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

Starvation

In example above, for thread B to eventually enter monitor procedure all these must happen:

a. Some thread A gets into monitor

b. Thread A establishes B's application-specific condition

c. Thread A signals condition B is waiting for

d. B is awakened WITHOUT an intervening thread undoing the condition as Thread C did in example above

There is NO guarantee that all this will eventually happen—B may be starved

(E.g., suppose d never happens ... there is always some intervening thread that undoes condition)

Use of Pthreads Mesa-style Condition Variables, VII

Above *conventional* use of `wait` and `signal` provides proper monitor entry & exit

Costs are:

1. Extra evaluation of condition (after return from wait)

2. Possible starvation, since no scheduler cooperation

Benefits are:

1. Decouples OS thread scheduler policy from monitors—can have monitors without language or OS support

2. Does not require forced (and possibly non-optimal) Hoare-style immediate switch to signaled thread

How to Program with Monitors, I

1. Identify logically related variables that might be accessed concurrently & place them inside monitor

2. Determine possible states for these variables

3. Determine state transitions that might occur concurrently

4. Write each state transition as a monitor procedure

How to Program with Monitors, II

Data can now be transitioned from one state to the next safely

Each monitor procedure performs a state transition following this pattern:

```
while (data not in state A)
    wait(cv for state A)
<transition data from state A to state B>
signal(cv for state B)
```

Signal vs. Broadcast, I

When to use broadcast?

pthread_cond_broadcast man page says:

The pthread_cond_broadcast() function is used whenever the shared-variable state has been changed in a way that more than one thread can proceed with its task.

Consider a single producer/multiple consumer problem, where the producer can insert multiple items on a list that is accessed one item at a time by the consumers.

By calling the pthread_cond_broadcast() function, the producer would notify all consumers that might be waiting, and thereby the application would receive more throughput on a multiprocessor.

Signal vs. Broadcast, II

Signal vs. broadcast is only a performance optimization

For code to be correct, it must ALWAYS be possible to change any signal to a broadcast (or vice versa)