Concurrency in Object Oriented Programs 4

Object-Oriented Software Development COMP4001
CSE UNSW Sydney
Lecturer: John Potter
Outline

• Thread Control
• Tasks and Task Control
• Finding and Exploiting Parallelism
Thread States

• A thread can be in one of the following states:
  – **NEW** A thread that has not yet started is in this state.
  – **RUNNABLE** A thread executing in the Java virtual machine is in this state.
  – **BLOCKED** A thread that is blocked waiting for a monitor lock is in this state.
  – **WAITING** A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
  – **TIMED_WAITING** A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
  – **TERMINATED** A thread that has exited is in this state.
Thread Control

- Manage the schedules of multiple threads
- Hand information from one thread to another
- Signal one thread that another thread has finished doing something

Methods for inter-thread control and synchronisation are:
- `getPriority`, `setPriority`, `join`, `interrupt`
- `yield`, `sleep` (for current thread)
- `wait`, `notify`, `notifyAll` (per object)
Thread Priorities

• Runnable threads are scheduled for execution by JVM
  – Priority mechanism is platform dependent
  – Depends on VM (and OS)
• Programmers can get and set priority for individual objects
  – getPriority()
  – setPriority()
• The scheduler uses priorities to allocate CPU time
• ADVICE: Avoid use of priorities
  – May cause liveness problems e.g. via priority inversion
  – Introduces platform dependence
  – Too weak to protect critical sections!
  – Usually sufficient to rely on default priority
Thread.

sleep

- Put target thread to sleep for a specified time
  - VM guarantees a minimum sleep time
  - **Does not give up locks**
    - Any held continue to be held while sleeping
    - Should usually release locks before sleep

- sleep is useful for
  - providing timeouts
  - controlling the speed of applications
    - e.g. specifying framerates for multimedia presentations

- sleep responds to an interrupt on the thread
  - by exiting, clearing interrupt status and throwing an InterruptedException
public class Sleep {
    public static void main(String args[]) throws InterruptedException {
        final int N = 10;
        for (int i = N; i > 0; --i) {
            // Pause for 10, 9, ... 1 seconds
            Thread.sleep(1000 * i);
            // Awake indicator
            System.out.print(".");
        }
    }
}

Thread.yield()

• Causes current thread to pause
  – Gives up the CPU

• Allows other threads to execute
  – But no guarantee about which order others may run
  – Current thread may just start up again

• Similar to Thread.sleep(0)
  – Except for throwing of InterruptedExceptions
object.wait()

• A method of the Object class
• Cause current thread to wait
  – Current thread must hold the target object’s lock when called
  – release the lock and place thread on wait queue for the lock
• Resumption occurs if one of following occurs
  – Waiting thread is interrupted
    • resumes, clears interrupt status, and throws InterruptedException
  – Another thread invokes the notify()/notifyAll() methods for the target object
object.notify() or notifyAll()

• Which of these should be called?
• Unless there is only one condition for an object that may be waited on
  – Call notifyAll()
  – and re-check the condition for that thread
    • while(!condition) wait();
      – with try catch InterruptedException handling
• Only use notify() when you are sure that any thread that might be waiting can make use of the condition causing the notification
thread.interrupt()

• Send an indication to a thread that it should stop and do something else
  – Useful for external signals
  – Programmers determine exactly how a thread responds to an interrupt

• Interrupt handling:
  – If thread is currently blocked
    • by wait, sleep or join
    • interrupt status is cleared, InterruptedException is thrown, and thread is resumed
  – otherwise interrupt status of thread is set
Programming with Interrupts

• Handle InterruptedException in blocking methods

```java
for (int i = 0; i < importantInfo.length; i++) {
    //Pause for 4 seconds
    try {
        Thread.sleep(4000);
    }
    catch (InterruptedException e) {
        //We've been interrupted: no more messages.
        return;
    }
    System.out.println(importantInfo[i]);
}
```
Interrupt Status

- Thread.\texttt{interrupted()} tests whether the current thread has been interrupted.
  - The \textit{interrupted status} of the thread is cleared by this method.
  - So normally programmer must code some interrupt handling whenever this returns true.

- \texttt{thread.isInterrupted()} tests whether this thread has been interrupted.
  - The \textit{interrupted status} of the thread is unaffected by this method.
Thread.interrupted()

• Programmers can explicitly check the interrupt status of thread
  – when no call throws InterruptedException
  – useful for long running methods

```java
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        //We've been interrupted: no more crunching.
        return;
    }
}
```
thread.join()

• Force the current thread to wait until another thread terminates
  – used to define synchronization points
  – This thread may continue after another thread has done its business
    • It can provide an alternative to use of locks when different threads are strongly interdependent

• join responds to an interrupt on the thread
  – by exiting, clearing interrupt status and throwing an InterruptedException
Example of Thread Join

Runnable r1 = ...;
Runnable r2 = ...;
Thread t1 = new Thread (r1);
Thread t2 = new Thread (r2);
t1.start();
t2.start();
// ... other computation. Do not use r1 or r2
	t1.join(); // block until t1 is done
	t2.join(); // block until t2 is done
//Now use r1 and r2 ... holding results of t1 and t2 computations
Terminating Threads

• Normal termination
  – run() terminates

• Abnormal termination
  – Exception that propagates beyond run() method causes thread termination
    • Can define uncaught exception handler for individual threads, or accept default

• System termination
  – Runtime.exit()
Comments on Thread Control

• In general, it is advisable not to directly control thread execution
• Better approach is to rely on a higher level pattern of thread use
• The Java Concurrency Framework provides a better option
  – Use executors to separate submission of runnable tasks from execution policy
Outline

• Thread Control
• Tasks and Task Control
• Finding and Exploiting Parallelism
Organise Task Execution

• Each task represents a small fraction of the application's processing capacity
• We should identify sensible task boundaries
  – ideally, tasks should be as independent as possible
  – independence facilitates concurrency
• Most server applications offer a natural choice of task boundary
  – individual requests as task boundaries
  – usually requests offer both independence and appropriate task sizing
Executing Tasks Sequentially

- The simplest way to organise tasks is to execute tasks sequentially in a single thread.

```java
class SingleThreadWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }
}
```
Creating Threads for Tasks

• A more responsive approach is to create a new thread for servicing each request

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

• Task processing is offloaded from the main thread
  – this enables the main loop to resume waiting for the next incoming connection more quickly

• Tasks can be processed in parallel
  – this enables multiple requests to be serviced simultaneously

• Task-handling code must be thread-safe
Disadvantages of Unbounded Thread Creation

• Thread life cycle overhead
  – thread creation and teardown are not free

• Resource consumption
  – active threads consume system resources, especially memory

• Stability
  – there is a limit on how many threads can be created

• We need to place some bound on how many threads the application creates
The Executor Framework

• We have seen two policies for executing tasks
  – execute tasks sequentially in a single thread
  – execute each task in its own thread
• Both have serious limitations
  – either poor responsiveness and throughput
  – or poor resource management
• The third option for thread management
  – use *bounded* queues to prevent an application from overloading
• The primary abstraction for task execution in the Java libraries is *not* Thread, but Executor
  – Since Java 5
The Executor Interface

• java.util.concurrent provides a flexible thread pool implementation as part of the Executor framework

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

• Decouple task submission from task execution
  – tasks are represented by Runnable objects
• Support a wide variety of task execution policies
Web Server Using Executor

class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec = Executors.newFixedThreadPool(NTHREADS);
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
}
Implement the Executor Interface

• Executor interface can be implemented to support different task execution policies

```java
public class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```

```java
public class WithinThreadExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    }
}
```
Execution Policies

- An execution policy specifies *what, where, when, and how* of task execution
  - in what thread will tasks be executed?
  - in what order should tasks be executed?
    - FIFO, LIFO, priority order
  - how many tasks may execute concurrently?
  - how many tasks may be queued pending execution?
  - if a task has to be rejected because the system is overloaded, which task should be selected as the victim, and how should the application be notified?
  - what actions should be taken before or after executing a task?
Execution Policies

• Execution policies must deal with resource management issues
  – the optimal policy depends on the available computing resources and your quality-of-service requirements

• Decoupling task submission from execution has benefits
  – we can easily specify the execution policy
    • and subsequently change it without great difficulty
  – it becomes practical to select an execution policy at deployment time
    • matching to available system resources e.g. number of CPUs
Threads Pools

• A thread pool manages a homogeneous pool of worker threads
  – bound to a queue holding tasks waiting to be executed

• The lifecycle of a worker thread
  – request the next task from the queue
  – execute the task
  – wait for another task
Creating Thread Pools

- `java.util.concurrent.Executors` provides static factory methods for creating thread pools
  - see Java API docs
- `newFixedThreadPool`
  - return a thread pool with constant size
- `newCachedThreadPool`
  - return a thread pool without bound size, but always try to reuse existing threads first
- `newSingleThreadExecutor`
- `newScheduledThreadPool`
Advantages of Thread Pools

• Reuse an existing thread instead of creating a new one
  – reduce overhead of thread creation/teardown
• Improve responsiveness
  – immediately execute a task without latency associated with thread creation
• Exploit multiple processors without exhausting resources
  – enough threads to keep processors busy
  – not too many threads to exhaust resources
ExecutorService Interface

• ExecutorService interface extends Executor, with methods for lifecycle management:

public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException;
    // ... additional convenience methods for task submission
}
Web Server with Shutdown

class LifecycleWebServer {
    private final ExecutorService exec = ...;
    public void start() throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown()) {
            try {
                Socket conn = socket.accept();
                exec.execute(new Runnable() {
                    public void run() { handleRequest(conn); }
                });
            } catch (RejectedExecutionException e) {
                if (!exec.isShutdown()) log("task submission rejected", e);
            }
        }
    }
}
public void stop() {
  exec.shutdown();
}

void handleRequest(Socket connection) {
  Request req = readRequest(connection);
  if (isShutdownRequest(req))
    stop();
  else
    dispatchRequest(req);
}
Outline

• Thread Control
• Tasks and Task Control
• Finding and Exploiting Parallelism
Case Study: Page Renderer

• The page-rendering portion of a browser application takes a page of HTML and renders it into an image buffer
• We will develop several versions that admit different degrees of concurrency
  – Examples are from Java Concurrency in Practice
• The simplest approach is to process the HTML document sequentially
  – as text markup is encountered, render it into the image buffer
  – as image references are encountered, fetch the image over the network and draw it into the image
public class SingleThreadRenderer {
    void renderPage(CharSequence source) {
        renderText(source);
        List<ImageData> imageData = new ArrayList<ImageData>();
        for (ImageInfo imagelInfo : scanForImageInfo(source))
            imageData.add(imagelInfo.downloadImage());
        for (ImageData data : imageData)
            renderImage(data);
    }
}
Runnable vs Callable

- Both *Runnable* and *Callable* are tasks that can be processed by an Executor.
  - Runnable cannot return a value or throw checked exceptions.
    - It can only have side effects.
  - Callable can return a value and throw an exception.

```java
public interface Callable<V> {
    V call() throws Exception
}
```
Callable and Future

• The lifecycle of a task executed by an Executor has four phases:
  – created, submitted, started, and completed
  – once a task is completed, it stays in that state forever

• Future represents the lifecycle of a task
  – provide methods to test whether the task has completed or been cancelled, retrieve its result, and cancel the task
public interface Future\<V\> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException,
               ExecutionException,
               CancellationException;
    V get(long timeout, TimeUnit unit) throws InterruptedException,
               ExecutionException,
               CancellationException,
               TimeoutException;
}
public interface ExecutorService extends Executor {
    // ... lifecycle management methods
    // additional convenience methods for task submission

    <T> Future<T> submit(Callable<T> task);
    <T> Future<T> submit(Runnable task, T result);
    Future<?> submit(Runnable task);

    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        long timeout, TimeUnit unit)
        throws InterruptedException;

    <T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException;
    <T> T invokeAny(Collection<? extends Callable<T>> tasks,
        long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}

ExecutorService Interface (ctd.)
Page Renderer with Future

• Divide page render into two tasks to make it more concurrent
  – one renders the text (largely CPU-bound)
  – one downloads all the images (largely I/O-bound)

• Callable and Future can help us express the interaction between these cooperating tasks
  – create a Callable to download all the images
  – a future is returned to describe the Callable execution
  – when we need the images, we call Future.get() to wait for the result
public class FutureRenderer {
    private final ExecutorService executor = ...;
    void renderPage(CharSequence source) {
        final List<ImageInfo> imageInfos = 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;
            }
        };
Future<List<ImageData>> future = executor.submit(task);
renderText(source);
try {
    List<ImageData> imageData = future.get();
    for (ImageData data : imageData)
        renderImage(data);
} catch (InterruptedException e) {
    // We don't need the result, so cancel the task too
    future.cancel(true);
} catch (ExecutionException e) {
    throw launderThrowable(e.getCause());
}
public class Renderer {
    private final ExecutorService executor;
    Renderer(ExecutorService executor) { this.executor = executor; }
    void renderPage(CharSequence source) {
        final List<ImageInfo> info = scanForImageInfo(source);
        CompletionService<ImageData> completionService =
            new ExecutorCompletionService<ImageData>(executor);
        for (final ImageInfo imageInfo : info)
            completionService.submit(new Callable<ImageData>() {
                public ImageData call() {
                    return imageInfo.downloadImage();
                }
            });
    }
}
Render Page Elements as They Become Available

renderText(source);

try {
    for (int t = 0, n = info.size(); t < n; t++) {
        Future<ImageData> f = completionService.take();
        ImageData imageData = f.get();
        renderImage(imageData);
    }
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
} catch (ExecutionException e) {
    throw launderThrowable(e.getCause());
}