

CMSC 433 – Programming Language Technologies and Paradigms

Applying Thread Pools

Tasks and Execution Policies

- Executor tries to decouple task submission from execution policy
- But some tasks are incompatible with some execution policies

Dependent Tasks

- If Task B can't run until Task A completes
 - Shouldn't use `Executors.newCachedThreadPool()`, which creates threads on-demand

Thread-Confined Tasks

- If task expects to run in a thread-confined manner
- Probably need to use
`Executors.newSingleThreadExecutor()`

Time-Sensitive Tasks

- If task needs to be very responsive
 - e.g., UI update code
- Need to make sure no individual tasks gets starved

Tasks Using ThreadLocals

- ThreadLocal variables are associated with a thread
- Executors can reuse threads across tasks
 - Can also kill & create threads

Thread Starvation Deadlock

- When all executing threads are blocked waiting for tasks on the work queue

Thread Starvation Deadlock

```
public class ThreadDeadlock {  
    ExecutorService exec = Executors.newSingleThreadExecutor();  
  
    public class RenderPageTask implements Callable<String> {  
        public String call() throws Exception {  
            Future<String> header, footer;  
            header = exec.submit(new LoadFileTask("header.html"));  
            footer = exec.submit(new LoadFileTask("footer.html"));  
            String page = renderBody();  
            // Will deadlock -- task waiting for result of subtask  
            return header.get() + page + footer.get();  
        }  
    }  
}
```


Thread Pool Sizing

- Can avoid the starvation problem by sizing thread pool correctly
- Additionally, pool size should be larger than the average number of long-running tasks
- Otherwise, likely that all threads will eventually be running long-running tasks

Sizing Thread Pools

- For CPU-bound applications on N processor machine
 - N+1 threads often a good choice
 - Use `Runtime.getRuntime().availableProcessors()` to determine number of CPUs dynamically

Sizing Thread Pools

- For I/O-intensive or blocking tasks, compute or estimate the following quantities
 - N_{cpu} = Number of CPUs
 - W/C = Waiting time / computing time
- To keep the CPUs 100% busy, you'd need
 - $N_{threads} = N_{cpu} * (1 + W/C)$
- Note: Usually don't want the CPUs to always be at 100% utilization

Configuring ThreadPoolExecutor

- ThreadPoolExecutor is the base implementation for the Executors returned by Executors factory methods

```
public ThreadPoolExecutor(  
    int corePoolSize,  
    int maximumPoolSize,  
    long keepAliveTime,  
    TimeUnit unit,  
    BlockingQueue<Runnable> workQueue,  
    ThreadFactory threadFactory,  
    RejectedExecutionHandler handler)
```

Creation & Teardown

- `corePoolSize` – target number of threads
- `maximumPoolSize` – max number of active threads
- `keepAliveTime` – threads idle for this long may be terminated

Queue Management

- Can supply a BlockingQueue to hold pending tasks that can't be run immediately
- Three choices
 - Unbounded – e.g., unbounded LinkedBlockingQueue
 - Bounded – e.g., ArrayBlockingQueue
 - Saturation policy - What happens when the queue fills up?
 - Synchronous handoff – e.g., SynchronousQueue
 - If all threads are unavailable, rejects the task according to saturation policy

Queue Management

- Can use Queue to order when tasks start
 - `LinkedBlockingQueue` - FIFO
 - `PriorityBlockingQueue` – ordered by `Comparable` or `Comparator`

Saturation Policy

- What happens when you submit a task to a full queue or to a shutdown executor?
- Specified by handler parameter
- Some impl's defined in ThreadPoolExecutor:
 - `AbortPolicy` – throw `RejectExecutionException`
 - `DiscardPolicy` – silently discard task
 - `DiscardOldestPolicy` – discard task that would be run next & resubmit the new task
 - `CallerRunsPolicy` – execute task in calling thread

BoundedExecutor

```
public class BoundedExecutor {
    private final Executor exec;    private final Semaphore semaphore;
    public BoundedExecutor(Executor exec, int bound) {
        this.exec = exec;    this.semaphore = new Semaphore(bound);
    }
    public void submitTask(final Runnable command) throws InterruptedException {
        semaphore.acquire();
        try {
            exec.execute(new Runnable() { public void run() {
                try {
                    command.run();
                } finally { semaphore.release(); }
            }});
        } catch (RejectedExecutionException e) { semaphore.release(); }
    }
}
```

Thread Factories

- Class that creates & returns new Threads for the executor's thread pool

```
public interface ThreadFactory {  
    Thread newThread(Runnable r);  
}
```

- Can implement this interface to customize
 - Exception handling
 - Functionality
 - Data fields
 - etc.

Extensions to ThreadPoolExecutor

- Override ThreadPoolExecutor to customize code
- beforeExecute(Thread t, Runnable r)
 - Method invoked prior to executing the given Runnable in the given thread
- afterExecute(Runnable r, Throwable t)
 - Method invoked upon completion of execution of the given Runnable
- terminated()
 - Method invoked when the Executor has terminated