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

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

- Organize data and code in structured fashion (unifies it)
- Neat contrast
- Separates mutual exclusion and 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 at 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 proposed by Hoare, how would you implement synchronization for a bounded buffer?

```java
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--;
        nonfull.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:

- \( \text{enqueue} \) P

Eq: C1 begins + waits
- P fills
  - C2 arrives, waits on monitor
  - P signals \( \rightarrow \) Remove C1 from cv queue + add to monitor Q
  - C2 might get monitor lock next
    - it will see count == 1 & use buffer
  - When C1 acquires 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 hint (okay to be sloppy)

- Change all * 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?

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

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

What could go wrong? Process 2 frees 8 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 becast (notify ALL)
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?

```c
reader_count = 0;
busy = false;  // (writing)

start_read() {
  if (busy || OK_to_write_queue) {
    OK_to_read_wait();
    reader_count ++;
    OK_to_read_signal();  // used for multiple readers to get going
  }
  end_read();
  reader_count --;
  if (!reader_count) OK_to_write_signal();
}

start_write() {
  if (reader_count || busy) {
    OK_to_write_wait();
  }
  busy = true;
  end_write();
  busy = false;
  if (OK_to_read_queue) OK_to_read_signal();  // change to beast
  else OK_to_write_signal();
}
```
if (OKtoWrite, queue) & OK to read, wait() j

while (busy) & OK to read, wait() j

reader count ++
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

\[
\text{device} \xrightarrow{\text{CV. Signal()}} \xrightarrow{\text{CV. wait()}} \text{while (!condition)} \xrightarrow{\text{CV. wait()}} \text{stuck forever!}
\]

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) $M \xrightarrow[\neg \text{can enter}]{N}$ $N \xrightarrow[\neg \text{can enter}]{M}$

true cyclic dependency

→ impose ordering

$\langle M, M, M \rangle$

$c) \quad M
\quad N
-wait (releases N lock)
\quad \downarrow
\quad 1) \text{Could release M lock (but tricky)}$

2) Separate M into multiple parts

$M', N, M''$

3) General rule: Don't embed monitor calls
Monitors + cv are here

Sometimes language support for monitors
-sometimes just explicit locks
w/ cv

Run into interesting issues when implement concepts