Lecture 10

- First mid-term: Wednesday, Feb 26, one week from today: open book, open notes. No computers (except for accessing the book)
- For Monday, Feb 24, there will be a recorded lecture. It will be the Erlang Introduction and will NOT be on the exam. The link will be posted on the course calendar
- Today:
  - Chapter 6
  - Exam Review

Exploiting Parallelism

- Sources of and constraints on parallelism
  - CPUs
  - Disk
  - Network
  - Memory
- Sources of delay: disk i/o, network i/o
- Complete success: keep computation and i/o resources busy all the time
- How many threads? ~nCPUs *ready* threads

Task-based design

- Instead of thinking about threads, think about tasks to be done
  - Compute this
  - Process a web client request (server side)
  - Fetch a URL (client side)
  - Fetch a disk block
  - Write a file
  - Etc.

Thread-per-task

- Manually create a thread for each task as you become aware of it
  - Risks:
    - thread lifecycle overhead
    - Overconsumption of resources (esp. memory) – leading to poor performance
    - Danger of reaching system limits followed by system collapse

Executor Model

- Java *executors* are objects that make policy decisions about how to execute tasks:
  - In a new thread
  - In a thread pool
  - In the current task
- public interface Executor {
  void execute(Runnable task);
  }
- Producer-consumer design with no explicit queues

Kinds of Executors

1. Thread-per-task
2. Fixed thread pool – a predetermined fixed number of threads
3. Cached thread pool – no fixed maximum number of threads; threads created as needed
4. Single thread executor – later tasks guaranteed to see the results (side effects) of earlier tasks

- Nrs 1-3 are good for independent tasks
- Nr 4 is good for sequentially dependent tasks

Additional Executors

- **ScheduledThreadPool and DelayQueue**

Executors sometimes not a good idea

- If your implementation is designed as a pipeline of threads interacting through queues
- Anything where threads need to dynamically interact

Uses of Threads

- Defer work – good application of executors
- Periodic work – cron jobs
- Scheduled future work
- Exploit Parallelism – how many threads?

Deadlock avoidance

- Recall deadlock involves multiple (two or more) threads waiting for one another

- Basic deadlock avoidance technique: always acquire locks in the same order (in all threads)

What if you don't know what locks you'll need?

- Required lock order: ABCDEFGHI
- you don't always need all of the locks and occasionally you learn you need a lock after you're allowed to acquire it
- Two possibilities: fork a thread to do the work under the missing lock (but you can't wait for it to finish)
- Fork a thread to redo the computation from the beginning but advising it to acquire the missing lock at the proper time

Pipelines

- Chains of threads interacting through bounded buffers (queues)
- T1 -> Q1 -> T2 -> Q2 -> …
- Threads may be thought of a “pumps” moving data through the pipeline
- Different threads may work on different size “chunks” of data. In particular if T2’s interactions with Q2 are very expensive it may be good to take many chunks from Q1 to form the inputs to Q2

Problem

- How long does T2 wait for “enough” data before passing it along to Q2?
  - As always, it depends
  - If there is no real-time requirement then however long it takes to fill a chunk
  - Otherwise, a good use for a timeout
- Beware of timeout-driven systems
  - Lousy performance, low CPU utilization