Announcements:

- Quiz #4 on 11/1 at start of class
- No RDZ office hours next week or week after due to travel
    - You can always see me by appointment
- Prelim #2 on evening of Tue 11/15, review session the night before

- Guest lecture: “Effective OCaml” on Thu 11/3 by Yaron Minsky
- Lectures the week of 11/8 will be give by Prof. Foster
- Guest lecture on Tue 11/22 (right before Thanksgiving break)

- Today’s topic: some important design patterns for concurrent programming
    - Producer/consumer
    - Thread pools
    - Beyond

Producer/Consumer and Thread Pools

- A classic concurrent programming design pattern is *producer-consumer*, where processes are designated as either producers or consumers.

- The producers are responsible for adding to some shared data structure
    - the consumers are responsible for removing from that structure.

- Only one party, either a single producer or a single consumer, can access the structure at any given time.

- Here we consider an example with a shared queue, using a mutex (introduced previously) to protect the queue

- **CODE AT END OF LECTURE**

- We divide the work of a producer into two parts:
  - `produce` which simulates the work of creating a product and
  - `store` which adds the product to the shared queue.

- `produce` increments the counter `p` that it is passed, sleeps for `d` seconds to simulate the time taken to produce, and outputs a status message:
- `store` acquires the mutex `m`, adds to the shared queue, outputs a status message and releases the mutex:
- The `producer` loops `n` times calling `produce` and then `store`, and then sleeping for a random amount of time up to 2.5 seconds.
  - When done looping it outputs a status message.

- The `consumer` loops `n` times, acquiring the mutex `m`, then attempting to take an item from the shared queue.
- If it succeeds it prints out the item, if not it prints out that it failed to get an item.
- In either event it unlocks the mutex and then waits a random amount of time up to 2.5 seconds.

- This use of mutual exclusion is very coarse grained.

- For instance it would be better to be able to have a consumer wait until something is in the queue, rather than returning empty handed.
    - Busy-waiting is never a good idea

- For this we can make use of condition variables, which were introduced previously.
    - Done in section

- Note: generally, a function that has the effect of locking (or unlocking) a mutex should be used with caution, and should be clearly documented as doing so.
    - This is a huge source of bugs!

Thread Pools

- A thread pool consists of a collection of threads, called workers that are used to process work.
    - Basic advantage: avoid overhead of thread creation
    - Similar to server farms
- Each worker looks for new work to be done, when it finds work to do it does it, and when finished goes back to get more work.
- The workers play the role of consumers in the producer-consumer model that we just considered above.
- In fact, thread pool implementations often use a shared queue to store the work, thus building quite directly on the previous example.

- Before considering implementation of thread pool, let's get a better idea of what it does and where it is useful.
- The basic operations for a thread pool are
    - to create a new thread pool with some specified number of workers,
    - to add work to an existing thread pool (which will subsequently be performed by one of the workers), and
    - to destroy an existing thread pool (shutting it down once all previously added work is complete).

- While deadlock is a risk in any multithreaded program, thread pools introduce another opportunity for deadlock, where all pool threads are executing tasks that are blocked waiting for the results of another task on the queue, but the other task cannot run because there is no unoccupied thread available.

- Here is the signature for a basic thread pool:

```
module type SIMPLE_THREAD_POOL = sig
  type pool
  (* A No_workers exception is thrown if addwork is called when the
     threadpool is being shut down.  The work is not added. *)
  exception No_workers
  (* create a thread pool with the specified number of worker threads *)
  val create: int -> pool
  (* add work to the pool, where work is any unit->unit function *)
  val addwork: (unit->unit) -> pool -> unit
  (* destroy a thread pool, stopping all the threads once all work
   * in the pool has been completed. *)
  val destroy: pool -> unit
end
```

- Thread pools are particularly useful in setting where work arrives asynchronously, such as occurs with a server where many network requests may need to be handled promptly.

- In such settings, a thread receives an event such as a network request,
  - adds the corresponding work to a thread pool (which will be run at some point in the future),
  - and then quickly returns indicating to the caller that the request will be handled.

- Sometimes it is also useful to have a handle associated with each unit of work to which some value is sent.
- The simple abstraction that we presented here does not have any means of returning a result,
  - as the functions representing work are of type `unit->unit`.

- Here we consider an implementation of the `SIMPLE_THREAD_POOL` interface in terms of a 4-tuple:
    - a mutable counter,
    - a mutable queue of functions that are the work remaining to be done,
    - a mutex, and
    - a condition variable.
- The mutex is used to protect the counter and the queue, and the condition variable is used to signal when a worker should wake up to get new work.

```
module Tpool : SIMPLE_THREAD_POOL = struct

  type pool = (int ref * (unit -> unit) Queue.t * Mutex.t * Condition.t)

  exception No_workers

  let dowork tp =  ...

  let create size =  ...

  let addwork f tp =  ...

  let rec done_wait tp n =  ...

  let destroy tp =  ...

end
```

- When the counter in the 4-tuple is a positive integer, then it indicates the number of worker threads that the thread pool was created with.
- When the counter is a non-positive integer then it indicates that the thread pool is being destroyed,
    - the absolute value of the counter is then the number of threads which have properly exited.
- This allows the destroy function to wait for the threads to finish their work and exit before returning.

- Each worker thread runs the function `dowork`.
- This function is not exposed in the interface and so can only be called from inside the implementation of `Tpool`.
- `dowork` loops as long as the thread pool is not finished, in which case it exits.
- A thread pool is finished when it is being destroyed and there also no work remaining to do.

- We use the counter in the 4-tuple, here called `nworkers`, to indicate that the thread pool is being destroyed by setting its value to something less than 1.
- In that case, if the queue of work is also empty then the thread exits as the pool is finished.
- Otherwise, on each loop the worker waits for work to do, and then takes that work from the queue, executing it inside a try to ensure that unhandled exceptions in the work do not cause the worker to exit.

```
let dowork (tp:pool) =
  match tp with (nworkers, q, m, c) ->
    Mutex.lock m ;
    (* When nworkers <=0 it means the thread pool is being
     * destroyed.  If that is true and there is also no work left to do
     * then stop looping and drop through to exit processing.*)
    while (!nworkers > 0) || (Queue.length q > 0) do
      (* In normal operation where nworkers>0 wait for stuff in the queue. *)
      while (!nworkers > 0) && (Queue.length q = 0) do
        dbgprint "waiting";
        Condition.wait c m
      done;
      (* Verify something in the queue rather than we are now being
       * shut down, then get the work from the queue, unlock the
       * mutex, do the work and relock the mutex before looping
       * back. *)
      if (Queue.length q  > 0)
      then
        let f = Queue.take q in
          dbgprint "starting work";
          Mutex.unlock m;
          (* Don't let an exception in the work, f, kill the thread,
           * just catch it and go on. *)
          (try ignore (f()) with _ -> ());
          Mutex.lock m
    done;
    (* A worker thread exits when the pool is being shut down.  It
     * decrements the worker count which when all threads are
     * finished should be -n, where n was the number of threads in
     * the pool (counts down from 0). *)
    nworkers := !nworkers-1;
    dbgprint "exiting";
    Mutex.unlock m
```

- Note the use of the mutex in the 4-tuple, called `m` here, to protect accesses to the queue and the counter.
- Before entering the while loop, and at the end of the while loop (thus before the next iteration of the loop), the mutex must be locked because both the counter and the queue are accessed.
- The nested while loop checking for work to do uses `Condition.wait` to release the mutex and sleep until it receives the condition of work being ready.
    - Recall that this reacquires the mutex before returning.
- It is important that the mutex is then unlocked before calling `f`, the work to be done.
    - This allows other workers to safely run concurrently, as `f` cannot access the queue or counter.
- Then the mutex is reacquired before the end of the while loop.
- When exiting the while loop, the mutex is already locked,
    - so the counter is simply decremented and then the mutex is released before the worker exits.

- The `create` function makes a 4-tuple, and starts up the specified number of threads, each of which runs the `dowork` function. `create` simply acquires the mutex at the beginning and releases it at the end. Other ways of writing this code are possible, where the mutex is not held during the entire process of creating the thread pool.

```
let create size =
  if size <1 then raise(Failure "Tpool create needs at least one thread")
  else
    let tp = (ref 0, Queue.create(), Mutex.create(), Condition.create()) in
    match tp with (nworkers, _, m, _) ->
      Mutex.lock m;
      while !nworkers < size do
        ignore(Thread.create dowork tp);
        nworkers := !nworkers+1
      done;
      Mutex.unlock m;
      tp
```

- The `add_work` function adds the given function to the queue. In doing so it first locks the mutex and checks whether the pool is being destroyed. If the pools is being destroyed instead of adding the work it throws the `No_workers` exception, after first releasing the mutex. If the pool is not being destroyed it adds the qork to the queue, signals there is work to be done, and unlocks the mutex.

```
let addwork (f:unit->unit) (tp:pool) =
  match tp with (nworkers, q, m, c) ->
    Mutex.lock m ;
    if !nworkers <1
    then (Mutex.unlock m; raise No_workers)
    else
   (Queue.add f q ;
    Condition.signal c;
    Mutex.unlock m )
```

- The `destroy` function acquires the mutex and then sets the number of workers to zero to indicate that the thread pool is being shut down. It then broadcasts to all the workers that there is something to be done, to ensure that all the workers exit including ones that were currently sleeping awaiting work. Finally it releases the mutex and then waits for the workers to all exit, using the helper function `done_wait`.

```
let destroy (tp:pool) =
  match tp with (nworkers, _, m, c) ->
    Mutex.lock m;
    let n = !nworkers in
   nworkers := 0;
   Condition.broadcast c;
   Mutex.unlock m;
   done_wait tp n
```

- Inter-process communication (IPC) generalizes threads
    - Imagine threads running on different machines, say in the cloud
    - Instead of shared memory consider “shared nothing”
    - Harder case than threads: distributed system with unreliable communications (such as TCP/IP)

- Communication via message passing instead of shared memory

- Definition of a “distributed system”
    - Failure due to a computer you’ve never heard of

- How can two computers coordinate to do something?
    - Example: one-lane road with stoplights at each end
    - Agree to “northbound traffic for 5 minutes at 9AM”
        - Master/slave via shared memory

- Difficulty of coordination with unreliable communications

```
let f = Queue.create() and m = Mutex.create ()

let produce i p d =
  incr p;
  Thread.delay d;
  print_string("Producer " ^ string_of_int(i) ^
            " has produced " ^ string_of_int(!p) ^ "\n");
  flush stdout

let store i p =
  Mutex.lock m ;
  Queue.add (i,!p) f ;
  print_string("Producer " ^ string_of_int(i) ^
            " has added its " ^ string_of_int(!p) ^ "-th product\n");
  flush stdout;
  Mutex.unlock m

let producer (n,i) =
  let p = ref 0
  and d = Random.float 2. in
    for j = 1 to n do
      produce i p d ;
      store i p ;
      Thread.delay (Random.float 2.5)
    done;
    print_string("Producer " ^ string_of_int(i) ^
            " is exiting.\n");
    flush stdout

let consumer (n,i) =
  for j = 1 to n do
    Mutex.lock m ;
    ( try
      let ip, p = Queue.take f
      in
        print_string("Consumer " ^ string_of_int(i) ^
                 " has taken product (" ^ string_of_int(ip) ^
                 "," ^ string_of_int(p) ^ ")\n");
        flush stdout
       with
        Queue.Empty ->
          print_string("Consumer " ^ string_of_int(i) ^
                   " has returned empty-handed\n");
          flush stdout);
    Mutex.unlock m ;
    Thread.delay (Random.float 2.5)
  done;
  print_string("Consumer " ^ string_of_int(i) ^
            " is exiting.\n");
  flush stdout
```