Thread Design Patterns

- Common ways of structuring programs using threads
  - you will need many of them for the project

High-Level Program Structure Ideas

A few common patterns have emerged that are useful in different threads applications

- **Boss/workers** model
  - boss gets assignments, dispatches tasks to workers
  - many variants (thread pool, single thread per connection, etc.)

- **Pipeline** model
  - do some work, pass the partial result to the next thread

- **Up-calls**
  - fast (or even synchronous) control flow transfer for layered systems

- **Version stamps**
  - a common technique for keeping shared information consistent

Classical Boss/Workers

- **Boss:**
  
  ```
  forever {
    get a request
    switch (request)
      case X: Fork (taskX)
      case Y: Fork (taskY)
    ...
  }
  
  Advantage: simplicity
  
  Disadvantages:
  - no bound on number of workers
    - (sidenote: Sun Pthreads implementation has a default 1MB stack per thread)
  - potential for contention if requests have interdependencies
  - overhead of thread creation for every request, although the threads will later die
  
  ```

- **Worker:**

  ```
  taskX() { do work, synchronize as needed }
  ```

Variants of Boss/Workers

- Many variants possible. For instance, boss/workers with a fixed thread pool (aka workpile, work queue):
  - all worker threads are created up front (a fixed number of them)
  - producer/consumer relation between boss and workers:
    - workers wait (e.g., sleeping on a condition variable)
    - boss puts work in queue and signals sleeping workers
  - workers have to decide what needs to be performed:
    ```
    forever {
      sleep until awoken by boss
      dequeue request
      switch (request)
        case X: taskX() 
        ...
    }
    ```
Pipeline
- Each thread completes a portion of a task, then passes the results to next thread
- Just like an assembly line or a processor pipeline
- Advantages:
  - simplicity (trivial synchronization)
  - low overhead for thread creation
- Disadvantages:
  - hand tuning is needed for determining the right number of pipeline stages
  - hardwired limit in the degree of parallelization
  - overall throughput is limited by slowest stage
  - more hand-tuning needed

Up-calls
- Layered applications (e.g., network protocol stacks) usually have a top-down control flow
- But often information flows upwards

  ![Diagram](Transport to Physical)

- Top-down:
  - advantage: natural ordering of locks (can always call layer below while holding lock)
  - disadvantage: up-flow cannot be handled efficiently (e.g., when incoming packet needs to be dispatched to the right transport layer protocol)

Up-calls (continued)
- Up-calls is a technique in which you structure layers so that they expect calls from below
- A thread pool exists in each layer—each thread is specialized (e.g., for a particular connection)
- Essentially an up-call pipeline is maintained per connection
- Best used with fast, synchronous control flow transfer mechanisms, but also useful as a program structuring tool
- Disadvantage: programming becomes more complicated (e.g., synchronization when calling a lower layer

Version Stamps
- (Not a program structure idea, but an overall useful technique for any kind of distributed environment.)
- Maintaining a “version number” for shared data is often a good idea
  - threads may keep local data that is not in-synch
  - data may need to be cached to improve performance
- It is a good idea to structure interfaces so that they are not data-only but data-and-version