Division of Labor

CSCI 5828: Foundations of Software Engineering
Lecture 07 — 02/07/2012

Goals

- Cover the material in Chapter 2 of our Concurrency textbook
    - Threading methodology
    - Conversion of sequential programs into concurrent programs
    - Task decomposition
    - Rule of thumb for the number of threads
    - Differences between compute-intensive and IO-intensive applications
    - Services of java.util.concurrent

Division of Labor

- The key concern in the design of concurrent systems is the division of labor
    - That is, how do you
        - break up the work of a sequential system
        - into well defined tasks
            - that can be executed independently
- If a sequential program's work can be split up into truly independent tasks, you are in an ideal situation for seeing significant performance improvements when those tasks are run concurrently
    - The problem is that it is often difficult to identify tasks that are "truly independent"

Task Creation Challenges (I)

- There are several challenges to task creation
  - A major challenge is **sequential dependencies between tasks**
    - Task B cannot be executed until the output of Task A is computed
      - as such, they can't be executed in parallel
        - of course, if you have hundreds of tasks As followed by hundreds of task B, you can make progress
- Another challenge is **redundant work**
  - Sometimes task A and B must perform the same calculation because it would be inefficient or too complex to ensure that all shared calculations are performed only once and then shared with all tasks

Task Creation Challenges (II)

- A third challenge is **shared data**
    - Sometimes tasks must share data
    - As we saw in previous lectures,
        - if both tasks update that data
            - then the developer must add synchronization to ensure that data is updated in a consistent fashion
                - thereby increase the complexity of the system
- Our book calls this shared mutability; we will discuss ways to both deal with it and to get rid of it later this semester

Task Creation Challenges (III)

- A fourth challenge is **finding substantial tasks**
    - The amount of computation in each task (its **granularity**) must be large enough to offset the overhead needed to manage tasks and threads
        - Goal
            - More tasks than threads; each task of high granularity
- Otherwise
    - You will not be utilizing your threads (cores) to their fullest extent possible AND the work devoted to task management may dominate the program
        - can lead to your concurrent program being slower than its single-threaded counterpart!

Task Creation Challenges (IV)

- A fifth challenge is whether tasks are created statically or dynamically
    - Sometimes all of the tasks that will be run during a program will be known ahead of time and the mapping of tasks to threads will be clear
        - The creation of such tasks and their assignment to threads can be handled statically (in the source code); the same thread will perform the same type of work each time the program is run
- In situations where tasks cannot be predicted ahead of time or there are simply way more tasks than threads, then a dynamic allocation strategy needs to be employed
    - for instance, you might put all the tasks in a queue and have threads pull tasks off the queue when they are ready for more work

Threading Methodology by Clay Breshears

- To design and implement a concurrent system
  - First, produce a tested single-threaded version of the program
    - Via standard reqs/design/implement/test/tune/maintenance life cycle
  - Then
    - Analyze program to find computations that are independent of each other AND take up a large amount of the serial execution time
      - if the single-threaded program spends 80% or more of its time in a particular section of code, look there for independent tasks
    - Design and Implement the concurrent system (typically straightforward)
    - Test for Correctness: Verify that the concurrent code produces correct output
    - Tune for performance: once the code is correct, find ways to speed up

Performance Tuning

- Tuning threaded code typically involves
  - identifying sources of contention on locks (synchronization)
  - identifying work imbalances across threads
  - reducing overhead
    - overhead refers to all of the code you put into the concurrent system that
      - creates threads
      - creates tasks
      - assigns tasks to threads
      - gathers results from completed tasks

From sequential to concurrent

- Our book looks at two types of applications you might encounter when trying to move from a single-threaded program to a concurrent program
    - “Programs don’t divide the same way and benefit from the same number of threads.”
- Some programs are computation intensive
    - They have a lot of calculations to perform; IO is secondary
- Some programs are IO intensive
    - They perform lots of reads/writes to databases, files, or sockets; computation is secondary

Examples

- To discuss the difference between the two, the book presents two different example programs
  - The first makes web service calls to calculate the value of a stock portfolio
  - The second computes the total number of prime numbers within a given range
- These two programs are used to demonstrate
  - how many threads each type of program needs
  - how to divide up the tasks
  - how much speed-up we can expect to see in a concurrent implementation

Determining the Number of Threads (I)

- **The lower bound for the number of threads** in either type of system (compute vs. IO) **is equal to the number of cores** on the machine
  - It doesn't make sense to have the number of threads be less than the number of cores
    - if they are available, the number of cores represent the maximum amount of parallelism you can see in your system
- **For compute-intensive systems**, the **number of cores** also represents the **upper bound for the number of threads** in your system
  - if you had 4 cores and created 8 threads in a compute-intensive app, then the scheduler is going to be wasting time switching between the 8 threads, making sure each of them is making progress, but, unfortunately, suspending a thread when it still had work to do

Determining the Number of Threads (II)

- For IO-intensive systems, the number of threads should be higher than the number of cores
  - The reason: a CPU is much faster at computation than it is at I/O
    - How much faster? Source: http://norvig.com/21-days.html

L1 cache reference	0.5 ns
Execute typical instruction	1 ns
Branch mis-prediction	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes w/ cheap algorithm	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Determining the Number of Threads (III)

- From the table
  - reading from main memory is two orders of magnitude slower than reading from cache
  - reading from disk is five orders of magnitude slower than reading from main memory!
- So, when a thread starts an I/O operation, its associated core has plenty of time to do other work while the thread waits for a response
  - Thus, an IO-intensive app needs lots of threads, so the core has plenty of active threads to choose from while blocked threads wait

Determining the Number of Threads (IV)

- How do we decide how many threads to spawn
  - By trying to determine the amount of time one of its tasks will be blocked (**the blocking coefficient**) waiting on IO operations
    - If you profile a task and learn that it takes 2 seconds to execute and 1 second is spent waiting for a response, then this task's blocking coefficient is 50% or .5
- Once you have determined the blocking coefficient, the equation for calculating the number of threads is
  - Number of Threads = Number of Cores / (1 - Blocking Coefficient)
- For a 16 core machine, if our tasks had a blocking coefficient of .5
  - Number of Threads = 16 / (1 - .5) = 16 / .5 = 16 * 2 = 32 threads

Determining the Number of Threads (V)

- Number of Threads = Number of Cores / (1 - Blocking Coefficient)
  - For non-blocking tasks, you get “num threads == num cores”
  - For tasks that spend most of their time blocked, you’ll get a very high number of threads
- We don’t have to worry about a division by zero
  - a blocking coefficient of 1 would means that a task will never complete
  - we shouldn’t waste our time with such tasks!

Determining the Tasks (I)

- Now, that we know how to calculate the number of threads our concurrent application needs, we need to think about what tasks we will create
- Your first step should be to decide what the task will be
- Your second step should be to determine if the effort associated with each task is constant or variable
  - If task effort is constant
    - that is, each task takes roughly the same amount of time
    - then split the tasks into a number of groups equal to the number of threads
  - If task effort is variable
    - that is, some tasks will take longer to complete than others
    - then additional planning is required (stay tuned)
- Your ultimate goal is an equal distribution of work across all threads to ensure your cores are maximized to their fullest capacity

Determining the Tasks (II)

- Step 1: Determine the Task
  - For our portfolio calculator, the task is
    - Take a stock symbol and the number of shares as input
    - Retrieve the current price of the stock
    - Multiply the price by the number of shares and return the result
- For our primes finder, the task is
  - Take a range of numbers as input
  - For each number in the range, calculate whether its prime
  - Keep track of each prime and return total when the iteration is complete

Determining the Tasks (III)

- Step 2: Determine the effort
  - For our portfolio calculator, the task effort is constant
    - The effort will be dominated by network latency (retrieving the stock price) and will for any given network be roughly the same for each task
    - The time to perform the multiplication once the stock price is retrieved is trivial
- For our primes finder, the task is variable
  - The time to calculate whether a number is prime is variable
    - in general, it takes (a lot) longer for larger numbers

Determining the Tasks (IV)

- Step 3: Allocate tasks to threads
  - For our portfolio calculator, allocation is trivial
    - n = number of stocks / number of threads
    - Allocate n stocks to each thread and run all threads at once
  - For our primes finder, the allocation is not so clear cut
    - Let's say we are asked to find all prime numbers from 1 to 10,000,000
    - If we had 8 cores, we would create 8 threads for our app
    - If we then split the range 1 to 10,000,000 into eight groups and assigned them to tasks, we would get an unequal distribution of effort

Determining the Tasks (V)

- Consider
  - Thread 0 calculating from 1 to 1,250,000 will finish fast
  - Thread 1 calculating from 1,250,001 to 2,500,000 will finish next
  - Thread 7 calculating from 8,750,000 to 10,000,000 will be the last to finish
- While Thread 7 is calculating the last portion of its range
  - all the other threads may be done, sitting idle, while thread 7 labors away!
- Unfortunately, the solution to this problem is not easy
  - Dividing 10,000,000 into 8 equal parts is straightforward when the parts are contiguous
  - if you wanted to somehow create 8 partitions which had a balance of short and long calculations, you've got quite a bit of work to do!

Determining the Tasks (VI)

- The ideal solution would be for Threads 0 - 6 to “steal work” from Thread 7 once they are finished with their work
    - In this way, all threads will be kept busy until the entire range has been checked
- The problem is that this type of work stealing is hard to program
    - and, remember, a single threaded solution to this problem doesn’t have to spend all this time creating threads, creating tasks, allocating tasks to threads, and reallocating tasks to idle threads
    - all of this work is **overhead** and slows us down
    - Too much overhead and the single-threaded program may actually be faster than its concurrent counterpart!

Determining the Tasks (VII)

- What does our book recommend?
  - To keep things simple, and to avoid unnecessarily complex overhead,
    - just create a lot of parts (way more than the number of threads)
    - and dole them out to the available threads in a straightforward manner
      - So, in our prime finder example,
        - If we have 8 threads, don't create 8 partitions
        - create (say) 64 partitions and give each thread 8 partitions in a round-robin fashion
- The book also later shows how Java can help with implementing a "work stealing" strategy like we discussed on the previous slide

Example: Portfolio Calculator

- **Demo**
- Overall program structure
    - Abstract specification of algorithm in superclass
    - Concrete implementation in subclass (Abstract Factory design pattern)
        - Retrieval of stock price is hidden in a helper class
        - Use of **final** keyword everywhere
            - Helps in two ways: compiler can make optimizations knowing that certain values are read-only and final variables can be referenced by anonymous classes that are created within their scope
- Amazing difference in speed: 19 seconds versus 0.97 seconds!

Example: Portfolio Calculator

- Concurrency techniques
  - Did NOT make use of Thread, wait(), notify(), synchronized, etc.
  - Instead made use of features in the “new” java.util.concurrent package
    - ExecutorService in place of Thread
      - Callable<Type> and Future<Type>
- Let’s look at these in a bit more detail

ExecutorService (I)

- ExecutorService is a Java interface that defines a common set of services for an abstract “thread pool”;
    - this interface has a variety of concrete implementations that provide a choice of concurrent behavior to developers
- What’s a thread pool?
    - Thread creation is a slow process
    - Thread pools create a bunch of threads all at once (typically as a system is booting up)
    - When a new thread is needed, one is taken from the pool and it starts executing immediately
        - very helpful in situations where, e.g., a server is responding to incoming network requests

ExecutorService (II)

- Static factory methods on the Executors class are used to create instances of the ExecutorService; for instance
  - **CachedThreadPool**: creates threads as needed but will reuse previous ones if they are available
  - **FixedThreadPool**: creates a fixed set of threads
  - **ScheduledThreadPool**: creates a thread pool that can execute tasks after a delay or periodically
  - **SingleThreadExecutor**: creates a thread pool with only a single thread
- You can write code that only depends on the interface ExecutorService and then be free to change the actual threading behavior you get at run-time based on external factors (you can even switch threading behaviors on the fly)

ExecutorService (III)

- The basic API of the ExecutorService allows you to
    - submit a single task for execution
    - submit a collection of tasks for execution
        - where you want all of the tasks results (invokeAll)
        - or where you want just one of the results (invokeAny)
- shutdown the thread pool when you are done with it

Callable/Future: Making this all work

- In order to give tasks to the thread pool and receive results back, you make use of two additional interfaces
  - Callable<T> and Future<T>
- Both make use of Java generics to give flexibility in the return types of the computation
- For instance, I can promise that my task returns a string
  - Callable<String> callMe = new Callable<String>() {
    - public String call() throws Exception { …; return result; }
  - }
- callMe is now a Task that I can hand to an ExecutorService

Callable/Future (II)

- When I give callMe to an ExecutorService, it is going to hand the task to a thread and ask the thread to execute it
    - At the time, we have no idea how long it will take for the task to complete
        - Thus, the ExecutorService gives me an instance of Future<String> back so I can get the value once the task is complete
            - Future<String> myString = service.submit(callMe);
- This call does NOT block, I get a reference to myString almost immediately
- I can then decide to retrieve the string whenever I need it by calling get()
    - String result = myString.get();
- This call MAY block, if the task is still being executed; otherwise, I get the result right away. I can also call a version of get() that accepts a timeout.

Bringing it all together

- You should now understand the ConcurrentNAV application
    - It creates an executor service with a fixed number of threads
        - using the equation on slide 16 and assuming a blocking coefficient of .9
    - It creates a collection of tasks of type Callable<Double>.
        - The code inside call() retrieves the stock price from Yahoo and computes the value of the holding
- All of these tasks are handed to the executor service and run in parallel
- It loops over the collection of Future<Double> instances that it received from the executor service and calls get() and tallies up the total value of the portfolio

Design Note: Isolated Mutability

- In this design, we have only one variable that is mutable
  - the variable that totals up all the individual holdings
  - however, we know that only one thread has access to this variable
  - all other computations happen in other threads and the results of those computations are handed back to the main thread via Future<T>.get()
- It is possible that when we loop over the collection of Future<Double> instances returned by the service that we block on a call to get() while other tasks later in the list are already completed
  - but in this particular application, that is okay behavior
  - if we want to process the tasks in order of completion, we can make use of another interface called the CompletionService. We'll see that later

Example: Prime Finder

- **Demo**
- Structure of concurrent app is nearly the same
    - Create fixed-size thread pool
    - Calculate partitions and store them as Callables
    - Invoke all the tasks at once
    - Loop over the Futures incrementing the total count
    - return the result
- This program allows us to vary the pool size and the number of tasks

Example: Prime Finder

- As predicted, we don't see a huge increase in speed over the sequential version of the program because the cores are being underutilized (see activity monitor)
    - eventually threads 0 to n-2 complete early while thread n-1 takes a lot longer because it was handed the largest numbers to compute
- Rather than deal with this problem by creating more complex code to create an equal distribution of work, the author simply creates more tasks (in this case, divides the total range into more partitions)
    - With 2 threads on a dual-core machine and 100 partitions, he saw nearly a two times speed-up (1.85) over the sequential version
- After testing on an 8-core machine, he saw the greatest speed-up when
    - number of threads == number of cores; parts == 32 * number of cores
- After that diminishing returns set-in, with no real increase of speed due to overhead

Summary

- Designing and developing concurrent software systems can be done in a methodical fashion
- Rule of thumb can be used to identify the number of threads to be used
    - and compensate for the differences between compute-intensive and IO-intensive systems
- Task decomposition requires both identifying the independent computations performed by a concurrent system as well as how tasks will be assigned to the available threads
- java.util.concurrent provides a powerful set of abstractions for designing and developing concurrent systems
    - ExecutorService provides an abstract interface for thread pools with a wide variety of run-time behaviors

Coming Up Next

- Lecture 8: Design Approaches for Concurrent Systems
- Lecture 9: Behavior-Driven Development and Cucumber