Instructor Notes

C.A.R. Hoare
Monitors: An Operating System Structuring Concept
Communications of the ACM 17, 10, October 1974, pp. 549-557

Butler W. Lampson, David D. Redell
Experiences with Processes and Monitors in Mesa
Communications of the ACM, 23 2, February 1980, pp. 105-117

Neat contrast

1. What are the advantages of using monitors and condition variables instead of semaphores?

+ Organize shared data + code + synchronization in structured fashion (unifies it)

+ Separates mutual exclusion + scheduling

+ Fewer errors - automatically release locks

+ Leads to more formalism - specify invariants (when enter monitor & after wait)

2. What is the idea of a monitor? What is contained in a monitor (assume Mesa)? What is the idea of a condition variable? What is the key difference between a semaphore and a condition variable?

- Only 1 process @ a time executing in monitor
  - automatically acquire lock on entry
  - " " release lock on exit

- Mesa: 3 types of procedures
  1) entry (acquires lock)
  2) internal (private) - already has lock
  3) external (non-monitor) - doesn't need lock but logically related

- CV: delay execution until some resource is available

- Sem? No state in CV!

→ Need other variables to record state

cv. wait - stuck until someone calls cv. signal afterwards

3. Why is it not a good idea to implement mutual exclusion by using a non-preemptive scheduler? (add yields explicitly)

- doesn't work on multi-processors
- need to be able time-critical events such as I/O
- modularity in critical sections is poor
  - does called procedure yield processor?
- What do you do on a page fault?

⟹ Want explicit locks for mutual exclusion

4. Using the ~~definitions~~ semantics proposed by Hoare, how would you implement synchronization for a bounded buffer?

```
monitor {
    int count = 0;
    cv nonfull, nonempty;
    produce () {
        if (count == N) {
            nonfull.wait ();
        }
        fill buffer ();
        count++;
        nonempty. signal ();
    }
    consume() {
        if (count == 0) {
            nonempty. wait();
        }
        use buffer ();
        count--;
        non full. signal();
    }
```

Ex:

C1 : waits on nonempty (releases monitor lock)

C2 : also waits...

P1 : produces – signals nonempty

C1 (or C2) wakes and is guaranteed to be next process to run in monitor

5. What does Hoare assume will occur when a process calls signal on a condition variable? What are the advantages of this assumption? What are the disadvantages?

Signal: stop running process, release monitor lock, ensure waiting process runs next (grabbing monitor lock)

\+ Helps with reasoning
· invariant set by signalling process is guaranteed to hold

\- Extra context switch

\- Imposes rules on process scheduler

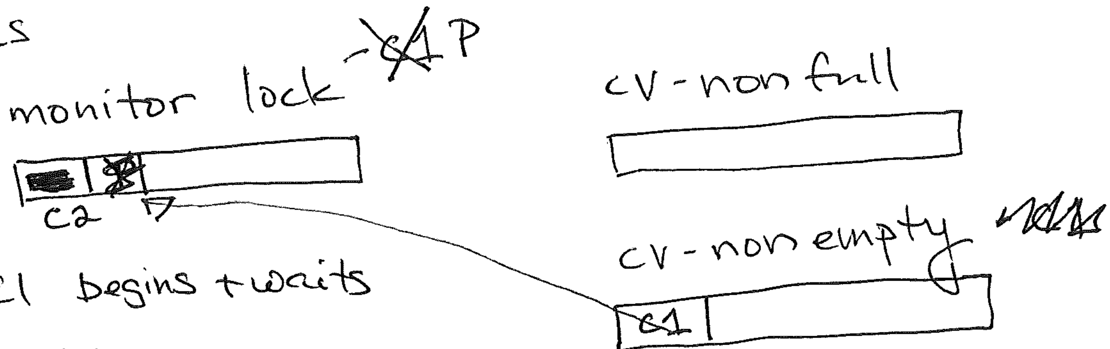
\- Doesn't mesh well with implementation and hand-off of monitor lock (separate monitors from process scheduler)

6. When implementing monitors and condition variables in Mesa, what important change was made from Hoare's assumptions? Why is this a more natural implementation? How do applications need to change in Mesa? What other changes can be made in the implementation and in applications because of this change?

Mesa Change: Wake 1 waiting process from wait, but not guaranteed to run next or grab monitor lock next

Implementation:

queues



Eg: C1 begins + waits

P fills

- C2 arrives, waits on monitor
- P signals → Remove C1 from cv queue + add to monitor q
- C2 might get monitor lock next
  - it will see count==1 & use buffer
- When C1 aquires monitor, invariant does not hold - incorrectly "uses" empty buffer

App changes: Recheck condition

→ Change "if" to while loop

- Once know apps are rechecking, signal can be a <u>hint</u> (okay to be sloppy)

- Change all ~~not~~ signals (notifies) to broadcasts (correct - help find bugs)

- Can be easier - make larger covering condition when don't know exactly which process should wake, wake all!

7. How could you implement the synchronization for a memory allocator in Mesa?

```
allocate (size) {
    while (availmem < size) {
        cv.wait();
    }
    getmem();
    availmem -= size;
}

free (ptr, size) {
    putmem();
    availmem += size;
    cv.notify();
}
```

What could go wrong? process 2 frees ~~8~~ B bytes

- 1 process wakes, waiting for large amount of mem - goes back to waiting
- meanwhile, process B needed that exact amount of memory - never woken!

Fix?

Change to bcast (notifyAll)

8. What should happen when monitor M calls monitor N which performs a wait? When must invariants be established?

```
M() {
    :
    N() {
        :
        wait()
        :
    }
}
```

- What lock should be released? Just N!

  M doesn't know that N called wait so doesn't know it could lose its monitor lock (would have to restore invariants before all unknown calls)

→ Problem: deadlocks

9. What are other differences between Hoare and Mesa interfaces?

- priority to wait()
- can look @ # waiters on cv.queue

10. How can reader/writer locks be implemented with Hoare semantics? What rules are enforced in this example? How do multiple readers start? How would this example be changed in Mesa?

readers — multiple okay

writer — 1 only

- what should happen when writer waiting?
- who should get to enter next?

```
readercount=0;
busy=false; (writing)

startread() {      ① if
    ~~while~~ ② while
    if (busy || OKtowrite.queue) {      <- don't starve writers
        OKtoread.wait();
    }
    readercount++;
    OKtoread.signal();      <- used for multiple readers to get going
}
endread() {
    readercount--;
    if (!readercount) OKtowrite.signal();      <- let writer go if we are last reader
}
startwrite() {
    while <- if (readercount || busy) {
        OKtowrite.wait();
    }
    busy=true;
}
endwrite() {
    busy=false;
    if (OKtoread.queue) {      <- policy: let readers go next
        OKtoread.signal();      <- change to bcast
    } else    OKtowrite.signal();
```

Proposed Change

??

```
if (OKtowrite.queue) {
    OKtoread.wait();
}
while (busy) {
    OKtoread.wait();
}
readercount++;
```

11. Why is it useful to have monitored objects in addition to monitor modules?

-Multiple monitor locks for better concurrency

12. Why is a "naked notify" needed in Mesa?

· Device needs to signal waiting process

· Can't grab monitor lock

device

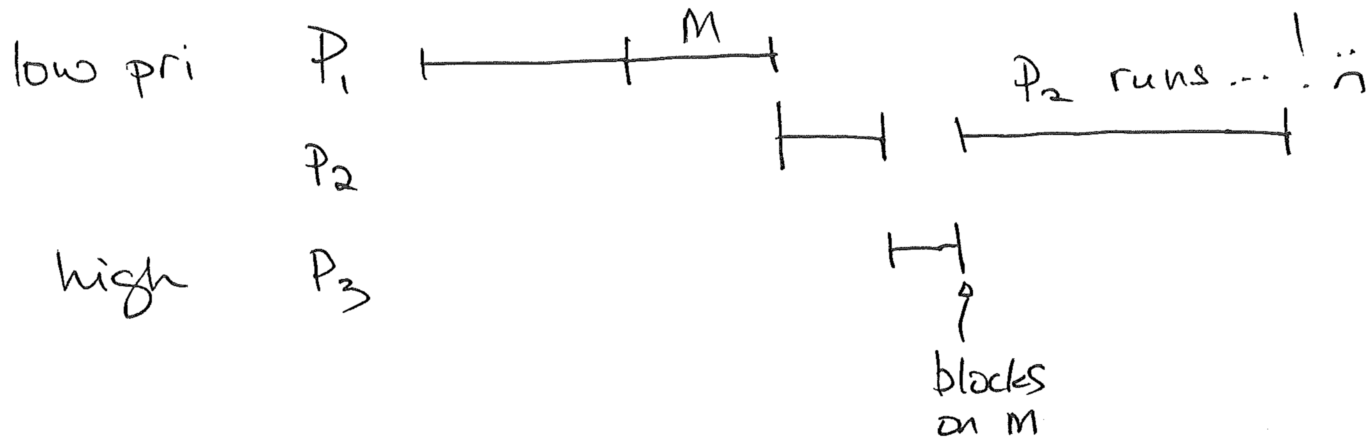
cv. signal() (naked notify) → while (! condition)
cv. wait(); ← stuck forever!

Solution: Add state - binary semaphore

notify: set to 1

wait: continue if 1; clear state

13. How can "priority inversion" occur with monitors? How can it be avoided?



Good solutions:

process grabbing M gets higher priority (static - highest of process known to use it)

or

dynamic - pri of process wanting resources (inheritance)

- See in lottery scheduling paper

14. How can deadlock occur with monitors and condition variables? How can these problems be fixed?

a)

```
if (cond) {
    wait();
}
```

```
if (cond2) {
    wait();
}
```

- no signal
- prog. error

b)

$\frac{M}{N}$ can't enter

$\frac{N}{M}$ can't enter

true cyclic dependency

→ impose ordering

M
N

M
N

c)

M
N
-wait (releases N lock)

M can't enter M
N
signal

1) Could release M lock (but tricky)

2) Separate M into multiple parts

M'
N
M''

3) General rule:
Don't embed monitor calls

15. Conclusions?

- Monitors + CV are here
- Sometimes language support for monitors
  - sometimes just explicit locks w/ CV
- Run into interesting issues when <u>implement</u> concepts