Synchronization Primitives

- **Locks**
  - Provide *mutual exclusion*
  - 2 operations: `acquire()` and `release()`

- **Semaphores**
  - Generalize locks with an integer count variable and a thread queue
  - 2 operations: `wait()` and `signal()`
  - If the integer count is negative, threads wait in a queue until another thread signals the semaphore

- **Monitors**
  - An abstraction that encapsulates shared data and operations on it in such a way that only a single process at a time may be executing “in” the monitor

Thursday, 24 January, 13
MORE ON MONITORS

- Programmer defines the scope of the monitor
  - ie: which data is “monitored”

- Local data can be accessed only by the monitor’s procedures (not by any external procedures)

- Before any monitor procedure may be invoked, mutual exclusion must be guaranteed
  - There is often a lock associated with each monitored object

- Other processes that attempt to enter the monitor are blocked. They must first acquire the lock before becoming active in the monitor
COMPLICATIONS WITH MONITORS

Complication

- A process may need to wait for something to happen
  - Input from another thread might be necessary for example
- The other thread may require access to the monitor to produce that event

Solution?

- Monitors support suspending execution within the monitor
  - wait() (suspend the invoking process and release the lock)
  - signal() (resume **exactly one** suspended process)
  - broadcast() (resumes **all** suspended processes)
    - If no process is suspended, signal/broadcast has no effect
      - (in contrast to semaphores, where signal **always** changes state of the semaphore)
Monitor Signal()
Who goes first?

Suppose P executes a signal operation that would wake up a suspended process Q
- Either process can continue execution, but both cannot simultaneously be active in the monitor

Who goes first?
- **Hoare** monitors: waiter first
  - `signal()` immediately switches from the caller to a waiting thread
  - Condition that the waiter was blocked on is guaranteed to hold when the waiter resumes
- **Mesa** monitors: signaler first
  - `signal()` places a waiter on the ready queue, but signaler continues inside the monitor
  - Condition that the waiter was blocked on is **not** guaranteed to hold when the waiter resumes (must check again...)

Thursday, 24 January, 13
HOARE VS. MESA MONITORS

- Hoare monitor wait
  if(...){
    wait(cv, lock);
  }

- Mesa monitor wait
  while(...){
    wait(cv, lock);
  }

- Tradeoffs
  - Hoare monitors are easier to reason with, but hard to implement
  - Mesa monitors are easier to implement, and support additional operations like `broadcast()`
We have a buffer of limited size N
  - Producers add to the buffer if it is not full
  - Consumers remove from the buffer if it is not empty

Want to control buffer as a monitor
  - Buffer can only be accessed by methods that are “part of” the monitor, that only give one producer or consumer access to the buffer at a time

Need 2 functions
  - add_to_buffer()
  - remove_from_buffer()

Need
  - One lock
  - Two conditions
    - One for producers to wait
    - One for consumers to wait
#define N 100

typedef struct buf_s {
    int data[N];
    int inpos; /* producer inserts here */
    int outpos; /* consumer removes from here */
    int numelements; /* # items in buffer */
    struct lock *mylock; /* access to monitor */
    struct cv *notFull; /* for producers to wait */
    struct cv *notEmpty; /* for consumers to wait */
} buf_t;

buf_t buffer;
void add_to_buff(int value);
int remove_from_buff();
void add_to_buf(int value) {
    lock_acquire(buffer.mylock);
    while (nelements == N) {
        /* buffer is full, wait */
        cv_wait(buffer.notFull, buffer.mylock);
    }
    buf.data[inpos] = value;
    inpos = (inpos + 1) % N;
    nelements++;
    cv_signal(buffer.notEmpty, buffer.mylock);
}

What kind of monitor is this?
int remove_from_buf() {
    int val;
    lock_acquire(buffer.mylock);
    while (nelements == 0) {
        /* buffer is empty, wait */
        cv_wait(buffer.notEmpty, buffer.mylock);
    }
    val = buf.data[outpos];
    outpos = (outpos + 1) % N;
    nelements--;
    cv_signal(buffer.notFull, buffer.mylock);
    lock_release(buffer.mylock);
}
Now let’s look at a fully-implemented Pthreads solution for bounded buffers...