CMSC 433 – Programming Language Technologies and Paradigms

Task Execution

Task

- Multithreaded programs often organized as tasks
  - i.e., abstract, discrete units of work
- Ideally, each task is independent
- Running program organizes & executes these tasks

Sequential Task Execution

```
public class SingleThreadWebServer {
    public static void main(String[] args) throws IOException
    {
    ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            // executes in this Thread
            handleRequest(connection);
        }
    }
    …
}
```

Sequential Task Exec. (cont.)

- Doesn’t exploit concurrency
- Handles requests one at a time
  - Current request blocks all others

Thread Per Connection

```
public class ThreadPerTaskWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            new Thread(task).start();
        }
    }
}
```

Thread Per Connection

- Handles requests concurrently
  - Requests don’t block each other
  - Exploits concurrency
- Risks
  - handleRequest() must be thread-safe
  - Can create an unbounded number of threads

Too Many Threads

- Thread management incurs costs
  - Time
  - Resource consumption
- Beyond a certain point, additional threads can be detrimental
  - Limit the number of threads created

The Executor Framework

- Tasks are logical units of work
- Threads allow tasks to run asynchronously
- But we sometimes need to manage how and when threads are created
- The Executor Framework serves this purpose

Executor

```
public interface Executor {
  void execute(Runnable command);
}
```

- Follows Producer/Consumer pattern
  - Producer submits a Runnable
  - A thread in a thread pool (consumer) executes the Runnable
- Decouples task submission from task execution

Executor (cont.)

- Executor implementation can create different execution policies
  - How many tasks can execute concurrently?
  - In what order will they run?
  - Can tasks be queued if they can't run immediately?
  - What happens when executor capacity is exceeded?

Executor (cont.)

- Two code styles supported
  - Actions: Runnables
  - Functions: Callables
  - Also has lifecycle mgmt: e.g., cancellation, shutdown

ExecutorService

- ExecutorService adds lifecycle mgmt to Executor

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination( long timeout, TimeUnit unit);
    // other convenience methods for submitting tasks
  }
```

Creating Executors

- Usually created via **Executors** factory class
  - Configures a **ThreadPoolExecutor**
  - Customizes shutdown methods, before/after hooks, saturation policies, queuing

Creating Executors

- Executors methods

```
public class Executors {
  static ExecutorService newSingleThreadedExecutor();
  static ExecutorService newFixedThreadPool(int n);
  static ExecutorService newCachedThreadPool(int n);
  static ScheduledExecutorService newScheduledThreadPool(int n);
  // additional versions & utility methods
  …
}
```

Simple WebServer

```
class WebServer {
  ExecutorService pool =
         Executors.newFixedThreadPool(7);
  public static void main(String[] args) {
    ServerSocket socket = new ServerSocket(80);
     while (true) {
        final Socket connection = socket.accept();
         Runnable r = new Runnable() {
            public void run() {handleRequest(connection);}
         };
        pool.execute(r);
    }
  }
}
```

Extended WebServer

```
public class LifecycleWebServer {
    private final ExecutorService exec = Executors.newCachedThreadPool();
    public void start() throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown()) {
            try {
                final Socket conn = socket.accept();
                exec.execute(new Runnable() {
                    public void run() {  handleReq(conn);  }
                });
            } catch (RejectedExecutionException e) { }
        }
    }

…
```

Extended WebServer (cont.)

```
…
  public void stop() { exec.shutdown(); }

  void handleReq(Socket connection) {
     Request req = readRequest(connection);
     if (isShutdownRequest(req))
        stop();
     else
      dispatchRequest(req);
  }
…
}
```

ScheduledExecutorService

- For deferred and recurring tasks, can schedule
  - **Callable or Runnable to run once** with a fixed delay after submission
  - Schedule a **Runnable to run periodically at a fixed** rate
  - Schedule a **Runnable to run periodically with a** fixed delay between executions
- Submission returns a **ScheduledFutureTask** handle which can be used to cancel the task
- Like **Timer**, but supports pooling and is more robust

ScheduledExecutorService

```
import static java.util.concurrent.TimeUnit.*;
 class BeeperControl {
    private final ScheduledExecutorService scheduler =
     executors.newScheduledThreadPool(1);

    public void beepForAMinute() {
        final Runnable beeper = new Runnable() {
                public void run() { System.out.println("beep"); }
            };
        final ScheduledFuture<?> beeperHandle =
            scheduler.scheduleAtFixedRate(beeper, 0, 1, SECONDS);
        scheduler.schedule(new Runnable() {
                public void run() { beeperHandle.cancel(true); }
            }, 60, SECONDS);
    }
 }
```

Exploiting Parallelism

```
public class SingleThreadRenderer {
    void renderPage(CharSequence source) {
        renderText(source);
        List<ImageData> imageData = new ArrayList<ImageData>();
        for (ImageInfo imageInfo : scanForImageInfo(source))
            imageData.add(imageInfo.downloadImage());
        for (ImageData data : imageData)
            renderImage(data);
    }
```

Exploiting Parallelism

```
public abstract class FutureRenderer {
    private final ExecutorService executor = Executors.newCachedThreadPool();
    void renderPage(CharSequence source) {
        final List<ImageInfo> imageInfo = scanForImageInfo(source);
        Callable<List<ImageData>> task =
                new Callable<List<ImageData>>() {
                    public List<ImageData> call() {
                        List<ImageData> result = new ArrayList<ImageData>();
                        for (ImageInfo imageInfo : imageInfos)
                            result.add(imageInfo.downloadImage());
                        return result;
                    }
                };
…
```

Exploiting Parallelism

```
…
Future<List<ImageData>> future = executor.submit(task);
renderText(source);
try {
  List<ImageData> imageData = future.get();
  for (ImageData data : imageData)
    renderImage(data);
  } catch (InterruptedException e) {
        // cleanup here
 }
…
}
```

Performance

- Performed two tasks concurrently
  - (1) Rendering text & (2) downloading images
- Speedup regulated by workload
  - If downloading 10x longer than rendering, max speedup is only 9%
  - If downloading takes same time, max speedup is only 2
- Best improvements when there are lots of independent, homogeneous tasks to be done

Kinds of Parallelism

- Data parallelism
  - The same task run on different data in parallel
- Task parallelism
  - Different tasks running on the same data
- Pipeline parallelism
  - A parallel pipeline of tasks, each of which might be data parallel
- Unstructured
  - Ad hoc combination of threads with no obvious top-level structure

Data Parallelism

- Example: convert all characters in an array to upper-case
  - Divide data between different tasks & perform the tasks in parallel
  - Key constraint: no dependencies between the tasks that require their results to be ordered

Task Parallelism

- Example: compute several functions on the same data (e.g., average, minimum, mean)
  - Divide work into multiple different tasks & perform the tasks in parallel
  - Key constraint: No dependencies between the tasks, so all tasks can run in parallel

Pipeline Parallelism

- Output of one task is the input to the next
  - Each task can run in parallel
  - Throughput gated by the longest-latency element in the pipeline

Pipeline Load Balancing

- Assign more than one computational process to each task
  - Combines data- and pipeline- parallelism

Design Recipe

- Here are some rules of thumb to follow when designing a concurrent system

Concurrency Strategy

- Determine wow to break down the entire task into smaller units of work
  - e.g., Data parallel, Task parallel, Pipeline
- Will need to anticipate some of the coding you'll do next

Implement Code Units

- Write down the parameters of that work
  - What varies for the chunk done by thread 1 vs. thread 2 vs. thread 3 etc.
- Write down functions that do the work, given the parameters
- Test those functions in a single-threaded scenario
  - e.g., make a loop that calls your functions N times to compute the total result.

Identify Concurrency Hazards

- Look for concurrent accesses to shared data structures
- Decide how to handle these accesses
  - e.g., use synchronization, make copies, use thread confinement, etc.

Choose Task Execution Policy

- Choose how you will execute your tasks
  - E.g., thread pool
- Choose the number of threads
  - May need several queues and executors if there are dependencies between the tasks.