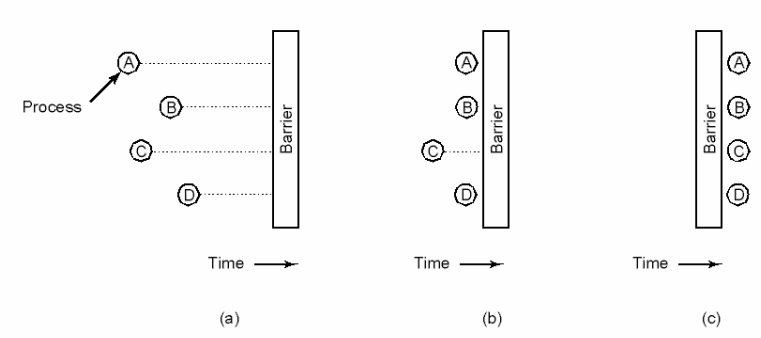
Monitors, …

Pål Halvorsen

(including slides from *Otto J. Anshus*, University of Tromsø,
and *Kai Li*, Princeton University)

Barriers



- Use of a barrier
  - useful for phase-based, cooperative computing

a. processes approaching a barrier
b. all processes but one blocked at barrier
c. last process arrives, all are let through

IPC is easy using semaphores!!??

- The producer-consumer problem using semaphores:

NB! P = DOWN and V = UP

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE is the constant 1 */
        item = produce_item( );                 /* generate something to put in buffer */
        down(&empty);                           /* decrement empty count */
        down(&mutex);                           /* enter critical region */
        insert_item(item);                      /* put new item in buffer */
        up(&mutex);                             /* leave critical region */
        up(&full);                              /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                              /* infinite loop */
        down(&full);                            /* decrement full count */
        down(&mutex);                           /* enter critical region */
        item = remove_item( );                  /* take item from buffer */
        up(&mutex);                             /* leave critical region */
        up(&empty);                             /* increment count of empty slots */
        consume_item(item);                     /* do something with the item */
    }
}
```

Can we switch down order?

If full buffer, producer blocks holding the mutex

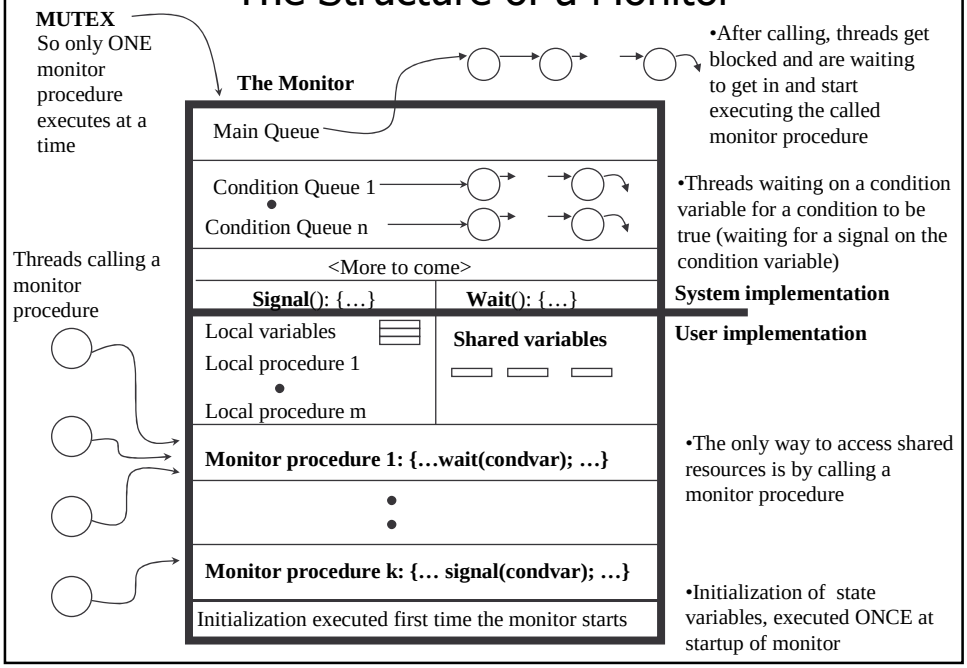
If empty buffer, consumer blocks holding the mutex

Can we switch `full` and `empty` operations? → NO, only `down` is blocking, and `full`/`empty` may reach values above N

Monitors (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
- Combine together in each module
  - Mutex
  - Shared data
  - Access methods to shared data
  - Condition synchronization
  - Local code and data
- Processes can call monitor functions, but not access internal data directly – only through functions
- Only one process can be active in a monitor at a time

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

Single Resource Monitor

All threads must follow the pattern:

Acquire;

<use shared resource>

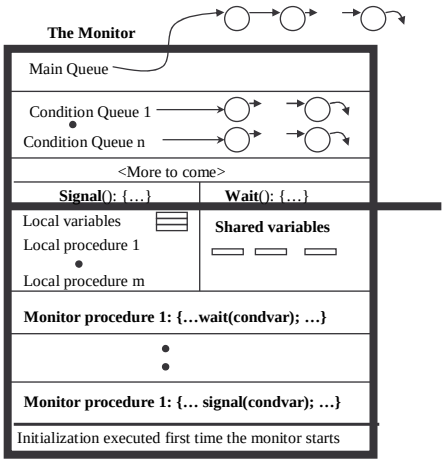
Release;

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

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

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

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

Observe

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

What is a Condition Variable?

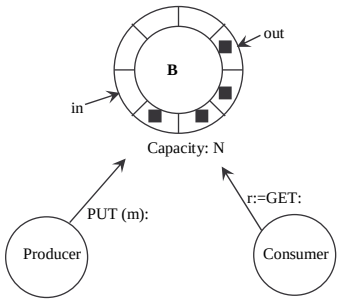


- No "value", no counting
- Used to represent a condition we need to wait for to be TRUE
- Unused signals are lost
- Waiting queue
- Initial "non-value" is EMPTY :-)

Semaphore vs. Monitor

Semaphore	Monitor
P(s) (or **down**) means WAIT if s=0 And s--	**Wait**(cond) means unconditional WAIT
V(s) (or **up**) means start a waiting thread and *REMEMBER* that a V call was made: s++	**Signal**(cond) means start a waiting thread. But *no memory*!
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)	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

Do we need these IFs?

Waiting producers?

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

Put (int m):          Buffer has N slots - full
{  if  (count = n) wait (nonfull);
   B(in):=m;
   in:=in+1 % n;
   count++;           Waiting consumers?
   if (count = 1) signal (nonempty);  }

int Get:              Nothing in buffer
{  if (count = 0) wait (nonempty);
   Get:=B(out);
   out:=out+1 % n;
   count--;
   if (count = N-1) signal (nonfull);  }

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

Readers and Writers Monitor

```
monitor ReaderWriter
        condition ok2read, OK2write
        int readercount
        bool busy

        procedure startwrite {
                if (readercount != 0 OR busy) OK2write.wait;
                busy := TRUE; }

        procedure endwrite {
                busy := FALSE;
                if (OK2read.queue ) OK2read.signal;
                else OK2write.signal; }

        procedure startread {
                if (busy OR OK2write.queue) OK2read.wait;
                readercount ++;
                OK2read.signal; }

        procedure endread {
                readercount --;
                if (readercount == 0) OK2write.signal; }


        readercount := 0;
        busy := FALSE;
        OK2write := OK2read := NONE;
end monitor
```

Rules (strong):
- Many readers, no writers
- One writer, no others
- No writer starvation
- No reader starvation

Prevents writer starvation

Prevents writer starvation

Where is critical region "executed"?

Busy indicates a writer is active

Dining-Philosopher Monitor

```
monitor DP
    condition self[5];
    enum {thinking, hungry, eating} state[5];

    procedure test(int i) {
        if ( state((i+1) % 5) != eating && state((i-1) % 5) != eating &&  state(i) = hungry ) {
            state[i] = eating;
            self[i].signal; }
    }

    procedure pickup(int i) {
        state[i] = hungry;
        test(i);
        if ( state[i] != eating ) self[i].wait;
    }

    procedure putdown(int i) {
        state[i] = thinking;
        test((i+1) % 5);
        test((i-1) % 5);
    }

    for (i=0;i<5;i++) state[i] = thinking;
end monitor
```

If hungry and neighbors not eating

See if neighbors are hungry and can start eating

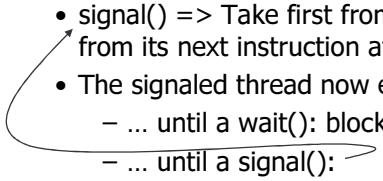
```
philosopher(i)
    while (1) {
        think();
        dp.pickup(i)
        ...eat...
        dp.putdown(i)
    }
}
```

Everyone are thinking at first

What about starving to death?

What will happen when a signal() is executed?

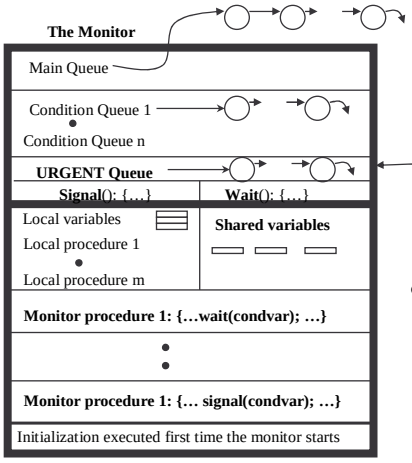
- Assume we have threads in Main_Queue and in a condition queue
- Main_Queue has lower "priority" than the signaled condition queue:
  - signal() => Take first from condition queue one and start it from its next instruction after the wait() which blocked it
  - The signaled thread now executes
    -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

- Relinquishes control to the woken process (Hoare)
  - Complex if the signaler has other work to to
  - To make sure there is no work to do is difficult because the signal implementation is not aware how it is used
  - It is easy to prove things
- Continues its execution (Mesa)
  - Easy to implement
  - But, the condition may not be true when the woken 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

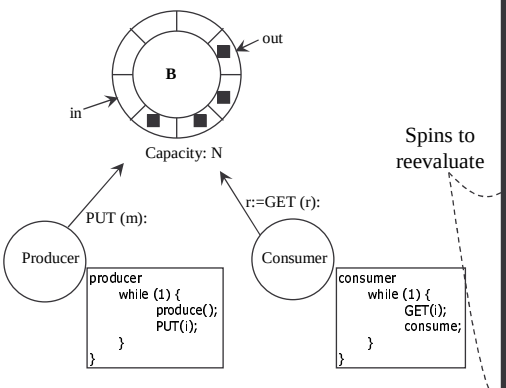
Performance problems of Monitors?

- 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
- 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)

- Mesa monitor is similar to condition variable + mutex
- Acquire and Release (lock and unlock)
- 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)
  - Noop 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) – signal to all
  - Wake up **all** threads waiting on the condition

Bounded Buffer Mesa Monitors



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

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

Wait will UNLOCK

```
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;
```

Implement 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
  - Used in many operating systems
  - Signaler keeps lock and CPU
  - Waiter simply put on ready queue, with no special priority
    - Must then spin and reevaluate! (replace IF with WHILE)
  - 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
  - Described in most textbooks
  - Signaler gives up lock and waiter runs immediately (more context switches, but guaranteed condition)
  - 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
  - Can do everything, but easy to introduce a bug
  - Does not loose "counts" - memory
- Monitors
  - Good for scheduling and mutex
  - Too costly for a simple signaling
  - No memory – lost signals

Why manage critical sections?

- Often, in common case, everything goes just fine...
- But, what if it doesn't....
  - Mars Pathfinder:
    priority inversion – introduced priority inheritance when holding a mutex
  - INSTANCE-I:
    lost wake-up call – introduced time-out
  - ...
  - I guess you'll find out the hard way yourself!!!! (P3)

Summary

- Parallel programming easier with monitors than semaphores (mutual exclusion is automatic)
- Most programming languages does not have Hoare monitors (or semaphores)
- Mesa-style monitors are often used in OS'es
- Something else in needed in a distributed environment without shared memory (later…)
- SMP and monitors might be inefficient (blocks much)
- Managing critical sections is important!!!

Some further reading…

- C.A.R Hoare: "Monitors: An Operating System Structuring Concept", Communications of the ACM, Vol. 17, No. 10. October 1974, pp. 549-557
- A.D. Birrell.: "An Introduction to Programming with Threads", Digital Equipment Corp. (DEC), 1989