Announcements:

- PS4 due Thursday Oct 20, 11:59PM
- Partner up for PS5!
    - Design reviews, TBD in section Monday
- Anonymous survey out soon
- Quiz #4 on 10/27 at start of class
- Prelim #2 on evening of Tue 11/15, review session the night before

- Guest lecture: “Effective OCaml” on Thu 11/3 by Yaron Minsky, Jane Street Capital
- Guest lecture on Tue 11/22 (right before Thanksgiving break)

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.

- 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.