Locks & CVs

- **Lock issues**
  - A thread cannot Acquire a lock it already holds
  - A thread cannot Release a lock it does not hold
  - A lock cannot be deleted if a thread is holding it

- **Condition Variable issues**
  - A thread can only call Wait and Signal if it holds the mutex
  - Wait must Release the mutex before the thread sleeps
  - Wait must Acquire the mutex after the thread wakes up
  - A condition variable cannot be deleted if a thread is waiting on it
Mailboxes

- Senders and receivers need to be synchronized
  - One sender and one receiver need to rendezvous
- Issues
  - Block all other senders while waiting for receiver in Send
  - Block all other receivers while waiting for sender in Receive
  - When a condition variable is signaled…
    » The waiting thread is placed on the ready list
    » But it has not necessarily re-acquired the lock
    » It only reacquires the lock when it runs again
    » If another thread runs before it does, that thread can acquire the lock before the waiter does
    » Let’s look at an example

Synchronizing with Wait/Signal

```
while (1) {
  mutex->Acquire();
  printf("ping\n");
  cond->Signal(mutex);
  mutex->Release();
}

while (1) {
  mutex->Acquire();
  cond->Wait(mutex);
  printf("pong\n");
  mutex->Release();
}
```

Signal places waiter on ready list, and then continues

BUT – the waiter now competes with the signaler to re-acquire the mutex

Output COULD be: ping…ping…ping
Interlocking with Wait/Signal

Waiting after signaling interlocks the two threads.
The thread that signals then does a wait, and cannot proceed until the other thread wakes up from its wait and follows with a signal.

Thread::Join

- Issues
  - A thread can only be Joined if specified during creation
  - A thread can only be Joined after it has forked
  - Only one thread can call Join on another
  - A thread cannot call Join on itself
  - A thread should be able to call Join on a thread that has already terminated
    - This is the tricky part
    - Should delay deleting thread object if it is to be joined
      - If it is not going to be Joined, then don’t change how it is deleted
    - Where is it deleted now? Look for use of threadToBeDestroyed
    - Where should joined threads be deleted?
    - Need to delete synch primitives used by Join as well
Thread::setPriority(int)

- Issues
  - Priorities have the entire range of an “int”
    - Both negative and positive
  - If one thread has a priority value that is greater than another, that thread has a higher priority (simple integer comparisons)
  - List implementation in list.cc has sorting capabilities
  - Only adjust priority of thread when it is placed on ready list
  - When transferring priority from a high thread to a low thread, the transfer is only temporary
    - When the low thread releases the lock, its priority reverts

Mating Whales

- Issues
  - This is a synchronization problem like Bounded-Buffer, Readers/Writers, and Smoking Barber
  - You do not need to implement anything inside of Nachos
    - But you will use the synchronization primitives you implemented
    - You can use any synch primitives you want
  - You will implement Male, Female, and Matchmaker as functions in threadtest.cc (or equivalent), and create and fork threads to execute these functions in ThreadTest:
    ```
    T1->Fork(Male, 0);  // could fork many males
    T2->Fork(Female, 0); // could fork many females
    T3->Fork(Matchmaker, 0); // could fork many matchmakers
    ```
  - There is no API -- we will compile, run, and visually examine your code for correctness
  - Comments will help (both you and us)
Tips

- Use DEBUG macro to trace the interaction of the synchronization primitives and thread context switches
  - Run "nachos -d s -d t" to enable synch and thread debugs
- Read the Nachos Road Map, Experience With Nachos Assignments, Synchronization for some good advice
  - http://www.cs.duke.edu/~narten/110/nachos/main/node32.html #SECTION0007200000000000000000