Announcements:

- PS4 due Thursday 10/18 11:59PM
    - Reduced late penalty (5% per day)
    - Grace day possible on PS4/5/6
- Anonymous survey out soon

- Guest lecture Thursday by Yaron Minsky, Jane Street Capital
- Guest lecture by Prof Ross Tate on 10/30

```
let ctr1 =
  let v = ref 0 in
  fun(x) -> (v := !v + x; v) (* Or !v to be functional *)

let ctr2 =
  fun(x) ->
  let v = ref 0 in
    (v := !v + x; v)

let ctr3 =
  let v = ref [0] in
  fun(x) -> (v := [List.hd(!v) + x]; v) (* List.hd(!v) to be functional *)

let ctr4 =
  let v = [ref 0] in
  fun(x) -> (List.hd(v) := !(List.hd v) + x; v) (* !List.hd(v) to be
functional *)
```

Concurrency

- So far in this class we've been talking about sequential programs.
    - Execution of a sequential program proceeds one step at a time, with no choice about which step to take next.
- Sequential programs are somewhat limited
    - both because they are not very good at dealing with multiple sources of simultaneous input
    - And because they are limited by the execution resources of a single processor.
- For this reason, many modern applications are written using parallel programming techniques.

- There are many different approaches to parallel programming
    - They all share the fact that a program is split into multiple different processes that run at the same time.
- Each process runs a sequential program,
    - But the collection of processes no longer results in a single overall predictable sequence of steps.
- Rather, steps execute *concurrently* with one another,
    - Resulting in potentially unpredictable order of execution for certain steps with respect to other steps.

- The granularity of parallel programming can vary widely,
    - from coarse-grained techniques that loosely coordinate the execution of separate programs, such as *pipes* in Unix
        - or the http protocol between a Web server and its clients
    - To fine-grained techniques where concurrent code shares the same memory, such as *lightweight threads*.

- In both cases it is necessary to coordinate the execution of multiple sequential programs.

- Two important types of coordination are commonly used:
    - **Synchronization**, where multiple processes wait for certain conditions.
    - **Communications**, where messages are passed between processes.

- In this lecture we will consider the lightweight thread mechanism in OCaml.

- The threads library provides concurrent programming primitives for multiple threads of control which execute concurrently in the same memory space.

- Threads communicate by modifying shared data structures or by sending and receiving data on communication channels.

- The threads library is not enabled by default. Compilation using threads is described in the threads library documentation.

- It should be noted that the OCaml threads library is implemented by time-sharing on a single processor
    - Does not take advantage of multi-processor machines.
- Thus the library will not make programs run faster

- However often programs are easier to write when structured as multiple communicating threads.

- For instance, most user interfaces concurrently handle user input and the processing necessary to respond to that input.

- A user interface that does not have a separate execution thread for user interaction is highly frustrating to use

    - Because it does not respond to the user in any way until a current action is completed.

    - For example, a web browser must be simultaneously:
        - handling input from the user interface,
        - reading and rendering web pages incrementally as new data comes in, and
        - Running programs embedded in web pages.

    - All these activities must happen at once, so separate threads are used to handle each of them.

- Another example of a naturally concurrent application is a web crawler, which traverses the web collecting information about its structure and content.
    - It doesn't make sense for the web crawler to access sites sequentially,
        - Most of the time would be spent waiting for the remote server and network to respond to each request.
    - Therefore, a typical web crawler is highly concurrent, simultaneously accessing thousands of different web sites.
    - This design uses the processor and network efficiently.

- Concurrency is a powerful language feature that enables new kinds of applications,
- But it also makes writing correct programs more difficult,
    - because execution of a concurrent program is nondeterministic:
    - The order in which things happen is not known ahead of time.
- The programmer must think about all possible orders in which the different threads might execute,
    - And make sure that in all of them the program works correctly.

- If the program is purely functional, nondeterminism is easier because evaluation of an expression always returns the same value
    - For example, the expression `(2*4)+(3*5)` could be executed concurrently, with the left and right products evaluated at the same time. The answer would not change.
- Note that many programming languages do not specify the order of argument evaluation
    - Why is this?
- Imperative programming is much more problematic.
    - For example, the expressions `(!x)` and `(a := !a+1)`, if executed by two different threads, could give different results depending on which thread executed first, if it happened that `x` and `a` were the same ref.
- An expression will thus have a set of possible values!

- A simple example

- Let's consider a simple example using multiple threads and a shared variable, to illustrate how what would be straightforward in a sequential program produces quite unexpected results in a concurrent program.
- A partial signature for the Thread module is

```
module type Thread = sig
  type t
  val create : ('a -> 'b) -> 'a -> t
  val self: unit -> t
  val id: t -> int
  val delay: float -> unit
end
```

- `Thread.create f a` creates a new thread in which the function `f` is applied to the argument `a`, returning the handle for the new thread as soon as it is created (not waiting for `f` to be run).

- The new thread runs concurrently with the other threads of the program. The thread exits when `f` exits (either normally or due to an uncaught exception).
    - `Thread.self()` returns the handle for the current thread, and `Thread.id(t)` returns the identifier for the given thread handle.
    - `Thread.delay(d)` causes the current thread to suspend itself (stop execution) for `d` seconds.
    - There are a number of other functions in the `Thread` module, however note that a number of these other functions are not implemented on all platforms.

- Now consider the following function, which defines an internal function `f` that simply loops `n` times, and on each loop increments the shared variable `result` by the specified amount, `i`, sleeping for a random amount of time up to one second in between reading `result` and incrementing it.
    - The function `f` is invoked in two separate threads, one of which increments in by 1 on each iteration and the other of which increments by 2.

```
let prog1 (n) =
  let result = ref 0 in
  let f (i) =
    for j = 1 to n do
      let v = !result in Thread.delay(Random.float 1.0); result := v+i;
   print_string("Value " ^ string_of_int(!result) ^ "\n");
   flush stdout

    done
  in
    ignore (Thread.create f 1);
    ignore (Thread.create f 2)
```

- Viewed as a sequential program, this function could never result in the value of `result` decreasing from one iteration to the next,
    - As the values passed in to `f` are positive, and are added to `result`.
    - However, with multiple threads, it is easy for the value of `result` to actually *decrease*.
    - If one thread reads the value of `result`, and then while it is sleeping that value is incremented by another thread, that increment will be overwritten, resulting in the value decreasing.
- For instance:

```
# prog1(10);;
Value 2
Value 1
Value 4
Value 2
Value 6
Value 3
Value 8
Value 4
Value 10
Value 5
Value 12
Value 6
Value 14
Value 7
Value 16
Value 18
Value 8
Value 9
Value 10
Value 20
- : unit = ()
```

- It is important to note that this same issue exists even without the thread sleeping between the time that it reads and updates the variable `result`.
    - The sleep increases the chance that we will see the code execute in an unexpected manner,
    - The simple act of incrementing a mutable variable inherently needs to first read that variable, do a calculation and then write the variable.
    - If a process is interrupted between the read and write steps by some other process that also modifies the variable, the results will be unexpected.

- 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
    - 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.

- 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.
- 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.
- `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`:

```
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)
```

- This function has the expected behavior of always incrementing the value of `result`.

- 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!

- Sometimes that behavior will only occur very rarely.
- 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).

```
(* 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.
```