusive)**
  - *code written by application programmer*
  - called/executed by threads
    - monitor procedures are implemented by all threads, data variables are shared
  - (called ("...") by processes (without shared address space) is also possible (HOW?))
- ***Condition variable*** on which threads are delayed
  - "declared" by application programmer implementing a monitor's procedures.
    - Appl. programmer sometimes use meaningful name like nonbusy, nonempty, nonfull,.... to describe the condition to wait for
  - the ABSTRACTION "condition variable" is implemented by/in the OS Kernel
  - Just a name. No "value" as such. Behind the scene, inside the OS kernel, there is a wait queue where threads having called wait() are waiting to be resumed by signal()
- ***Primitives*** on condition variables *(implemented by the monitor abstraction)*
  - **Wait** (cond_var_name) (*called inside a monitor procedures)*
    - called when a thread discovers that a condition is such (say, FALSE) that it should wait for the condition to change (say, to TRUE)
    - calling thread will unconditionally be removed as *current* and from R_Q, and inserted into the waiting queue associated with the condition variable
      - then the OS kernel scheduler must select another process from the R_Q to become the new *current*
  - **Signal** (cond_var_name) (*called inside a monitor procedures)*
    - resume (wakeup) a blocked thread (*immediately* for Hoare Monitors, *eventually* for Mesa Monitors)
    - if no threads in wait queue, signal() has *no effect* (NB: no memory of the number of signals as we had with semaphores)

How a Monitor Can Look As Seen By UL Code

To use a monitor all threads better respect this pattern:

```
...anycode...
P(mname);
  ...moncode...
V(mname)
...anycode...
```

mname is the name of a mutex

WHY do we need it?

moncode is the "monitor procedure", typically **sys**calling wait() to delay itself:

EXAMPLE: **if busy {wait(nonbusy)}**

or **sys**calling signal() to resume another thread (which called wait() at an earlier time:

EXAMPLE: **signal(nonbusy)**

How a Monitor Can Look As Seen By UL Code

To use a monitor all threads better respect this pattern:

```
...anycode...
{ P(mname);
    ...moncode...
  V(mname)
...anycode...
```

mname is the name of a mutex

WHY do we need it?

moncode is the “monitor procedure”, typically **sys**calling wait() to delay itself:

EXAMPLE: **if busy {wait(nonbusy)}**

or **sys**calling signal() to resume another thread (which called wait() at an earlier time:

EXAMPLE: **signal(nonbusy)**

Got You: You block inside a mutex - this will probably result in a deadlock

How a Monitor Can Look As Seen By UL Code

To use a monitor all threads better respect this pattern:

```
...anycode...
P(mname);
  ...moncode...
V(mname)
...anycode...
```

mname is the name of a mutex

WHY do we need it?

moncode is the "monitor procedure", typically **sys**calling wait() to delay itself:

EXAMPLE: **if busy {wait(nonbusy)}**

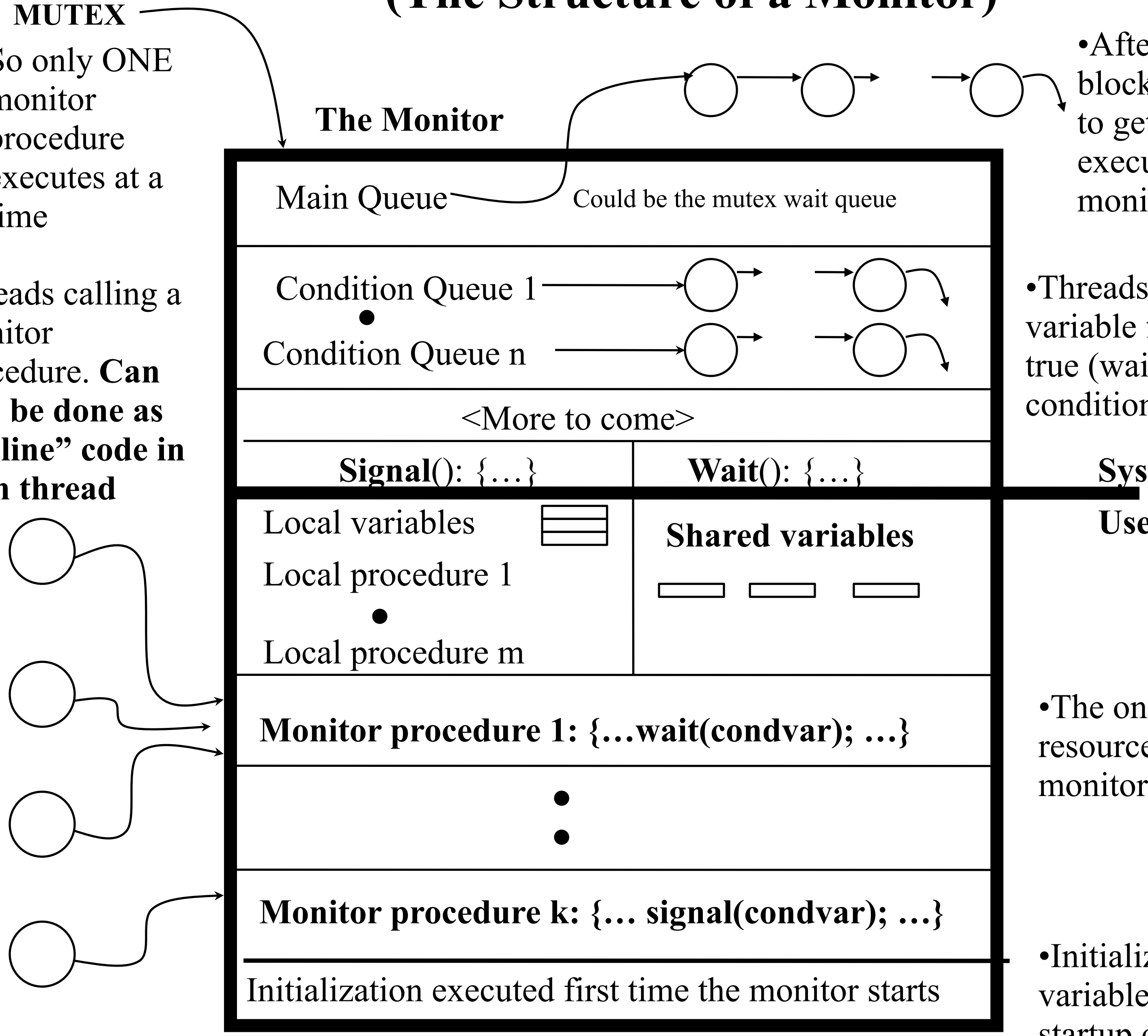
or **sys**calling signal() to resume another thread (which called wait() at an earlier time:

EXAMPLE: **signal(nonbusy)**

Got You: You block inside a mutex - this will probably result in a deadlock

Not so fast: The implementation of wait() inside the Kernel will open up the mutex.

One way of remembering what the monitor abstraction is
(The Structure of a Monitor)



MUTEX

So only ONE monitor procedure executes at a time

The Monitor

Main Queue

Could be the mutex wait queue

•After calling, threads get blocked and are waiting to get in and start executing the called monitor procedure

Condition Queue 1

Condition Queue n

•Threads waiting on a condition variable for a condition to be true (waiting for a signal on the condition variable)

<More to come>

Signal(): {…}

Wait(): {…}

System implementation

User implementation

Threads calling a monitor procedure. **Can also be done as "in-line" code in each thread**

Local variables

Local procedure 1

Local procedure m

Shared variables

Monitor procedure 1: {…wait(condvar); …}

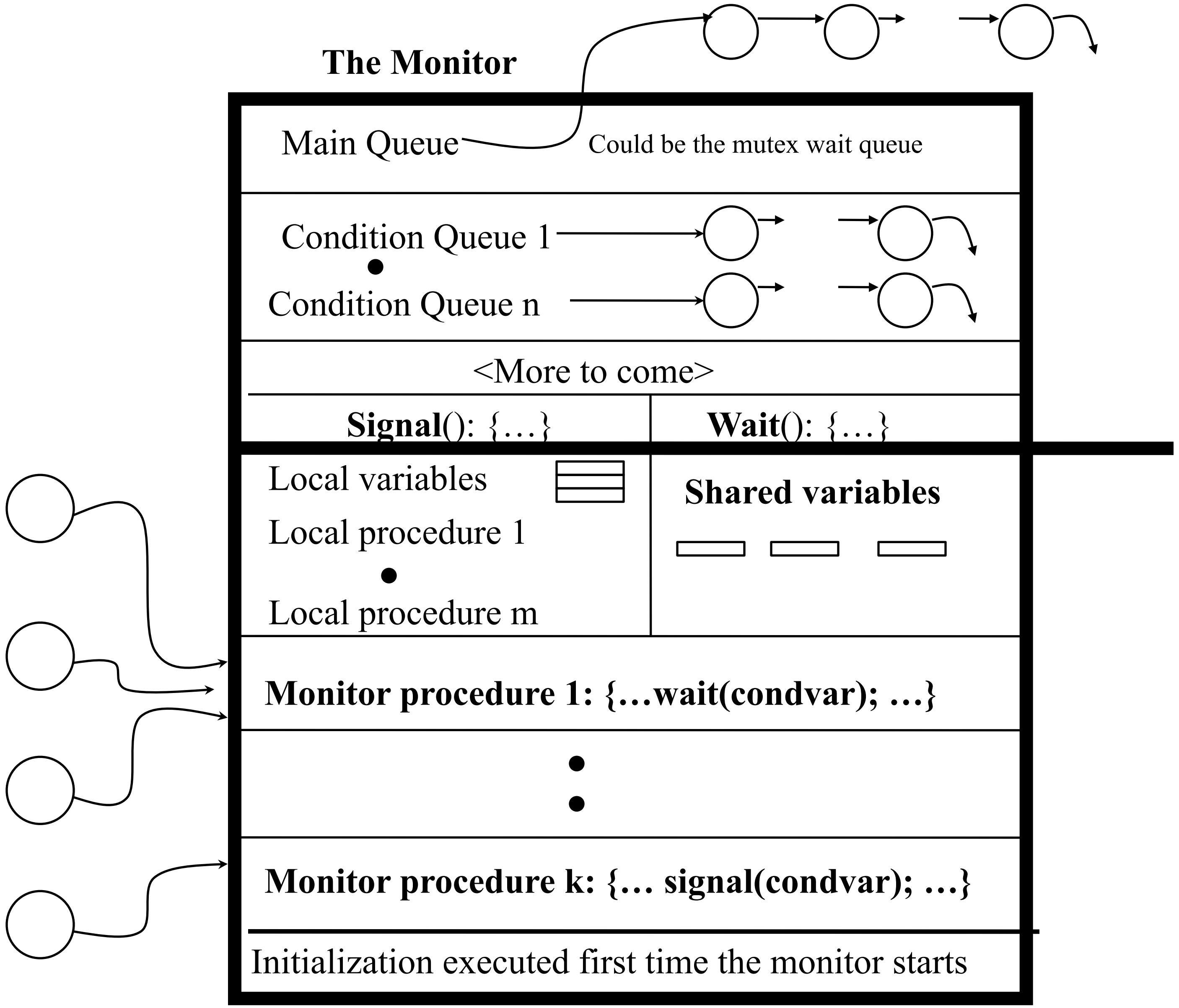
•The only way to access shared resources is by calling a monitor procedure

Monitor procedure k: {… signal(condvar); …}

Initialization executed first time the monitor starts

•Initialization of state variables, executed ONCE at startup of monitor

The Monitor



Main Queue

Could be the mutex wait queue

Condition Queue 1

●

Condition Queue n

<More to come>

Signal(): {…}

Wait(): {…}

Local variables

Local procedure 1

●

Local procedure m

Shared variables

Monitor procedure 1: {…wait(condvar); …}

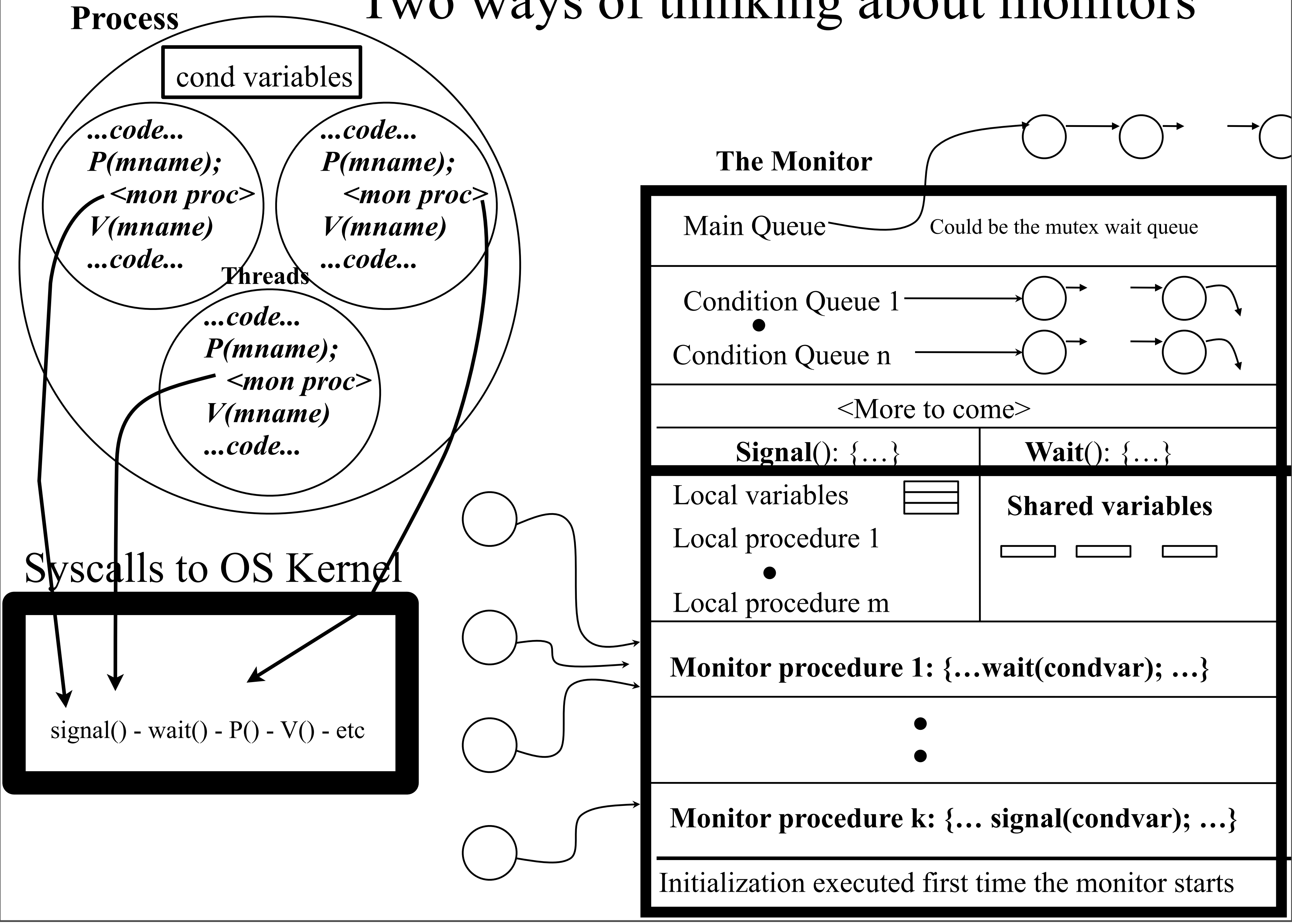
●

●

Monitor procedure k: {… signal(condvar); …}

Initialization executed first time the monitor starts

Two ways of thinking about monitors



Process

cond variables

...code...
P(mname);
<mon proc>
V(mname)
...code...

Threads

Syscalls to OS Kernel

signal() - wait() - P() - V() - etc

The Monitor

Main Queue

Could be the mutex wait queue

Condition Queue 1

●

Condition Queue n

<More to come>

Signal(): {…}

Wait(): {…}

Local variables

Local procedure 1

●

Local procedure m

Shared variables

Monitor procedure 1: {…wait(condvar); …}

●

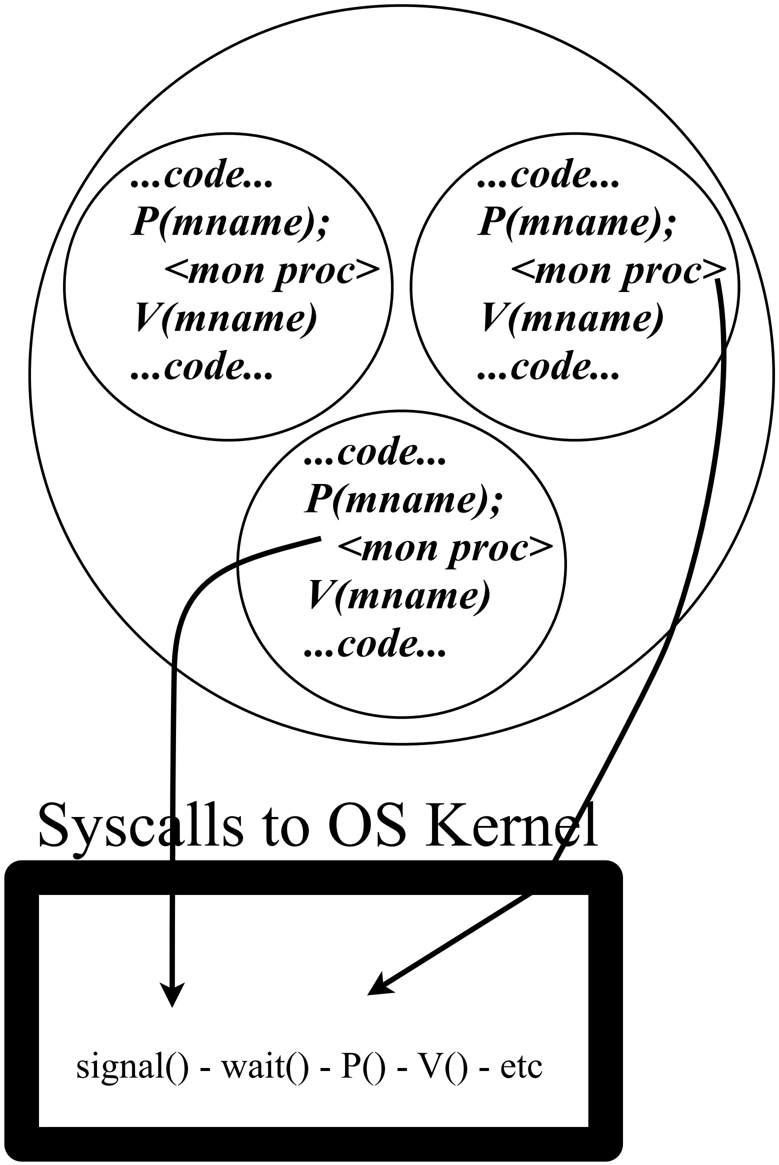
●

Monitor procedure k: {… signal(condvar); …}

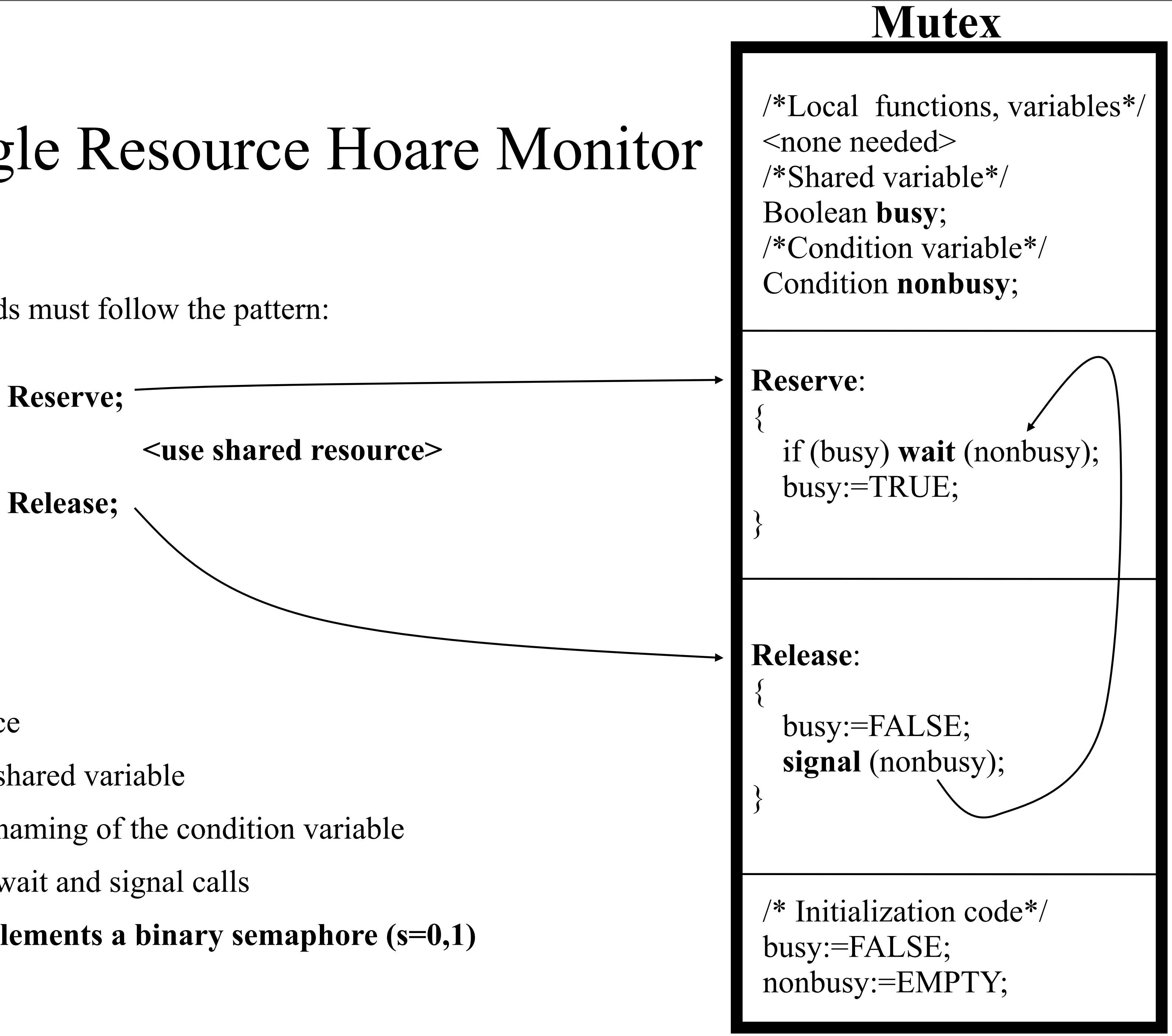
Initialization executed first time the monitor starts

Approaches to Implementing the Monitor Abstraction

- As a primitive in a language (Mesa, Java)
- By 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
- **What we will do**
  - **User** Level code
    - **mutex by P-V**
    - Use wait() and signal() and condition variables
  - **Kernel**
    - **condition variables (the queues)**
    - **wait(), signal()**



Single Resource Hoare Monitor



All threads must follow the pattern:

Reserve;

<use shared resource>

Release;

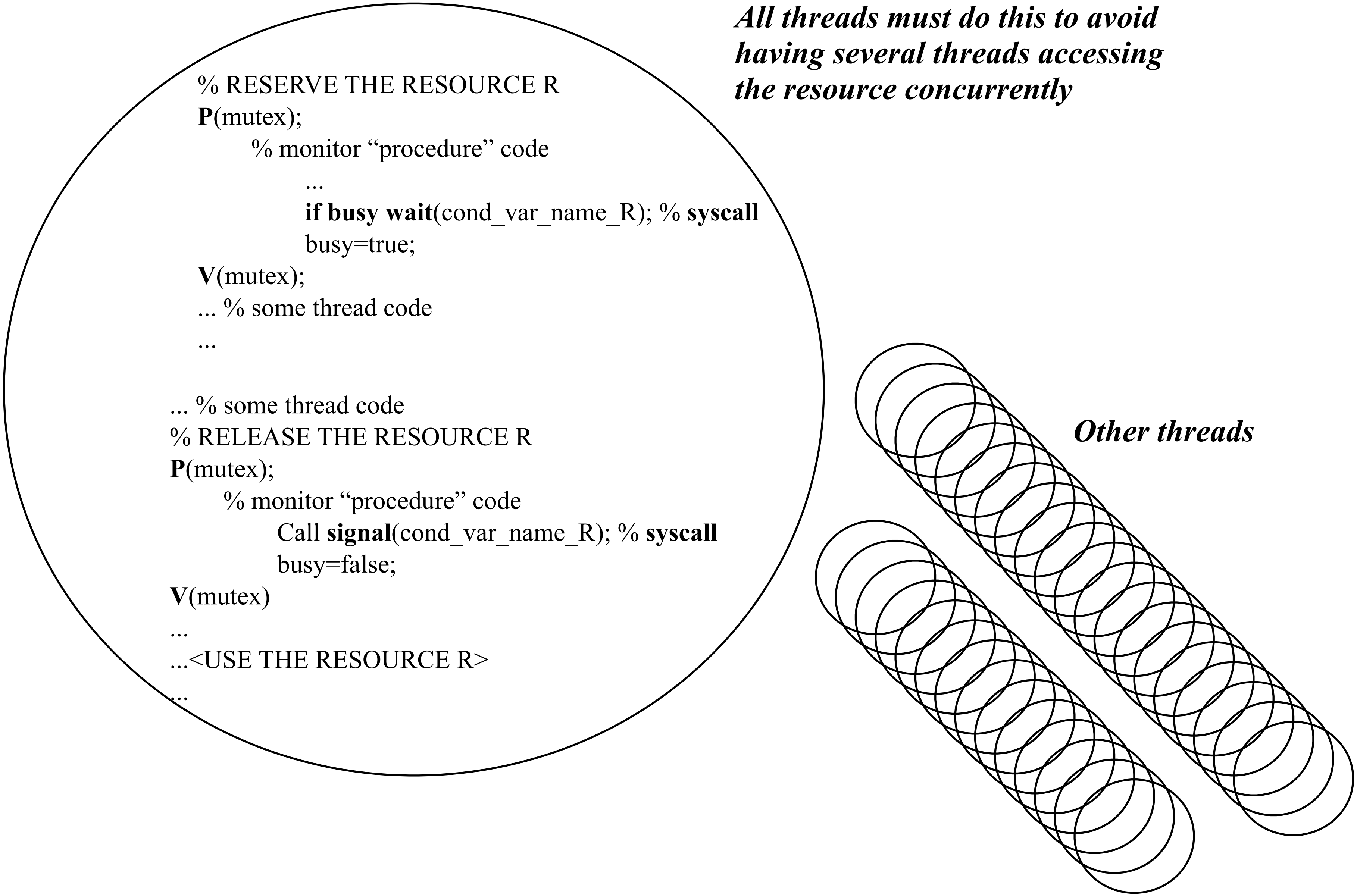
Mutex

```
/*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;
```

Notice

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

Single Resource Monitor

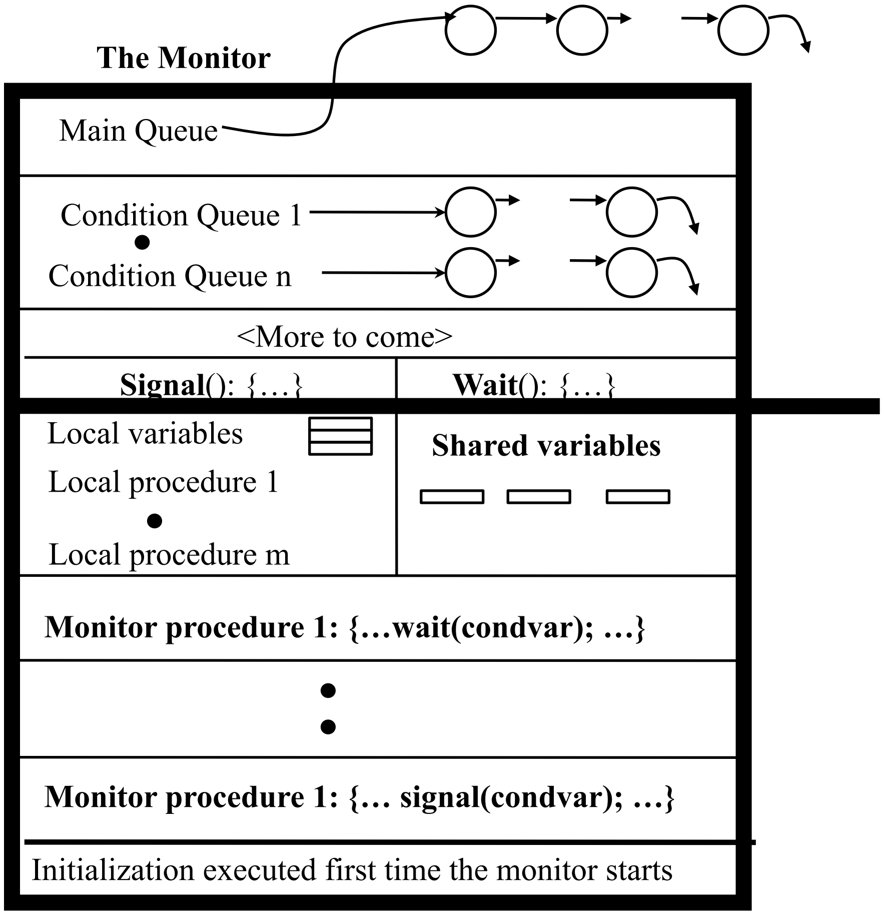


All threads must do this to avoid having several threads accessing the resource concurrently

```
% RESERVE THE RESOURCE R
P(mutex);
        % monitor “procedure” code
                ...
                if busy wait(cond_var_name_R); % syscall
                busy=true;
V(mutex);
... % some thread code
...

... % some thread code
% RELEASE THE RESOURCE R
P(mutex);
        % monitor “procedure” code
                Call signal(cond_var_name_R); % syscall
                busy=false;
V(mutex)
...
...<USE THE RESOURCE R>
...
```

Other threads

What is a Condition Variable?



The Monitor

Main Queue

Condition Queue 1

•

Condition Queue n

<More to come>

Signal(): {…} | **Wait**(): {…}

Local variables

Local procedure 1

•

Local procedure m

Shared variables

Monitor procedure 1: {…wait(condvar); …}

•

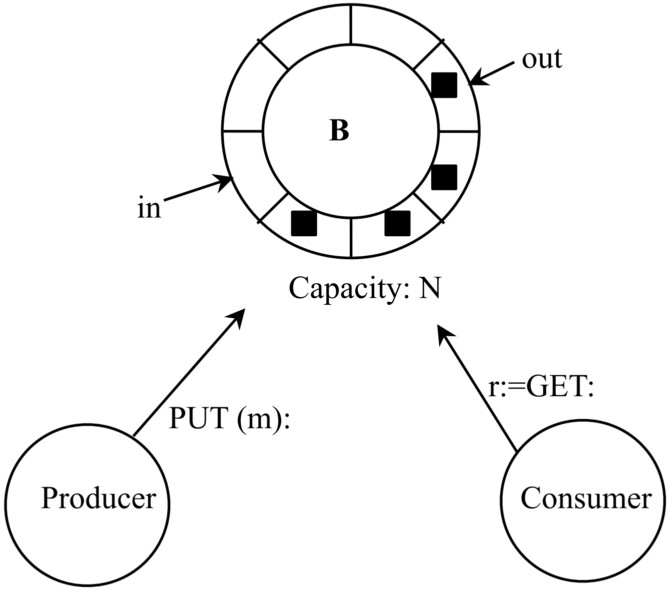
•

Monitor procedure 1: {… signal(condvar); …}

Initialization executed first time the monitor starts

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

Bounded Buffer Monitor

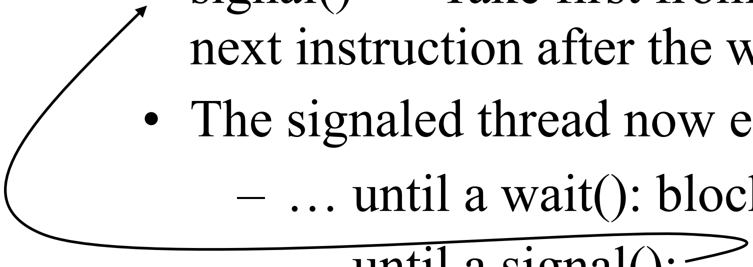


Rules for the buffer B:	**One condition variable for each condition:**
•No Get when empty	•nonempty
•No Put when full	•nonfull
•B shared, so must have mutex between Put and Get	•MUTEX is already provided by the monitor

```
/*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?

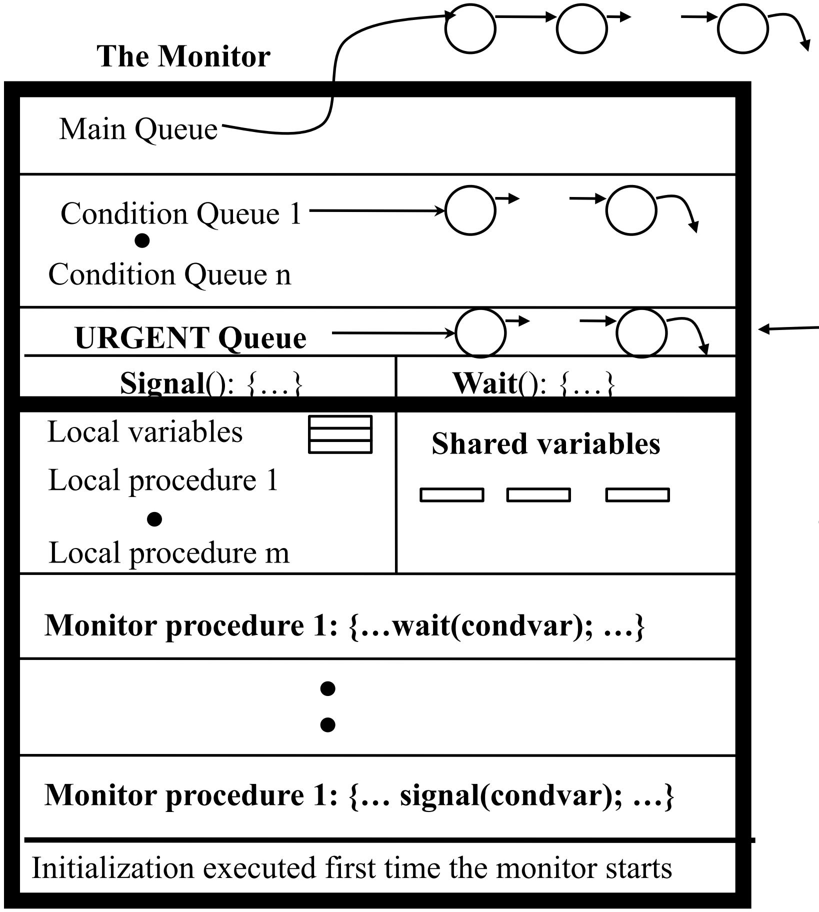
- Assume we have threads in Main_Queue and in a condition queue
- Main_Queue has lower “priority” than the signaled condition queue:



    - signal() => Take first from condition queue 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

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



The Monitor

Main Queue

Condition Queue 1

Condition Queue n

URGENT Queue

Signal(): {…}

Wait(): {…}

Local variables

Local procedure 1

Local procedure m

Shared variables

Monitor procedure 1: {…wait(condvar); …}

Monitor procedure 1: {… signal(condvar); …}

Initialization executed first time the monitor starts

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 complicated
  - 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 easier to prove things
- Exit the monitor (**Hansen)**
  - Just let signal be the last statement before return from a monitor procedure
- Just continue to execute the caller of signal() (**Mesa)**
  - Easy to implement
  - But, the condition may not be true when the awaken process actually gets a chance to run
    - Consequently the monitor procedures must be rewritten just a little bit

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
    - Takes time and results in increased latency (and we don't want a SLOW synchronization mechanism :))
        - Made even worse since we can have nested calls
    - Even worse for process context switches
  - Solution?
    - **Brilliant idea: Avoid starting the signaled thread immediately**
        - But then race conditions can happen so must be careful and think here...

Mesa Style “Monitor” (Birrell’s Paper)

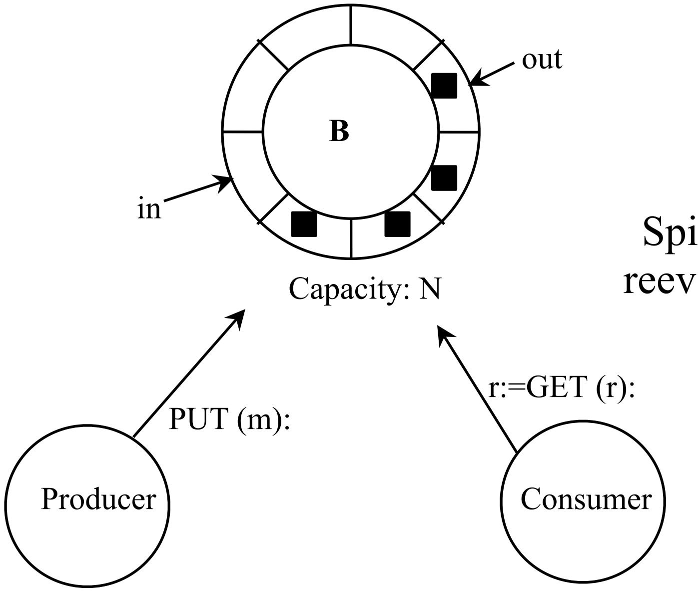
- Condition variables are always associated with a mutex
- Wait(**mutex**, condition)
  - Atomically unlock the mutex and enqueue on the condition variable (block the thread)
  - Re-lock the lock when it is awaken
- **Signal**(condition)

Is really a NOTIFY or a HINT

  - 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)
    - **Simple to do**: Just insert the threads into the Ready_Queue
- **Broadcast**(condition)
  - Wake up **all** threads waiting on the condition
    - ALL gets to reevaluate condition resulting in the wait() call they did some time ago
  - **Simple to do**: insert them all into the Ready_Queue

In this course we will implement the MESA style monitor concept in the OS Kernel

Bounded Buffer Mesa Monitors



```
  /*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;
```

Wait will UNLOCK

Spins to reevaluate

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 for each condition:

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

Mesa-Style vs. Hoare-Style Monitor

- Mesa-style
  - Signaler keeps lock and CPU
  - The awakened thread is simply inserted into the 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

Programming Style w/Mesa Monitors

- 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.

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.

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