Networks and Database Systems

Professor Tom Ellman
Lecture 3
Problem with TCPEchoServer

- Server thread blocks when waiting for a connection.
- The server thread cannot do any other work while it is blocked.
- While communicating with a client, the server is not processing new connections.
- Only one client can be connected at a time.

Solutions:
- Multi-threaded server.
- Non-blocking IO.
Multi-Threading

• A Java program has several parts that need to run concurrently.
• Each part is implemented as a thread.
• Only one thread can use the CPU at a time.
• Threads take turns using the CPU.
• Java virtual machine allocates CPU time-slices to threads.
Threads

• A *thread* is a single sequential flow of control with a program.

• Each thread has its own program counter and run time stack.

• All threads share access to objects in heap storage.
Round-Robin Scheduling

• Run program 1 for one time slice.
• Run program 2 for one time slice.
• Run program 3 for one time slice.
• Resume program 1 for another time slice.
• Resume program 2 for another time slice
• Resume program 3 for another time slice.
• Etc.
SimpleThreadsDemos A & B

- Demonstrate two different ways of constructing and running threads.
- Notice that the ordering of output by the two threads will vary as the program is run repeatedly.
- The interleaving of outputs is unpredictable, unless we explicitly set different priorities for the threads.
- Makes debugging a challenge.
Defining and Running Threads
SimpleThreadsDemoA

• Defining a thread:
  – Define a class `SimpleThread` that extends `Thread`.
  – Provide `SimpleThread` with a `run` method that overrides the `run` method of `Thread`.

• Running a thread.
  – Use the `new` operator to create an instance of `SimpleThread`.
  – Invoke the `start()` method on the newly created `SimpleThread` object.
Defining and Running Threads

SimpleThreadsDemoB

• Defining a thread:
  – Define a class SimpleThread that implements Runnable.
  – Provide SimpleThread with a run method.

• Running a thread.
  – Use the new operator to create an instance of SimpleThread.
  – Use the new operator to create an instance of Thread, passing the SimpleThread instance as its target parameter.
  – Invoke the start() method on the newly created Thread object.
Thread Methods

- **void start()**: Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

- **void run()**: If this thread was constructed using a separate `Runnable` run object, then that object's run method is called; otherwise, this method does nothing and returns.

- **void sleep(long ms)**: Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
Interaction among Threads

• Thread $T_1$ may call a method of thread $T_2$.
• The method may modify an instance variable of $T_2$.
• Thread $T_2$ may notice that the value of the variable has changed.
• Thread $T_2$ may then respond accordingly.
Animation Controller Example

- The main program’s run thread creates and starts a thread to animate an expanding ball.
- The main-run thread continues running concurrently with the animation thread.
- The main-run thread waits for mouse clicks and process them by setting instance variables in the animation thread.
- The animation thread notices the changed variables and responds by starting, stopping or quitting the animation.
Compare

• Object on which a method is invoked:
  – The `processClick` method is invoked on the `expandingBall` object, which is an instance of the `ExpandingBall` class.

• Thread in which a method runs:
  – The `processClick` method is executed in the main-run thread, which is not the same thread as the `expandingBall` thread that is running the animation.
Interaction among Threads

• Thread $T_1$ may call a method to *set* the value of an instance variable of an object $S$.

• Thread $T_2$ may call a method to *get* the value of the instance variable of the object $S$. 
MultiEchoServer

- The `main` method of the `MultiEchoServer` class runs in the main thread.
- It repeatedly does the following:
  - Accept a connection from a client.
  - Constructs and runs a `ClientHandler` thread to process the connection.
- The server remains (almost) continuously available to accept connections.
Problems with MultiEchoServer

- Java takes a nontrivial amount of time to set up and run a new thread.
- Problem for servers accepting many new connections per second.
- Java uses a nontrivial amount of space to manage each thread, approximately 1MB.
- Problem for servers processing many connections at once.
Solution

- Limit the number of threads to be created, and therefore the number of connections that can be processed simultaneously.

- When a connection is terminated, reuse the thread that was handling that connection.
Producer-Consumer Example

• Producer repeatedly puts data objects in a queue.
  – E.g., Thread downloading an image off the internet puts successive image chunks in a queue.

• Consumer repeatedly gets data objects from a queue.
  – E.g., Thread displaying the image gets successive image chunks out of the queue.
Producer-Consumer Example

Producer Thread → Buffer → Consumer Thread

Enqueue Data → Dequeue Data
Need for Synchronization

• What if producer and consumer run at different speeds?
  – Fast producer might try to put data into a full queue.
  – Fast consumer might try to take data out of an empty queue.

• What if producer and consumer try to access the queue at the same time?
  – Producer or consumer might find the queue in an inconsistent state.
Critical Sections

• A method \texttt{m() \text{of class C}} may be declared to be "\textit{synchronized}".

• When a thread \texttt{T_1} calls method \texttt{m()} on an object \texttt{s} of class \texttt{C}, the thread \texttt{T_1} \textit{acquires a lock} on object \texttt{s}.

• If another thread \texttt{T_2} tries to invoke a synchronized method on object \texttt{s}, thread \texttt{T_2} will be \textit{blocked}.

• When thread \texttt{T_1} returns from method \texttt{m()} on object \texttt{s}, thread \texttt{T_1} \textit{relinquishes the lock}.

• Thread \texttt{T_2} will become \textit{unblocked} and will proceed to run the synchronized method on object \texttt{s}. 
Critical Sections Example

1. Thread $T_1$ calls $s.m()$, and locks object $s$.
2. Thread $T_1$ begins running method $s.m()$.
3. Thread $T_2$ calls $s.m()$, and gets blocked.
4. Thread $T_1$ continues running method $s.m()$.
5. Thread $T_1$ returns from method $s.m()$, and unlocks $s$.
6. Thread $T_2$ gets unblocked and proceeds to run $s.m()$. 
Synchronization Methods of the Object Class

- `void wait()`: Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method from a synchronized method of this object.
- `void notify()`: Wakes up a single thread that is waiting on this object.
- `void notifyAll()`: Wakes up all threads that are waiting on this object.
MultiEchoServerThreadPool

- Main method creates/starts one `ConnectionAccepter` thread.
- Also sets up a synchronized `ThreadPool` (queue) initialized to hold a limited number of `ClientHandler` threads.
- `ConnectionAccepter` repeatedly accepts connection, gets `ClientHandler` thread from the `ThreadPool` and gives it the connection socket.
- `ConnectionAccepter` will block if `ThreadPool` is empty.
- `ClientHandler` thread communicates with the client using the socket it received from the `ConnectionAccepter`.
- When the client closes the connection, the `ClientHandler` thread returns itself to the `ThreadPool`. 
**MultiEchoServerTaskPool**

- Main method createsstartsWith one `ConnectionAccepter` thread.
- Also sets up a synchronized `TaskPool` (queue).
- Also createsstartswiths a limited number of `ClientHandler` threads.
- `ConnectionAccepter` (producer) thread puts tasks (sockets) into the `TaskPool`.
- `ClientHandler` threads remove tasks (sockets) from `TaskPool` and process them.
- `ClientHandler` threads block when `Task` pool is empty.
Connection Accepter

Tasks (Sockets)

Task Pool

Client Handler (Waiting)

Client Handler (Waiting)

Client Handler (Busy)

Network

Connection Requests
Connection Accepter

Tasks (Sockets)

Task Pool

Client Handler (Waiting)

Network

Client Handler (Busy)

Client Handler (Busy)
Limiting the Task Pool

• MultiEchoServerTaskPool uses an ArrayList (of unbounded size) to implement the task pool.
• In practice, we might want to limit the size of the pool of tasks waiting to be serviced.
• E.g., Use QueueArrayLimited from the Producer/Consumer example.
• When the task pool is full, and a new connection request arrives, the ConnectionAccepter will block.
Choosing the Number of Threads

• Goal is to match server processor speed with network data delivery speed.

• Gradually increase the number of threads until enough data comes from the network to keep the server CPU busy.
Reusing an Unbounded Number of Threads

• Keep track of the number of waiting/busy client handler threads.

• If all threads are busy when a connection request arrives, create a new one to service the request.

• Otherwise, put the task (socket) in the task pool, where it will be serviced by one of the waiting client handler threads.

• Periodically discard excess client handler threads. When? How?