Announcements:

- Guest lecture next Tuesday 10/30, Professor Tate
- Quiz #3 probably on Thursday 10/25, might not be returned until 10/31

- Mutual exclusion

- A basic principle of concurrent programming is that reading and writing of mutable shared variables must be *synchronized*

    - so that shared data is used and modified in a predictable sequential manner by a single process,

    - rather than in an unpredictable interleaved manner by multiple processes at once.

- The term *critical section* is commonly used to refer to code which accesses a shared variable or data structure that must be protected against simultaneous access.

- The simplest means of protecting a critical section is to block any other process from running until the current process has finished with the critical section of code.

- This is commonly done using a *mutual exclusion lock* or *mutex*.
- There actually needs to be hardware support for this efficiently
    - Atomic (uninterruptable) operation to test and set a memory location.
    - Intel: XCHG operation, swap memory with register. Store 1 in register, then XCHG. If the register has a 0 you have the lock.

- Metaphors: Ithaca one-lane bridges (over Fall creek); train tracks; airport runways (compare with cars, centralized vs distributed)

- A *mutex* is a program object which only one party at a time can have control over. In OCaml mutexes are provided by the Mutex module. The signature for this module is:

```
module type Mutex = sig
  type t
  val create : unit -> t
  val lock: t -> unit
  val try_lock: t -> bool
  val unlock: t -> unit
end
```

- `Mutex.create` creates a new mutex and returns a handle to it.

- `Mutex.lock m` returns once the specified mutex has been successfully locked by the calling thread.
    - Can cause thread to block!

- If the mutex is already locked by some other thread then the current thread is suspended until the mutex becomes available.

- `Mutex.try_lock m` returns `true` if the specified mutex has been successfuly locked by the current thread, and `false` if it is already locked by some other thread.
    - Returns immediately!

- `Mutex.unlock m` unlocks the specified mutex, which in turn causes other threads suspended trying to lock `m` to restart (and only one of those threads will successfully get the lock).

- `Mutex.unlock` throws an exception if the current thread does not have the specified mutex locked.

- If all the code that access some shared data structure acquires a given mutex before such access, and releases it after the access, then this guarantees access by only one process at a time.

```
Mutex.lock m;
foo(d); (* Critical section operating on some shared data structure *)
Mutex.unlock m
```

- We commonly refer to the mutex `m` as protecting the data structure `d`.
    - Note that this protection is only guaranteed if all code that access `d` correctly obtains and releases the mutex.
- Now we can rewrite the function `prog1` above to use a mutex to protect the critical section that reads and modifies the shared variable `result`:

```
(* Correct code below, result always increases *)

let prog2 (n) =
  let result = ref 0 in
  let m = Mutex.create() in
  let f (i) =
    for j = 1 to n do
      Mutex.lock m;
      let v = !result in
          Thread.delay(Random.float 1.0); result := v+i;
     print_string("Value " ^ string_of_int(!result) ^ "\n");
     flush stdout;
     Mutex.unlock m;
     Thread.delay(Random.float 1.0)
    done
  in
    ignore (Thread.create f 1);
    ignore (Thread.create f 2)
```

- Too much locking with mutexes results in code not being concurrent.
- In fact use of excessive locking can result in code that is slower than a single-threaded version.
- That said, however, sharing variables across threads without proper synchronization will yield unpredictable behavior!

- There is closely related behavior in the design of operating systems
    - Perform an operation “without interrupts”
        - I.e. lock the CPU for yourself
    - See CS4110

- Sometimes that behavior will only occur very rarely.
- Recent CS colloquium proposed trying lots of execution orders
- Concurrent programming is hard.
    - Often a good approach is to write code in as functional a style as possible as this minimizes the need for synchronization of threads.

- Another hazard of concurrent programming is the potential for deadlocks, where multiple threads have permanently prevented each another from running because they are waiting for conditions that cannot become true given that other threads are also waiting.
- A simple example of a deadlock can occur with two mutexes, call them *m* and *n*.
- Say one thread tries to lock *m* and then *n*, whereas another thread tries to lock *n* and then *m*.
- If the first thread has succeeded in locking *m* and the second thread has succeeded in locking *n*, then no forward progress can ever be made because each is waiting on the other (this is sometimes referred to as deadly embrace).

- Other approaches:

    - Message passing to someone who holds the shared resource
    - Software synchronization
        - tends to be expensive and requires busy waiting
        - requires precise ordering of memory access within a thread

- Condition variables are used when one thread wants to wait until another thread has finished doing something: the former thread ``waits'' on the condition variable, the latter thread ``signals'' the condition when it is done.
- They will be covered in section

```
(* Reader/writer; a classic concurrency pattern (concurrent readers and
 * one exclusive writer, CRXW).  There is mutual exclusion between a
 * single writer and any of many readers, but readers can operate at
 * the same time (because they do not change any shared state).
 *
 * This is accomplished with a shared variable n which counts the
 * number of readers currently active. Each reader momentarily acquires
 * the mutex to increment the count, then does their work, then
 * momentarily acquires the mutex to decrement the count.
 *
 * The writer needs to wait until there are no readers.  This is
 * achieved using a condition variable to signal when no are readers
 * active.  The writer waits for the condition to be true.  The readers
 * signal the condition if when they finish there are no readers
 * active.
 *
 * Such waiting on a condition before taking a mutex is known as a
 * semaphore.
 * )
```