2. Concurrency and Java

Oscar Nierstrasz

Selected material © Magee and Kramer
Roadmap

- **Threads in Java**
  - Threads (vs. Processes)
  - Thread creation
  - Thread lifecycle

- **Synchronization in Java**
  - `wait()` and `notify()`

- **Modelling Concurrency**
  - Finite State Processes
  - Labelled Transition Systems
Threads (vs. Processes)

- Processes
  - have their own memory
  - communicate via IPC

- Threads (lightweight processes)
  - exist within a process
  - share memory and open files
Threads are fundamental for implementing concurrency

- based on *monitors*
- Thread class and Runnable interface
- synchronized keyword (insures mutual exclusion, and thus atomicity)
- `wait()` and `notify()` methods

> The JVM runs a single process
To define a thread extend Thread

and override run()

```java
public class Competitor extends java.lang.Thread {

    public Competitor(String name) {
        super(name); // Call Thread constructor
    }

    @Override
    public void run() { // What the thread actually does
        for (int km = 0; km < 5; km++) {
            System.out.println(km + " " + getName());
            try {
                sleep(100);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("DONE: " + getName());
    }
}

ThreadDemo
```
... or implement Runnable

```java
public interface java.lang.Runnable {
    public abstract void run();
}
```

*Since Java does not support multiple inheritance, it is impossible to inherit from both Thread and another class.*

Instead, simply define:

```java
class MTUsefulStuff extends UsefulStuff implements Runnable ...
```

and instantiate:

```java
new Thread(new MTUsefulStuff())
```
Example: A thread-based visual clock component

```java
public class Clock extends Canvas implements Runnable {
    private Thread clockThread = null;

    public Clock() {
        super();
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }

    public void paintComponent(Graphics g) {
        ...
        String time = dateFormat.format(new Date());
        g2d.drawString(...);
    }
}
```
The `run()` repaints the canvas in a loop while the thread is not null.

```java
public void run() {
    // stops when clockThread is set to null
    while (Thread.currentThread() == clockThread) {
        repaint();
        try {clockThread.sleep(1000); }
        catch (InterruptedException e){ } 
    }
}

public void stopThread() {
    clockThread = null;
}
```
Instantiating Threads

A Java thread can either *inherit* from java.lang.Thread, or *contain* a Runnable object:

```java
public class java.lang.Thread
    extends java.lang.Object
    implements java.lang.Runnable
{
    public Thread();
    public Thread(String name);
    public Thread(Runnable target);
    public Thread(Runnable target, String name);
    ...
}
```
After construction...

Thread

you are here!

start()

Runnable

yield()

run() exits

time elapsed
notify() / notifyAll()
i/o completed

Not Runnable

sleep()
wait()
i/o block
To start a thread... call start()

A Thread’s run method is never called directly but is executed when the Thread is \textit{start()-ed}:

```java
class Race5K {
    public static void main (String[] args) {
        // Instantiate and start threads
        new Competitor("Tortoise").start();
        new Competitor("Hare").start();
    }
    ...
}
```
The Racing Day!

Different runs can have different results.

No given ordering of threads is guaranteed by the JVM.

Could the output be garbled? Why?

0 Hare0 Tortoise
0 Hare
1 Hare
1 Tortoise
2 Hare
2 Tortoise
3 Hare
3 Tortoise
4 Hare
4 Tortoise
DONE: Hare
DONE: Tortoise

0 Tortoise0 Hare
1 Hare
1 Tortoise
2 Hare
2 Tortoise
3 Hare
3 Tortoise
4 Hare
4 Tortoise
DONE: Tortoise
DONE: Hare

...
Roadmap

> Threads in Java
  — Threads (vs. Processes)
  — Thread creation
  — Thread lifecycle

> Synchronization in Java
  — wait() and notify()

> Modelling Concurrency
  — Finite State Processes
  — Labelled Transition Systems
Synchronization

Without synchronization, an arbitrary number of threads may run at any time within the methods of an object.

—Class invariant may not hold when a method starts!
—So can’t guarantee any post-condition!

A solution: consider critical sections which lock access to the object while it is running.
Critical Section...

> ... a piece of code that accesses a shared resource (e.g. memory location) that must not be concurrently accessed by multiple threads.

> This works as long as methods cooperate in locking and unlocking access!

```c
/* This is the critical section object (statically allocated). */
static pthread_mutex_t cs_mutex = PTHREAD_MUTEX_INITIALIZER;

void f()
{
    /* Enter the critical section
     -- other threads are locked out */
    pthread_mutex_lock( &cs_mutex );

    /* Do some thread-safe processing! */

    /* Leave the critical section
     -- other threads can now
     pthread_mutex_lock() */
    pthread_mutex_unlock( &cs_mutex );
}
```

Critical section in C/C++ on Unix/Linux
Synchronized blocks

*Either:* synchronize an individual block within a method with respect to some object:

```java
public Object aMethod() {
    // unsynchronized code
    ...
    synchronized (resource) {
        // lock resource
        ...
    } // unlock resource
    ...
}
```

*Note:* synchronized methods are a particular case of synchronizing on the *this* object
**Synchronized methods**

*Or:* declare an entire method to be *synchronized* with other synchronized methods of an object **on that object:**

```java
public class PrintStream extends FilterOutputStream {
    ...
    public synchronized void println(String s);
    public synchronized void println(char c);
    ...
}
```
wait() and notify()

Synchronization must sometimes be interrupted:

```java
public class Account {
    protected long assets = 0;
    public synchronized void withdraw(int amount) {
        while (amount > assets) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        assets -= amount;
    }

    public synchronized void deposit(int amount) {
        assets += amount;
        notifyAll();
    }
}
```

`NB: you must either catch or throw InterruptedException`
wait and notify in action …

```java
final Account myAccount = new Account();

new Thread() { // Withdrawing
    public void run() {
        int amount = 100;
        System.out.println("Waiting to withdraw"+amount+"units...");
        myAccount.withdraw(amount);
        System.out.println("I withdrew " + amount + " units!");
    }
}.start();

Thread.sleep(1000); ...

new Thread() { // Depositing
    public void run() {
        int amount = 200;
        System.out.println("Depositing " + amount + " units ...");
        myAccount.deposit(amount);
        System.out.println("I deposited " + amount + " units!");
    }
}.start();
```

Waiting to withdraw 100 units ...

Depositing 200 units ...
I deposited 200 units!
I withdrew 100 units!
NB: wait() and notify() are methods rather than keywords:

```java
public class java.lang.Object {
  ...
  public final void wait()
      throws InterruptedException;
  public final void notify();
  public final void notifyAll();
  ...
}
```
Roadmap

> Threads in Java
  — Threads (vs. Processes)
  — Thread creation
  — Thread lifecycle

> Synchronization in Java
  — wait() and notify()

> Modelling Concurrency
  — Finite State Processes
  — Labelled Transition Systems
Non-determinism

> Multiple threads are rotated by the processor(s)
> A thread might be interrupted at any time
> No two runs are guaranteed to be the same
Because concurrent systems are non-deterministic, it can be difficult to build them and reason about their properties.

A model is an abstraction of the real world that makes it easier to focus on the points of interest.

**Approach:**

Model concurrent systems as sets of sequential finite state processes.
Finite State Processes

**FSP** is a *textual notation* for specifying a finite state process:

\[ \text{SWITCH} = (\text{on} \rightarrow \text{off} \rightarrow \text{SWITCH}). \]

**LTS** (labeled transition system) is a *graphical notation* for interpreting a processes as a labelled transition system:

The *meaning* of a process is a set of possible *traces*:

\[ \text{on} \rightarrow \text{off} \rightarrow \text{on} \rightarrow \text{off} \rightarrow \text{on} \rightarrow \text{off} \rightarrow \text{on} \rightarrow \ldots \]
# FSP Summary

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Action prefix</strong></td>
<td>((x-&gt;P))</td>
</tr>
<tr>
<td><strong>Choice</strong></td>
<td>((x-&gt;P \mid y-&gt;Q))</td>
</tr>
<tr>
<td><strong>Guarded Action</strong></td>
<td>((\text{when } B \ x-&gt;P \mid y-&gt;Q))</td>
</tr>
<tr>
<td><strong>Alphabet extension</strong></td>
<td>(P + S)</td>
</tr>
<tr>
<td><strong>Conditional</strong></td>
<td>(x-&gt; \text{If } B \text{ then } P \text{ else } Q)</td>
</tr>
<tr>
<td><strong>Relabelling</strong></td>
<td>({\text{new}_1/\text{old}_1,\ldots})</td>
</tr>
<tr>
<td><strong>Hiding</strong></td>
<td>({a_1,\ldots, a_n})</td>
</tr>
<tr>
<td><strong>Interface</strong></td>
<td>({a_1,\ldots, a_n})</td>
</tr>
</tbody>
</table>

| **Parallel composition**| \((P \mid | Q)\)                                                            |
| **Replicator**          | \(\text{forall } [I:1..N] \ P(I)\)                                        |
| **Process labelling**   | \(a:P\)                                                                    |
| **Process sharing**     | \(\{a_1,\ldots, a_n\}:P\)                                                |
| **Priority High**       | \(\|C=(P \mid | Q)\ll\{a_1,\ldots, a_n\}\)                                |
| **Priority Low**        | \(\|C=(P \mid | Q)\gg\{a_1,\ldots, a_n\}\)                                |
| **Safety property**     | Property \(P\)                                                            |
| **Progress property**   | Progress \(P = \{a_1,\ldots, a_n\}\)                                     |

We will encounter and use these features in the lectures to come …
FSP — Action Prefix

If \( x \) is an action and \( P \) a process then \( (x \rightarrow P) \) is a process that initially engages in the action \( x \) and then behaves like \( P \).

ONESHOT = \((\text{once} \rightarrow \text{STOP})\).

Convention:
Processes start with UPPERCASE, actions start with lowercase.
Repetitive behaviour uses recursion:

\[
\text{SWITCH} = \text{OFF}, \\
\text{OFF} = (\text{on} \to \text{ON}), \\
\text{ON} = (\text{off} \to \text{OFF}).
\]
If $x$ and $y$ are actions then $(x \rightarrow P \mid y \rightarrow Q)$ is a process which initially engages in *either* of the actions $x$ or $y$.

If $x$ occurs, the process then behaves like $P$; otherwise, if $y$ occurs, it behaves like $Q$.

**FSP — Choice**

**DRINKS** =

( red $\rightarrow$ coffee $\rightarrow$ DRINKS  
| blue $\rightarrow$ tea $\rightarrow$ DRINKS  
).

What are the possible traces of **DRINKS**?
FSP — Non-determinism

\((x \rightarrow P \ | \ x \rightarrow Q)\) performs \(x\) and then behaves as either \(P\) or \(Q\).

\[
\text{COIN} = \\
( \text{toss} \rightarrow \text{heads} \rightarrow \text{COIN} \\
| \text{toss} \rightarrow \text{tails} \rightarrow \text{COIN} \\
).
\]
(when \( B \ x\rightarrow P \mid y\rightarrow Q \)) means that when the guard \( B \) is true then either \( x \) or \( y \) may be chosen; otherwise if \( B \) is false then only \( y \) may be chosen. (default case is optional)

\[
\text{COUNT (N=3)} = \text{COUNT}[0], \\
\text{COUNT}[i:0..N] = \\
( \text{when}(i<N) \text{inc} \rightarrow \text{COUNT}[i+1] \\
| \text{when}(i>0) \text{dec} \rightarrow \text{COUNT}[i-1] \\
).
\]
Competitor can be modelled as a single, sequential, finite state process:

\[ \text{Competitor } = ([1] \rightarrow [2] \rightarrow [3] \rightarrow [4] \rightarrow \text{done} \rightarrow \text{STOP}). \]

Or, more generically:

\[
\begin{align*}
\text{const } N &= 4 \\
\text{COMPETITOR} &= \text{KM}[0], \\
\text{KM}[n:0..N] &= (\text{when } (n<N) [n+1] \rightarrow \text{KM}[n+1] \\
&\quad \text{when } (n==N) \text{ done } \rightarrow \text{STOP}).
\end{align*}
\]
We can *relabel* the transitions of *Simple* and concurrently *compose* two copies of it:

\[ ||\text{Race5K} = (\text{tortoise:Competitor} \ || \ \text{hare:Competitor}). \]
If we restrict ourselves to two steps, the composition will have nine states:
The state space of two composed processes is (at most) the Cartesian product of the individual state spaces.
Transitions between Thread States

Thread

Runnable

Not Runnable

start()
yield()
run() exits
time elapsed
notify() / notifyAll()
i/o completed

sleep()
wait()
i/o block
LTS for Thread States

Thread = ( start -> Runnable ),
Runnable =
  ( yield -> Runnable
  | {sleep, wait, blockio} -> NotRunnable
  | stop -> STOP ),
NotRunnable =
  ( {awake, notify, unblockio} -> Runnable ).
What you should know!

- What are finite state processes?
- How are they used to model concurrency?
- What are traces, and what do they model?
- How can the same FSP have multiple traces?
- How do you create a new thread in Java?
- What states can a Java thread be in?
- How can it change state?
- What is the Runnable interface good for?
- What is a critical section?
- When should you declare a method to be synchronized?
Can you answer these questions?

> How do you specify an FSP that repeatedly performs `hello`, but may stop at any time?
> How many states and how many possible traces does the full Race5K FSP have?
> When should your class inherit from `Thread`?
> How can concurrency invalidate a class invariant?
> What happens if you call `wait` or `notify` outside a synchronized method or block?
> When is it better to use synchronized blocks rather than methods?
Attribution-ShareAlike 3.0

You are free:
- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

BY: Attribution. You must attribute the work in the manner specified by the author or licensor.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

http://creativecommons.org/licenses/by-sa/3.0/