Hoare vs. Mesa, I

As with test-and-set, semaphore, etc., there is more than one definition of monitor

"Hoare formulation" uses:

```
if (<data NOT in needed condition>)
    WAIT(<condition variable for that condition>)
```

"Mesa formulation" uses:

```
while (<data NOT in needed condition>)
    WAIT(<condition variable for that condition>)
```

Hoare vs. Mesa, II

Beyond the *monitor invariant*, it is typical for monitor procedures to have an *application-specific condition* that must be satisfied before they can proceed

Programmer places

```
if (not app-specific condition)
     then wait(cv)
```

at start of procedure (Hoare formulation)

Wait ends when another monitor procedure executes **signal** (or **broadcast**) with same `cv` as argument—this will awaken some waiter

Hoare vs. Mesa, III

Waiters and signalers agree to perform all communication about the establishment of the application-specific condition via `wait` and `signal` operations on the relevant condition variable

Since at most 1 thread can be "in the monitor," what to do with signaling thread?

(Waiter will awaken in the monitor, making both signaler and waiter seemingly "in the monitor" simultaneously)

Hoare vs. Mesa, IV

Alternatives:

1. Halt signaler & run signaled thread *immediately*. (Hoare formulation.)

   Note: when a thread is halted this way, it is not considered "in the monitor" because monitor-aware thread scheduler will ensure it doesn't run again until signaled thread is out of monitor.

2. Require signaler to exit the monitor immediately after signaling—merely a convention. (Mesa formulation.) Here, "in the monitor" means: accessing monitor data.

Mesa Formulation: Thread Scheduling Policies

Which waiter does signal wake up? FCFS? Random?

Thread scheduler's policy for which signal-ee to wakeup should be purposely left *unspecified*—user can assume *nothing*

Implementor is free to choose policy that is easiest, or fastest, or best for certain circumstances

Mesa Monitors, I

In Hoare's formulation, programmer puts

```
if (not app-specific condition)
     wait(cv)
```

at start of monitor procedures that have application-specific consistency conditions on the data

This works because signaled procedure executes *immediately*—the VERY NEXT ONE to execute—after signaler executes `signal()`

But ... this assumes that thread scheduler knows about monitors & cooperates

Mesa Monitors, II

Q: How to have monitors when ...

1. Language doesn't have monitor primitive (e.g., C)

2. OS thread scheduler doesn't/can't cooperate (e.g., UNIX)

A: Pthreads provides one example

(Mesa begat C-Threads begat Pthreads)

Pthreads Condition Primitives

Pthreads API for condition variables:

```
pthread_cond_init(pthread_cond_t *c,
                                pthread_cond_attr_t *attr)

pthread_cond_destroy(pthread_cond_t *c)

pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)

pthread_cond_signal(pthread_cond_t *c)

pthread_cond_broadcast(pthread_cond_t *c)
```

Use of Pthreads Mesa-style Condition Variables, I

Operation of `pthread_cond_wait(c, m)`:

1. Set thread state to "waiting"

2. Add thread to set of threads waiting for condition variable `c`

3. `pthread_mutex_unlock(m)`

4. Some time later: thread state set to "ready" by another thread's signal

5. Eventually, scheduler will pick this thread to run; then its state will be "running"

6. `pthread_mutex_lock(m)`

Operation of `pthread_cond_signal(c)`:

- Select thread from `c`'s wait-set according to policy & set its state to "ready"

Use of Pthreads Mesa-style Condition Variables, II

```
pthread_mutex_t mutex;        // explicit monitor lock

pthread_cond_t spaces, items;   // condition variables

// this is a monitor procedure
void produce() {
    pthread_mutex_lock(&mutex);    // get monitor lock
    while (<there is no space>)              // first action:
        pthread_cond_wait(&spaces, &mutex); // verify state

    ... produce ...                // change state

    pthread_cond_signal(&items);   // last action:
                                      signal state change
    pthread_mutex_unlock(&mutex);  // drop monitor lock

    return;
}
```

Use of Pthreads Mesa-style Condition Variables, III

Mutex implemented by programmer, not language, enforces monitor invariant

Data must be in state associated with app-specific condition "`spaces`" BEFORE `produce` operation runs, and will be in state associated with app-specific condition "`items`" AFTER operation runs

Use of Pthreads Mesa-style Condition Variables, IV

In Hoare's formulation, after `signal()`, signaled thread is guaranteed next to run

Therefore, wait with:

```
if (<not condition>)
    wait();
```


Use of Pthreads Mesa-style Condition Variables, V

In Pthreads for C/UNIX ...

1. Other threads may run between signal-er and signal-ee

   Between when thread A's `signal()` awakens thread B and when thread B waits for the lock, some thread C may have run and undone the application-specific condition that A established for B

Use of Pthreads Mesa-style Condition Variables, VI

2. Therefore, condition must be re-checked when signal-ee awakens
3. Therefore, waiters wait with:

```
while (<not condition>)
    wait();
```

Example, I

Ideal execution:

0. Thread A holds lock & is in monitor procedure
1. Thread A establishes application-specific condition on monitor data
2. Thread A signals appropriate condition variable, thereby changing state of Thread B from waiting to ready
3. Thread A drops lock
4. Thread A returns from monitor procedure
5. Thread B (which is in pthread_cond_wait) is scheduled
6. Thread B tries to get lock
7. Thread B succeeds in getting lock & its pthread_cond_wait returns
8. Thread B tests application-specific condition in test of while statement
9. Test is passed & thread B enters monitor procedure

Example, II

In Mesa formulation (OS thread scheduler doesn't know about monitors), what COULD happen:

```
... same as above, steps 0 thru 4
4.25. Thread C is scheduled
4.50. Thread C gets lock, enters a monitor procedure
      and changes monitor data so as to UNDO thread B's
      application-specific condition
4.75. Thread C drops lock & leaves its monitor procedure
5. Thread B (which is in pthread_cond_wait) is scheduled
6. Thread B tries to get lock
7. Thread B succeeds in getting lock & its pthread_cond_wait
   returns
8. Thread B tests application-specific condition in test of
   while statement
9. Test FAILS!
10. Thread B loops & calls pthread_cond_wait again
```

Intervention by thread C shows why thread B must re-test application-specific condition after returning from wait