Lecture 10 – Using Thread Pools

- Housekeeping:
  - exam return Monday
  - Programming assignment on Monday
    - Purpose: practice use of the mechanisms we've discussed
    - The next assignment will be more oriented toward system design using concurrency
    - Implement the synchronizers of Section 5.5

Quickly, task cancellation

- Chapter 7, not covering in detail
- Only a few observations on cancellation
  - Why? Nobody cares about the work a thread is doing any more – taking too long, some other thread got the answer, another thread got an error that is fatal for the app as a whole
  - How? Cooperatively! Much saner than C signal (interrupt) handling
    - Thread handles cancellation only when in known state
    - C program may have to handle a signal in any state
  - Beware: Intrinsic locks don't respond to interruption
  - Daemon thread: in Java a thread whose existence

Chapter 8 – Applying Thread Pools

- A lot of Java-specific details in this chapter
- Lecture will try to illuminate the underlying issues a bit more
    - Client dependence on execution policy
    - Thread-pool sizing
    - Queuing policy
    - Saturation policy
    - Thread-creation

Client dependence on execution policy

- Tasks that use objects that rely on thread-confinement for safety
  - Recall the single-threaded executor
- Inter-dependent tasks – ones that rely on state changes or results produced by other tasks
  - Avoiding thread-starvation deadlock
  - Timing-sensitive tasks – a less dramatic form of starvation

Thread-starvation deadlock

- Task A, currently executing in a thread, requires results from task B
- Task B cannot be executed because there are no available threads in the thread pool
- Obvious possibility in a single-thread executor but can happen with any executor that has a maximum number of threads
    - ○ Consider: is the situation any better using Thread directly?

You can have more than one thread pool

- Suggestion: in any given thread-pool have *homogeneous* tasks – similar lifetimes, similar work.
    - It is hard to choose thread-pool policies if arbitrary tasks may be thrown at the pool
        - Long-running tasks keep short-running tasks from executing
        - Priority between vastly dissimilar tasks may be hard to pin down
    - Neither thread pools nor threads are so expensive that you have to worry very much about whether you have 100 or 200
        - You may have to worry about the difference between 100 and 1000
- Remember that once a thread pool assigns a task to a

Thread-pool sizing

- Three thread states: waiting, ready-to-run, running
  - Waiting threads main effect is that they take up memory
  - Ready-to-run threads compete for the processor; lots of ready threads may make each scheduling decision more costly
  - Running threads – limited to the number of processors
  - Switching between ready and run states is

Pool sizing (2)

- The following are all system-wide (not per thread-pool) considerations
- Too many waiting threads is bad
  - How many is too many?
- Too many ready threads is bad
  - How many is too many?
  - Better question: how many is enough?
- Answer: too few running threads is bad, so ideally # of ready threads is 0 and number of running threads = number of CPUs

Pool sizing (3)

- For a resource-limited pool consider
  - Characteristics of tasks it will service
    - Task execution time
    - Task wait time
  - and how many resources should be devoted to those tasks
    - What else does the system have to do concurrently?
  - Nthreads = Ncpu * U *(1 + W/C)
    - U == desired CPU utilization for the whole pool
    - W == task wait time, C == task compute time

Pool sizing (4)

- Nthreads * per-task-disk-utilization = desired pool-disk-utilization
- Nthreads * per-task-network-utilization = desired pool-disk-utilization
- Etc.
- Pool size = minimum of all of the above – it doesn't help to make it any bigger

Pool sizing (5)

- Need to be concerned also with the size of the queue “in front of” the thread pool
  - Implicit assumption: a task can be represented by much less data than a thread
    - Number of waiting tasks >> number of threads
    - On the other hand…
  - What are queuing systems good for?
    - Smoothing out variations in arrival rate and service time
    - If work continuously arrives at a faster rate than it can be performed then queue length and wait times

Queuing Policy

- **Fundamental question: bounded or unbounded queue**
  - **Unbounded queue allows memory exhaustion and unbounded wait times**
  - **Bounded queue raises question of what to do when it is full – the *saturation policy***
    - **Abort – throw an exception**
    - **Discard – silently ignore submitted task**
    - **Discard oldest – or other policy that throws away some already queued task**
    - **Caller runs – run the task in the submitting thread**

Work rejection by the system

- In the exam problem work is internally generated by the application. Conceptually easy to slow down work submission
- What if work is being delivered by an external source, e.g. over the network – need to think about how back pressure can be exerted on the work sources
  - Classic denial of service attack – no effective way to exert such backpressure
  - Other kinds of DoS may work by breaking

Discussion

- Given the dangers of unbounded thread pools and unbounded queues why is an unbounded queue the default for fixed size thread pools and why do cachedThread pools support an unlimited number of threads?

Thread Creation

- Thread pools need to create threads from time to time
- A default thread Factory is supplied but you can customize the thread pool with your own factory (what is a factory?)
  - Statistics gathering
  - Customized unhandled exception processing
  - Customized security policies
  - …
- All this can be done without using a custom factory without using a thread pool at all but a thread pool provides a nice point of focus