Concurrent Programming

Prof. O. Nierstrasz

IAM, University of Berne

Winter 2001/2002
# Table of Contents

1. Concurrent Programming
   - Goals of this course 1
   - Schedule 2
   - Introduction 3
   - Recommended reading 4
   - Concurrency 5
   - Parallelism 6
   - Why do we need concurrent programs? 7
   - Difficulties 8
   - Concurrency and atomicity 9
   - Safety 10
   - Liveness 11
   - Expressing Concurrency 12
   - Process Creation 13
   - Co-routines 14
   - Fork and Join 15
   - Cobegin/coend 16
   - Communication and Synchronization 17
   - Synchronization Techniques 18
   - Busy-Waiting 19
   - Semaphores 20
   - Programming with semaphores 21
   - Monitors 22
   - Programming with monitors 23
   - Problems with monitors 24
   - Path Expressions 25
   - Message Passing 26
   - Send and Receive 27
   - Remote Procedure Calls and Rendezvous 28
   - What you should know! 29

2. Java and Concurrency
   - Can you answer these questions? 31
   - Modelling Concurrency 32
   - Finite State Processes 33
   - FSP — Action Prefix 34
   - FSP — Recursion 35
   - FSP — Choice 36
   - FSP — Non-determinism 37
   - FSP — Guarded actions 38
   - Java 39
   - Threads 40
   - SimpleThread FSP 41
   - Multiple Threads ... 42
   - Running the TwoThreadsDemo 43
   - FSP — Concurrency 44
   - FSP — Composition 45
   - java.lang.Thread (creation) 46
   - java.lang.Thread (methods) 47
   - java.langRunnable 48
   - Transitions between Thread States 49
   - LTS for Threads 50
   - Creating Threads 51
   - Creating Threads ... 52
   - ... And stopping them 53
   - Synchronization 54
   - Synchronized methods 55
   - Synchronized blocks 56
   - wait and notify 57
   - java.lang.Object 58
3. Safety and Synchronization

Modelling interaction — shared actions
Modelling interaction — handshake
Modelling interaction — multiple processes
Safety problems
Atomicity and interference
Atomic actions
Sequential behaviour
Concurrent behaviour
Locking
Synchronization
Synchronization in Java
Busy-Wait Mutual Exclusion Protocol
Atomic read and write
Modelling the busy-wait protocol
Busy-wait composition
Checking for errors
Conditional synchronization
Producer/Consumer composition
Wait and notify
Slot (put)
Slot (get)
Producer in Java
Consumer in Java
Composing Producers and Consumers
What you should know!
Can you answer these questions?

4. Safety Patterns

Idioms, Patterns and Architectural Styles
Pattern: Immutable classes
Immutability variants
Immutable classes — design steps
Design steps...

Pattern: Fully Synchronized Objects
Applicability...
Full Synchronization — design steps
Design steps...

Example: a BalkingBoundedCounter
Example: an ExpandableArray
Example...

Bundling Atomicity
Using inner classes
Pattern: Partial Synchronization
Partial Synchronization — design steps
Example: LinkedCells
Example...

Pattern: Containment
Applicability...
Contained Objects — design steps
Design steps...
Managed Ownership
Managed Ownership...
A minimal transfer protocol class
What you should know!
Can you answer these questions?

5. Liveness and Deadlock

Safety revisited
Safety — property specification
Safety properties
Safety properties
Liveness
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Liveness Problems</td>
<td>124</td>
</tr>
<tr>
<td>Progress properties — fair choice</td>
<td>125</td>
</tr>
<tr>
<td>Progress properties</td>
<td>126</td>
</tr>
<tr>
<td>Progress properties</td>
<td>127</td>
</tr>
<tr>
<td>Progress analysis</td>
<td>128</td>
</tr>
<tr>
<td>Deadlock</td>
<td>129</td>
</tr>
<tr>
<td>Waits-for cycle</td>
<td>130</td>
</tr>
<tr>
<td>Deadlock analysis - primitive processes</td>
<td>131</td>
</tr>
<tr>
<td>The Dining Philosophers Problem</td>
<td>132</td>
</tr>
<tr>
<td>Deadlocked diners</td>
<td>133</td>
</tr>
<tr>
<td>Dining Philosophers, Safety and Liveness</td>
<td>134</td>
</tr>
<tr>
<td>Dining Philosophers ...</td>
<td>135</td>
</tr>
<tr>
<td>Modeling Dining Philosophers</td>
<td>136</td>
</tr>
<tr>
<td>Dining Philosophers Analysis</td>
<td>137</td>
</tr>
<tr>
<td>Eliminating Deadlock</td>
<td>138</td>
</tr>
<tr>
<td>Dining Philosopher Solutions</td>
<td>139</td>
</tr>
<tr>
<td>What you should know!</td>
<td>140</td>
</tr>
<tr>
<td>Can you answer these questions?</td>
<td>141</td>
</tr>
<tr>
<td>6. Liveness and Guarded Methods</td>
<td>142</td>
</tr>
<tr>
<td>Achieving Liveness</td>
<td>143</td>
</tr>
<tr>
<td>Pattern: Guarded Methods</td>
<td>144</td>
</tr>
<tr>
<td>Guarded Methods — applicability</td>
<td>145</td>
</tr>
<tr>
<td>Applicability ...</td>
<td>146</td>
</tr>
<tr>
<td>Guarded Methods — design steps</td>
<td>147</td>
</tr>
<tr>
<td>Step: Separate interface from policy</td>
<td>148</td>
</tr>
<tr>
<td>Step: Check guard conditions</td>
<td>149</td>
</tr>
<tr>
<td>Step: Check guard conditions ...</td>
<td>150</td>
</tr>
<tr>
<td>Step: Handle interrupts</td>
<td>151</td>
</tr>
<tr>
<td>Step: Signal state changes</td>
<td>152</td>
</tr>
<tr>
<td>Notify() vs notifyall()</td>
<td>153</td>
</tr>
<tr>
<td>Step: Structure notifications</td>
<td>154</td>
</tr>
<tr>
<td>Encapsulating assignment</td>
<td>156</td>
</tr>
<tr>
<td>Tracking State</td>
<td>158</td>
</tr>
<tr>
<td>Tracking State Variables</td>
<td>159</td>
</tr>
<tr>
<td>Delegating notifications</td>
<td>161</td>
</tr>
<tr>
<td>Delegating notifications ...</td>
<td>162</td>
</tr>
<tr>
<td>What you should know!</td>
<td>163</td>
</tr>
<tr>
<td>Can you answer these questions?</td>
<td>164</td>
</tr>
<tr>
<td>7. Lab session I</td>
<td>165</td>
</tr>
<tr>
<td>8. Liveness and Asynchrony</td>
<td>166</td>
</tr>
<tr>
<td>Pattern: Asynchronous Invocations</td>
<td>167</td>
</tr>
<tr>
<td>Asynchronous Invocations — form</td>
<td>168</td>
</tr>
<tr>
<td>Asynchronous Invocations — design steps</td>
<td>169</td>
</tr>
<tr>
<td>Simple Relays — three variants</td>
<td>171</td>
</tr>
<tr>
<td>Variant: Direct invocations</td>
<td>172</td>
</tr>
<tr>
<td>Direct invocations ...</td>
<td>173</td>
</tr>
<tr>
<td>Variant: Thread-based messages</td>
<td>174</td>
</tr>
<tr>
<td>Thread-based messages ...</td>
<td>175</td>
</tr>
<tr>
<td>Thread-per-message Gateways</td>
<td>176</td>
</tr>
<tr>
<td>Variant: Command-based messages</td>
<td>177</td>
</tr>
<tr>
<td>Tail calls</td>
<td>178</td>
</tr>
<tr>
<td>Tail calls with new threads</td>
<td>179</td>
</tr>
<tr>
<td>Early Reply</td>
<td>180</td>
</tr>
<tr>
<td>Simulating Early Reply</td>
<td>181</td>
</tr>
<tr>
<td>Early Reply in Java</td>
<td>182</td>
</tr>
<tr>
<td>Futures</td>
<td>183</td>
</tr>
<tr>
<td>A Future Class</td>
<td>184</td>
</tr>
<tr>
<td>Using Futures in Java</td>
<td>185</td>
</tr>
<tr>
<td>What you should know!</td>
<td>186</td>
</tr>
<tr>
<td>Can you answer these questions?</td>
<td>187</td>
</tr>
<tr>
<td>9. Condition Objects</td>
<td>188</td>
</tr>
<tr>
<td>Pattern: Condition Objects</td>
<td>189</td>
</tr>
<tr>
<td>Condition Objects — applicability</td>
<td>190</td>
</tr>
<tr>
<td>Condition Objects</td>
<td>191</td>
</tr>
<tr>
<td>A Simple Condition Object</td>
<td>192</td>
</tr>
<tr>
<td>The Nested Monitor problem</td>
<td>193</td>
</tr>
<tr>
<td>The Nested Monitor problem</td>
<td>194</td>
</tr>
<tr>
<td>The Nested Monitor problem</td>
<td>195</td>
</tr>
<tr>
<td>Nested Monitors in FSP</td>
<td>196</td>
</tr>
<tr>
<td>Nested Monitors in FSP</td>
<td>197</td>
</tr>
<tr>
<td>Nested Monitors in FSP</td>
<td>198</td>
</tr>
<tr>
<td>Solving the Nested Monitors problem</td>
<td>199</td>
</tr>
<tr>
<td>Solving Nested Monitors</td>
<td>200</td>
</tr>
<tr>
<td>Example solution</td>
<td>201</td>
</tr>
<tr>
<td>Pattern: Permits and Semaphores</td>
<td>202</td>
</tr>
<tr>
<td>Permits and Semaphores — design steps</td>
<td>203</td>
</tr>
<tr>
<td>Design steps</td>
<td>204</td>
</tr>
<tr>
<td>Variants</td>
<td>205</td>
</tr>
<tr>
<td>Semaphores in Java</td>
<td>206</td>
</tr>
<tr>
<td>Using Semaphores</td>
<td>207</td>
</tr>
<tr>
<td>Using Semaphores</td>
<td>208</td>
</tr>
<tr>
<td>Using Semaphores</td>
<td>209</td>
</tr>
<tr>
<td>What you should know!</td>
<td>210</td>
</tr>
<tr>
<td>Can you answer these questions?</td>
<td>211</td>
</tr>
<tr>
<td>10. Fairness and Optimism</td>
<td>212</td>
</tr>
<tr>
<td>Pattern: Concurrently Available Methods</td>
<td>213</td>
</tr>
<tr>
<td>Concurrent Methods — design steps</td>
<td>214</td>
</tr>
<tr>
<td>Priority</td>
<td>215</td>
</tr>
<tr>
<td>Fairness</td>
<td>216</td>
</tr>
<tr>
<td>Interception</td>
<td>217</td>
</tr>
<tr>
<td>Concurrent Reader and Writers</td>
<td>218</td>
</tr>
<tr>
<td>Readers/Writers Model</td>
<td>219</td>
</tr>
<tr>
<td>A Simple RW Protocol</td>
<td>220</td>
</tr>
<tr>
<td>Safety properties</td>
<td>221</td>
</tr>
<tr>
<td>Safety properties</td>
<td>222</td>
</tr>
<tr>
<td>Composing the Readers and Writers</td>
<td>223</td>
</tr>
<tr>
<td>Progress properties</td>
<td>224</td>
</tr>
<tr>
<td>Starvation</td>
<td>225</td>
</tr>
<tr>
<td>Readers and Writers Policies</td>
<td>226</td>
</tr>
<tr>
<td>Policies</td>
<td>227</td>
</tr>
<tr>
<td>Readers and Writers example</td>
<td>228</td>
</tr>
<tr>
<td>Readers and Writers example</td>
<td>229</td>
</tr>
<tr>
<td>Readers and Writers example</td>
<td>230</td>
</tr>
<tr>
<td>Readers and Writers example</td>
<td>231</td>
</tr>
<tr>
<td>Pattern: Optimistic Methods</td>
<td>232</td>
</tr>
<tr>
<td>Optimistic Methods — design steps</td>
<td>233</td>
</tr>
<tr>
<td>Detect failure</td>
<td>234</td>
</tr>
<tr>
<td>Detect failure</td>
<td>235</td>
</tr>
<tr>
<td>Handle conflicts</td>
<td>236</td>
</tr>
<tr>
<td>Ensure progress</td>
<td>237</td>
</tr>
<tr>
<td>An Optimistic Bounded Counter</td>
<td>238</td>
</tr>
<tr>
<td>An Optimistic Bounded Counter</td>
<td>239</td>
</tr>
<tr>
<td>What you should know!</td>
<td>240</td>
</tr>
<tr>
<td>Can you answer these questions?</td>
<td>241</td>
</tr>
<tr>
<td>11. Lab session II</td>
<td>242</td>
</tr>
<tr>
<td>12. Architectural Styles for Concurrency</td>
<td>243</td>
</tr>
<tr>
<td>Sources</td>
<td>244</td>
</tr>
<tr>
<td>Software Architecture</td>
<td>245</td>
</tr>
<tr>
<td>Architectural style</td>
<td>246</td>
</tr>
<tr>
<td>Communication Styles</td>
<td>247</td>
</tr>
<tr>
<td>Simulated Message-Passing</td>
<td>248</td>
</tr>
<tr>
<td>Three-layered Application Architectures</td>
<td>249</td>
</tr>
<tr>
<td>Problems with Layered Designs</td>
<td>250</td>
</tr>
<tr>
<td>Flow Architectures</td>
<td>251</td>
</tr>
<tr>
<td>Unix Pipes</td>
<td>252</td>
</tr>
<tr>
<td>Unix Pipes</td>
<td>253</td>
</tr>
<tr>
<td>Flow Stages</td>
<td>254</td>
</tr>
<tr>
<td>Flow Policies</td>
<td>255</td>
</tr>
<tr>
<td>Limiting Flow</td>
<td>Producers and Consumers</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>Example: a Pull-based Prime Sieve</td>
<td>Bounded Buffers</td>
</tr>
<tr>
<td>Using Put-Take Buffers</td>
<td>Reachability and Boundedness</td>
</tr>
<tr>
<td>The PrimeSieve</td>
<td>Liveness and Deadlock</td>
</tr>
<tr>
<td>Pull-based integer sources</td>
<td>Related Models</td>
</tr>
<tr>
<td>The ActivePrime Class</td>
<td>Zero-testing Nets</td>
</tr>
<tr>
<td>The ActivePrime Class ...</td>
<td>Other Variants</td>
</tr>
<tr>
<td>The ActivePrime Class ...</td>
<td>Applications of Petri nets</td>
</tr>
<tr>
<td>Blackboard Architectures</td>
<td>Implementing Petri nets</td>
</tr>
<tr>
<td>Result Parallelism</td>
<td>Centralized schemes</td>
</tr>
<tr>
<td>Agenda Parallelism</td>
<td>Decentralized schemes</td>
</tr>
<tr>
<td>Specialist Parallelism</td>
<td>Transactions</td>
</tr>
<tr>
<td>Linda</td>
<td>Coordinated interaction</td>
</tr>
<tr>
<td>Linda primitives</td>
<td>What you should know!</td>
</tr>
<tr>
<td>Example: Fibonacci</td>
<td>Can you answer these questions?</td>
</tr>
</tbody>
</table>

**13. Petri Nets**

<table>
<thead>
<tr>
<th>Petri nets: a definition</th>
<th>281</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firing transitions</td>
<td>282</td>
</tr>
<tr>
<td>Modelling with Petri nets</td>
<td>283</td>
</tr>
<tr>
<td>Concurrency</td>
<td>284</td>
</tr>
<tr>
<td>Conflict</td>
<td>285</td>
</tr>
<tr>
<td>Mutual Exclusion</td>
<td>286</td>
</tr>
<tr>
<td>Fork and Join</td>
<td>287</td>
</tr>
</tbody>
</table>
## 1. Concurrent Programming

<table>
<thead>
<tr>
<th>Lecturer</th>
<th>Prof. Oscar Nierstrasz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assistants</td>
<td>Franz Achermann, Nathanael Scherli</td>
</tr>
<tr>
<td>Where</td>
<td>IWI 001, Mondays @ 10h15-12h00</td>
</tr>
<tr>
<td>WWW</td>
<td><a href="http://www.iam.unibe.ch/~scg/Teaching/CP/">www.iam.unibe.ch/~scg/Teaching/CP/</a></td>
</tr>
<tr>
<td>Texts</td>
<td>D. Lea, <em>Concurrent Programming in Java: Design Principles and Patterns</em>, Addison-Wesley, 1996</td>
</tr>
</tbody>
</table>
Goals of this course

- Introduce *basic concepts* of concurrency
  - safety, liveness, fairness

- Present tools for *reasoning* about concurrency
  - LTS, Petri nets

- Learn the *best practice* programming techniques
  - idioms and patterns

- Get *experience* with the techniques
  - lab sessions
Schedule

1. 10 - 29  Introduction
2. 11 - 05  Concurrency and Java
3. 11 - 12  Safety and Synchronization
4. 11 - 19  Safety Patterns
5. 11 - 26  Liveness and Deadlock
6. 12 - 03  Liveness and Guarded Methods
7. 12 - 10  Lab session
8. 12 - 17  Liveness and Asynchrony
9. 01 - 07  Condition Objects
10. 01 - 14  Fairness and Optimism
11. 01 - 21  Lab session
12. 01 - 28  Architectural Styles for Concurrency
13. 02 - 04  Petri Nets
Introduction

Overview
- Concurrency and Parallelism
- Applications
- Difficulties
  - safety, liveness, non-determinism ...

Concurrent Programming Approaches
- Process creation
- Communication and synchronization
  - Shared variables
  - Message Passing Approaches
Recommended reading

- A. Burns, G. Davies, *Concurrent Programming*, Addison-Wesley, 1993
Concurrency

- A **sequential program** has a *single thread of control*. Its execution is called a **process**.

- A **concurrent program** has *multiple threads of control*. These may be executed as parallel processes.
# Parallelism

A concurrent program can be executed by:

<table>
<thead>
<tr>
<th>Multiprogramming:</th>
<th>processes share one or more processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiprocessing:</td>
<td>each process runs on its own processor but with shared memory</td>
</tr>
<tr>
<td>Distributed processing:</td>
<td>each process runs on its own processor connected by a network to others</td>
</tr>
</tbody>
</table>

Assume only that all processes make positive finite progress.
Why do we need concurrent programs?

Reactive programming
- minimize response delay; maximize throughput

Real-time programming
- process control applications

Simulation
- modelling real-world concurrency

Parallelism
- speed up execution by using multiple CPUs

Distribution
- coordinate distributed services
Difficulties

But concurrent applications introduce complexity:

Safety
- concurrent processes may corrupt shared data

Liveness
- processes may “starve” if not properly coordinated

Non-determinism
- the same program run twice may give different results

Run-time overhead
- thread construction, context switching and synchronization take time
Concurrency and atomicity

Programs P1 and P2 execute concurrently:

\[
\{ x = 0 \}
\]

P1: \[ x := x + 1 \]
P2: \[ x := x + 2 \]

\[
\{ x = ? \}
\]

What are possible values of x after P1 and P2 complete?

What is the intended final value of x?
Safety

Safety = ensuring \textit{consistency}

A \textit{safety property} says “nothing bad happens”

- \textbf{Mutual exclusion:} shared resources must be \textit{updated atomically}

- \textbf{Condition synchronization:} operations may be \textit{delayed} if shared resources are in the wrong state
  (e.g., read from empty buffer)
Liveness

Liveness = ensuring progress

A liveness property says “something good happens”

- No Deadlock: some process can always access a shared resource

- No Starvation: all processes can eventually access shared resources
Expressing Concurrency

A programming language must provide mechanisms for:

Process creation
- how do you specify concurrent processes?

Communication
- how do processes exchange information?

Synchronization
- how do processes maintain consistency?
Process Creation

Most concurrent languages offer some variant of the following:

- Co-routines
- Fork and Join
- Cobegin/coend
Co-routines are only pseudo-concurrent and require explicit transfers of control:

Co-routines can be used to implement most higher-level concurrent mechanisms.
**Fork and Join**

*Fork* can be used to create any number of processes:

```
Program P1  Program P2  Program P3
fork P2  join P2  fork P3
```

*Join* waits for another process to terminate.

Fork and join are *unstructured*, so require *care and discipline*.
Cobegin/coend blocks are better structured:

```plaintext
cobegin S1 || S2 || ... || Sn coend
```

but they can only create a fixed number of processes.

The caller continues when all of the coblocks have terminated.
Communication and Synchronization

In approaches based on *shared variables*, processes communicate *indirectly*. *Explicit synchronization mechanisms* are needed.

In *message passing* approaches, communication and synchronization are *combined*. Communication may be either *synchronous* or *asynchronous*.
Synchronization Techniques

Different approaches are roughly *equivalent in expressive power* and can be used to implement each other.

*Procedure Oriented*

- Busy-Waiting
- Semaphores
- Monitors
- Path Expressions

*Message Oriented*

- Message Passing
- Remote Procedure Call

*Operation Oriented*

*Each approach emphasizes a different style of programming.*
Busy-Waiting

Busy-waiting is primitive but effective

Processes atomically set and test shared variables.

Condition synchronization is easy to implement:

- to signal a condition, a process sets a shared variable
- to wait for a condition, a process repeatedly tests the variable

Mutual exclusion is more difficult to realize correctly and efficiently.
Semaphores

Semaphores were introduced by Dijkstra (1968) as a higher-level primitive for process synchronization.

A **semaphore** is a non-negative, integer-valued variable $s$ with two operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(s)$:</td>
<td>delays until $s&gt;0$ then, atomically executes $s := s-1$</td>
</tr>
<tr>
<td>$V(s)$</td>
<td>atomically executes $s := s+1$</td>
</tr>
</tbody>
</table>
Programming with semaphores

Many problems can be solved using *binary semaphores*, which take on values 0 or 1.

```plaintext
process P1
  loop
    P(mutex) { wants to enter }
    Critical Section
    V(mutex) { exits }
    Non-critical Section
  end
end

process P2
  loop
    P(mutex)
    Critical Section
    V(mutex)
    Non-critical Section
  end
end
```
Monitors

A monitor encapsulates resources and operations that manipulate them:

- operations are invoked like ordinary procedure calls
  - invocations are guaranteed to be mutually exclusive

  condition synchronization is realized using wait and signal primitives

  there exist many variations of wait and signal ...
Programming with monitors

**type** buffer(T) = monitor

var

slots : array [0..N-1] of T;

head, tail : 0..N-1;

size : 0..N;

notfull, notempty:condition;

procedure deposit(p : T);

begin

if size = N then

    notfull.wait

slots[tail] := p;

size := size + 1;

tail := (tail+1) mod N;

notempty.signal
end

procedure fetch(var it : T);

begin

    if size = 0 then

        notempty.wait

    it := slots[head];

    size := size - 1;

    head := (head+1) mod N;

    notfull.signal

begin

    size := 0;

    head := 0;

    tail := 0;

end
Problems with monitors

Monitors are more structured than semaphores, but they are still tricky to program:

- Conditions must be manually checked
- Simultaneous signal and return is not supported

A signalling process is temporarily suspended to allow waiting processes to enter!

- Monitor state may change between signal and resumption of signaller
- Unlike with semaphores, multiple signals are not saved
- Nested monitor calls must be specially handled to prevent deadlock
Path Expressions

Path expressions express the *allowable sequence of operations* as a kind of regular expression:

```
buffer : (put; get) *
```

*Although they elegantly express solutions to many problems, path expressions are too limited for general concurrent programming.*
Message Passing

Message passing combines communication and synchronization:

- The sender specifies the message and a destination
  - a process, a port, a set of processes, ...

- The receiver specifies message variables and a source
  - source may or may not be explicitly identified

- Message transfer may be:
  - **asynchronous**: send operations never block
  - **buffered**: sender may block if the buffer is full
  - **synchronous**: sender and receiver must both be ready
Send and Receive

In CSP and Occam, source and destination are explicitly named:

PROC buffer(CHAN OF INT give, take, signal)

...SEQ

numitems := 0 ...
WHILE TRUE
ALT

numitems ≤ size & give?thebuffer[inindex]

SEQ
numitems := numitems + 1
inindex := (inindex + 1) REM size
numitems > 0 & signal?any

SEQ
take!thebuffer[outindex]
numitems := numitems - 1
outindex := (outindex + 1) REM size
Remote Procedure Calls and Rendezvous

In Ada, the caller identity need not be known in advance:

task body buffer is ...
begin loop
    select
        when no_of_items < size =>
            accept give(x : in item) do
                the_buffer(in_index) := x;
            end give;
            no_of_items := no_of_items + 1; ...
        or
            when no_of_items > 0 =>
                accept take(x : out item) do
                    x := the_buffer(out_index);
                end take;
                no_of_items := no_of_items - 1; ...
        end select;
    end loop; ...
What you should know!

✎ Why do we need concurrent programs?
✎ What problems do concurrent programs introduce?
✎ What are safety and liveness?
✎ What is the difference between deadlock and starvation?
✎ How are concurrent processes created?
✎ How do processes communicate?
✎ Why do we need synchronization mechanisms?
✎ How do monitors differ from semaphores?
✎ In what way are monitors equivalent to message-passing?
Can you answer these questions?

✎ What is the difference between concurrency and parallelism?
✎ When does it make sense to use busy-waiting?
✎ Are binary semaphores as good as counting semaphores?
✎ How could you implement a semaphore using monitors?
✎ How would you implement monitors using semaphores?
✎ What problems could nested monitors cause?
✎ Is it better when message passing is synchronous or asynchronous?
2. Java and Concurrency

Overview

- Modelling Concurrency
  - Finite State Processes
  - Labelled Transition Systems

- Java
  - Thread creation
  - Thread lifecycle
  - Synchronization

Selected material © Magee and Kramer
Modelling Concurrency

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** is a *graphical* notation for interpreting a process as a labelled transition system:

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

\[ \text{on} \rightarrow \text{off} \rightarrow \text{on} \rightarrow \text{off} \rightarrow \text{on} \rightarrow \text{off} \rightarrow \text{on} \rightarrow \text{off} \rightarrow \text{on} \ldots \]
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 $P$.

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

Convention:
- Processes start with UPPERCASE, actions start with lowercase.
FSP — Recursion

Repetitive behaviour uses recursion:

\[
\begin{align*}
\text{SWITCH} & = \text{OFF}, \\
\text{OFF} & = (\text{on} \rightarrow \text{ON}), \\
\text{ON} & = (\text{off} \rightarrow \text{OFF}).
\end{align*}
\]
FSP — Choice

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 \( Q \).

\[
\text{DRINKS} = (\text{red} \rightarrow \text{coffee} \rightarrow \text{DRINKS} \mid \text{blue} \rightarrow \text{tea} \rightarrow \text{DRINKS}).
\]

What are the possible traces of \( \text{DRINKS} \)?
FSP — Non-determinism

\[(x \rightarrow P \mid x \rightarrow Q)\] performs \(x\) and then behaves as either \(P\) or \(Q\).

\[
\text{COIN} = ( \text{toss} \rightarrow \text{heads} \rightarrow \text{COIN} \\
| \text{toss} \rightarrow \text{tails} \rightarrow \text{COIN} \\
).
\]
FSP — Guarded actions

(when \( B \) x->P | y->Q) means that \textit{when the guard} \( B \) \textit{is true}
then \textit{either} \( x \) \textit{or} \( y \) \textit{may be chosen};
otherwise if \( B \) \textit{is false} then \textit{only} \( y \) \textit{may be chosen}.

\[
\begin{align*}
\text{COUNT} \ (N=3) &= \text{COUNT}[0], \\
\text{COUNT}[i:0..N] &= (\ \text{when}(i<N) \ \text{inc}->\text{COUNT}[i+1] \\
&\quad | \ \text{when}(i>0) \ \text{dec}->\text{COUNT}[i-1]) .
\end{align*}
\]
Java

Syntax resembles C++; semantics resembles Smalltalk:
- Strongly-typed, concurrent, “pure” object-oriented
- Single-inheritance but *multiple subtyping*
- Automatic *garbage collection*

Innovation in support for network applications:
- *Standard APIs* for concurrency, network interaction
- Classes can be *dynamically loaded* over network
- *Security model* protects clients from malicious objects

*Java applications do not have to be installed by users*
Threads

A Java Thread has a **run method** defining its behaviour:

```java
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str); // Call Thread constructor
    }
    public void run() { // What the thread does
        for (int i=0; i<5; i++) {
            System.out.println(i + " " + getName());
            try { sleep((int)(Math.random()*1000));
            } catch (InterruptedException e) { } }
        System.out.println("DONE! " + getName());
    }
}
```
SimpleThread FSP

SimpleThread can be modelled as a single, sequential, finite state process:

\[
\text{Simple} = ([1]->[2]->[3]->[4]-> \text{done} -> \text{STOP}).
\]

Or, more generically:

\[
\begin{align*}
\text{const } N &= 5 \\
\text{Simple} &= \text{Print}[1], \\
\text{Print}[n:1..N] &= ( \text{when}(n<N) [n] -> \text{Print}[n+1] \\
&\quad \text{when}(n==N) \text{done} -> \text{STOP}).
\end{align*}
\]
Multiple Threads ...

A Thread's run method is never called directly but is executed when the Thread is started:

class TwoThreadsDemo {
    public static void main (String[] args) {
        // Instantiate a Thread, then start it:
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}

Running the TwoThreadsDemo

In this implementation of Java, the execution of the two threads is **interleaved**.

- This is *not guaranteed* for all implementations!

- Why are the output lines never garbled?

E.g.

- 0 Ja0 Fimajiica
- ...

- 0 Jamaica
- 0 Fiji
- 1 Jamaica
- 1 Fiji
- 2 Fiji
- 3 Fiji
- 2 Jamaica
- 4 Fiji
- 3 Jamaica
- DONE! Fiji
- 4 Jamaica
- DONE! Jamaica
FSP — Concurrency

We can *relabel* the transitions of Simple and concurrently *compose* two copies of it:

```
||TwoThreadsDemo = ( fiji:Simple
|| jamaica:Simple )
```

What are all the possible traces?
FSP — Composition

If we restrict ourselves to two steps, the composition will have nine states:
java.lang.Thread (creation)

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(Runnable target);
    public Thread(Runnable target, String name);
    public Thread(String name);
    ...
```
java.lang.Thread (methods)

A thread must be created, and then started:

... public void run();
public synchronized void start();
public static void sleep(long millis)
    throws InterruptedException;
public static void yield();
public final String getName();
...

NB: suspend(), resume() and stop() are now deprecated!
Since Java does not support multiple inheritance, it is impossible to inherit from both Thread and another class.

Instead, simply define:

class MyStuff extends UsefulStuff
 implements Runnable ...
Transitions between Thread States

Thread

Runnable

yield()

run() exits

start()

yield()

time elapsed

notify() or notifyAll()

I/O completed

sleep()

wait()

block on I/O

Not Runnable

© O. Nierstrasz — U. Berne
Java and Concurrency
LTS for Threads

Thread = ( start -> Runnable ),
Runnable =
    ( yield -> Runnable
 | {sleep, wait, blockio} -> NotRunnable
 | stop -> STOP ),
NotRunnable =
    ( {awake, notify, unblockio} -> Runnable ).
Creating Threads

This Clock applet uses a thread to update the time:

```java
public class Clock extends java.applet.Applet
    implements Runnable
{
    Thread clockThread = null;
    public void start() {
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        } ...
```
Creating Threads ...

...

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

...
... And stopping them

... public void paint(Graphics g) {
    Date now = new Date();
    g.drawString(now.getHours() + " : " + now.getMinutes() + " : " + now.getSeconds(), 5, 10);
}

// When the applet stops, stop its thread
public void stop() { clockThread = null; }

Be careful — Applets and Threads have strangely similar interfaces!
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 a method to be a critical section which locks access to the object while it is running.

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

Either: declare an entire method to be synchronized with other synchronized methods of an object:

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

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

```java
public Object aMethod() {
    // unsynchronized code
    ...
    synchronized(resource) { // Lock resource
        ...
    } // unlock resource
    ...
}
```
wait and notify

Synchronization must sometimes be interrupted:

class *Slot* implements Buffer {
    private Object *slotVal*;
    public synchronized void *put* (Object val) {
        while (*slotVal* != null) { // wait till empty
            try {
                *wait*();
            }
            catch (InterruptedException e) {
            }
        }
        *slotVal* = val;
        *notifyAll*();
        return;
    }
} ...
java.lang.Object

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();
    ...
}
```
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 would 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 TwoThreadsDemo FSP have?
- When should you 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?
- How would you model synchronization in FSP?
3. Safety and Synchronization

Overview

- Modelling interaction in FSP
- Safety — synchronizing critical sections
  - Locking for atomicity
  - The busy-wait mutual exclusion protocol
- Conditional synchronization
  - Slots in FSP
  - wait(), notify() and notifyAll()
  - Slots in Java

Selected material © Magee and Kramer
Modelling interaction — shared actions

Actions that are common between two processes are shared and can be used to model process interaction:

- Unshared actions may be arbitrarily interleaved
- Shared actions occur simultaneously for all participants

\[
\text{MAKER} = ( \text{make} \rightarrow \text{ready} \rightarrow \text{MAKER} ). \\
\text{USER} = ( \text{ready} \rightarrow \text{use} \rightarrow \text{USER} ).
\]

\[
\mid \mid \text{MAKER_USER} = ( \text{MAKER} \mid \mid \text{USER} ).
\]

✏️ What are the states of the LTS?
✏️ The traces?
Modelling interaction — handshake

A *handshake* is an action that signals acknowledgement

\[
\text{MAKERv2} = ( \text{make} \to \text{ready} \to \text{used} \to \text{MAKERv2} ).
\]
\[
\text{USERv2} = ( \text{ready} \to \text{use} \to \text{used} \to \text{USERv2} ).
\]
\[
| | \text{MAKER}_{-}\text{USERv2} = ( \text{MAKERv2} | | \text{USERv2} ).
\]

What are the states and traces of the LTS?
Modelling interaction — multiple processes

Shared actions can be used to *synchronize multiple processes*:

\[
\text{MAKE\textsubscript{A}} = ( \text{makeA} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE\textsubscript{A}} ).
\]

\[
\text{MAKE\textsubscript{B}} = ( \text{makeB} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE\textsubscript{B}} ).
\]

\[
\text{ASSEMBLE} = ( \text{ready} \rightarrow \text{assemble} \rightarrow \text{used} \rightarrow \text{ASSEMBLE} ).
\]

\[
\text{FACTORY} = ( \text{MAKE\textsubscript{A}} \mid \mid \text{MAKE\textsubscript{B}} \mid \mid \text{ASSEMBLE} ).
\]

What are the states and traces of the LTS?
Safety problems

Objects must only be accessed when they are in a consistent state, formalized by a class invariant.

Each method assumes the class invariant holds when it starts, and it re-establishes it when done.

If methods interleave arbitrarily, an inconsistent state may be accessed, and the object may be left in a “dirty” state.

Where shared resources are updated may be a critical section.
Atomicity and interference

Consider the two processes:

\[
\begin{align*}
\{ & x = 0 \} \\
\text{AInc:} & \quad x := x + 1 \\
\text{BInc:} & \quad x := x + 1 \\
\{ & x = ? \}
\end{align*}
\]

How can these processes interfere?
Atomic actions

Individual reads and writes may be atomic actions:

\[
\begin{align*}
\text{const } N &= 3 \\
\text{range } T &= 0..N \\
\text{Var} &= \text{Var}[0], \\
\text{Var}[u:T] &= \{ \text{read}[u] \to \text{Var}[u] \\
&\quad \mid \text{write}[v:T] \to \text{Var}[v] \}. \\
\text{set } \text{VarAlpha} &= \{ \text{read}[T], \text{write}[T] \} \\
\text{Inc} &= \{ \text{read}[v:0..N-1] \\
&\to \text{write}[v+1] \\
&\to \text{STOP} \} + \text{VarAlpha}.
\end{align*}
\]
Sequential behaviour

A single sequential thread requires no synchronization:

(Var || Inc)
**Concurrent behaviour**

*Without synchronization, concurrent threads may interfere:*

\[
\{a,b\}::\text{Var} \mid \mid a:\text{Inc} \mid \mid b:\text{Inc}
\]
Locking

Locks are used to make a critical section atomic:

\[
\text{LOCK} = (\ acquire \rightarrow release \rightarrow \text{LOCK} ).
\]

\[
\text{INC} = (\ acquire \\
\rightarrow \text{read}[v:0..N-1] \\
\rightarrow \text{write}[v+1] \\
\rightarrow \text{release} \\
\rightarrow \text{STOP} \ ) + \text{VarAlpha}.
\]
Synchronization

Processes can synchronize critical sections by sharing a lock:

\[
\{a, b\}::\text{VAR} \mid \{a, b\}::\text{LOCK} \mid a::\text{INC} \mid b::\text{INC}
\]
Synchronization in Java

Java Threads also synchronize using locks:

```java
synchronized T m() {
    // method body
}
```

is just convenient syntax for:

```java
T m() {
    synchronized (this) {
        // method body
    }
}
```

Every object has a lock, and Threads may use them to synchronize with each other.
Busy-Wait Mutual Exclusion Protocol

P1 sets $\text{enter1} := \text{true}$ when it wants to enter its CS, but sets $\text{turn} := \text{"P2"}$ to yield priority to P2:

```
process P1
  loop
    $\text{enter1} := \text{true}$
    $\text{turn} := \text{"P2"}$
    while $\text{enter2}$ and $\text{turn} = \text{"P2"}$
      $\text{turn} = \text{"P2"}$
      do skip
    Critical Section
    $\text{enter1} := \text{false}$
  Non-critical Section
  end
end
```

```
process P2
  loop
    $\text{enter2} := \text{true}$
    $\text{turn} := \text{"P1"}$
    while $\text{enter1}$ and $\text{turn} = \text{"P1"}$
      $\text{turn} = \text{"P1"}$
      do skip
    Critical Section
    $\text{enter2} := \text{false}$
  Non-critical Section
  end
end
```

queeze

Is this protocol correct? Is it fair? Deadlock-free?
Atomic read and write

We can model integer and boolean variables as processes with atomic read and write actions:

range T = 1..2

Var = Var[1],
Var[u:T] =
    ( read[u] -> Var[u]

set Bool = {true, false}

BOOL(Init='false) = BOOL[Init],
BOOL[b:Bool] =
    ( is[b] -> BOOL[b]
Modelling the busy-wait protocol

Each process performs two actions in its CS:

\[
P_1 = (\text{enter1}.\text{setTo}[^\text{true}] \\
\quad \rightarrow \text{turn}.\text{write}[2] \\
\quad \rightarrow \text{Gd}1),
\]

\[
\text{Gd}1 = \\
(\text{enter2}.\text{is}[^\text{false}] \rightarrow \text{CS}1 \\
\quad | \quad \text{enter2}.\text{is}[^\text{true}] \rightarrow \\
\quad \quad (\text{turn}.\text{read}[1] \rightarrow \text{CS}1 \\
\quad \quad | \quad \text{turn}.\text{read}[2] \rightarrow \text{Gd}1)),
\]

\[
\text{CS}1 = (\quad \text{a} \rightarrow \text{b} \\
\quad \rightarrow \text{enter1}.\text{setTo}[^\text{false}] \\
\quad \rightarrow \text{P}1).
\]

\[
P_2 = (\text{enter2}.\text{setTo}[^\text{true}] \\
\quad \rightarrow \text{turn}.\text{write}[1] \\
\quad \rightarrow \text{Gd}2),
\]

\[
\text{Gd}2 = \\
(\text{enter1}.\text{is}[^\text{false}] \rightarrow \text{CS}2 \\
\quad | \quad \text{enter1}.\text{is}[^\text{true}] \rightarrow \\
\quad \quad (\text{turn}.\text{read}[2] \rightarrow \text{CS}2 \\
\quad \quad | \quad \text{turn}.\text{read}[1] \rightarrow \text{Gd}2)),
\]

\[
\text{CS}2 = (\quad \text{c} \rightarrow \text{d} \\
\quad \rightarrow \text{enter2}.\text{setTo}[^\text{false}] \\
\quad \rightarrow \text{P}2).
\]

\[
\text{Test} = (\text{enter1}:\text{BOOL} | \text{enter2}:\text{BOOL} | \text{turn}:\text{Var} | \text{P}1 | \text{P}2)@\{\text{a}, \text{b}, \text{c}, \text{d}\}.
\]
Busy-wait composition
Checking for errors

We can check for errors by composing our system with an agent that moves to the `ERROR` state if atomicity is violated:

\[
Ok = ( a \rightarrow ( c \rightarrow \text{ERROR} \mid b \rightarrow \text{Ok} ) \\
| c \rightarrow ( a \rightarrow \text{ERROR} \mid d \rightarrow \text{Ok} ) ).
\]

What happens if we break the protocol?
Conditional synchronization

A lock \textit{delays} an acquire request if it is already locked:

\[
\text{LOCK} = ( \text{acquire} \rightarrow \text{release} \rightarrow \text{LOCK}).
\]

Similarly, a one-slot buffer delays a put request if it is \textit{full} and delays a get request if it is \textit{empty}:

\[
\text{const N = 2} \\
\text{Slot} = ( \text{put[v:0..N]} \\
\quad \rightarrow \text{get[v]} \\
\quad \rightarrow \text{Slot}).
\]
Producer/Consumer composition

Producer = ( put[0]
            -> put[1]
            -> put[2]
            -> Producer).

Consumer = ( get[x:0..N]
            -> Consumer).

||Chain = ( Producer
           ||Slot
           ||Consumer)
Wait and notify

A Java object whose methods are all synchronized behaves like a monitor

Within a synchronized method or block:
- `wait()` suspends the current thread, releasing the lock
- `notify()` wakes up one thread waiting on that object
- `notifyAll()` wakes up all threads waiting on that object

Outside of a synchronized block, `wait()` and `notify()` will raise an `IllegalMonitorStateException`

Always use `notifyAll()` unless you are sure it doesn’t matter which thread you wake up!
Slot (put)

class Slot implements Buffer {
    private Object slotVal;

    public synchronized void put(Object val) {
        while (slotVal != null) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        slotVal = val;
        notifyAll(); // make waiting threads Runnable
        return;
    }

    ...
}
Slot (get)

...  
public synchronized Object get() {  
   Object rval;  
   while (slotVal == null) {  
      try {  
         wait();  
      } catch (InterruptedException e) { }  
   }  
   rval = slotVal;  
   slotVal = null;  
   notifyAll();  
   return rval;  
}
}
Producer in Java

The Producer puts _count messages to the slot:

```java
class Producer extends Thread {
    protected int _count;
    protected Buffer _slot;
    Producer(String name, Buffer slot, int count) {
        super(name);
        _slot = slot;
        _count = count;
    }

    public void run() {
        int i;
        for (i=1;i<=_count;i++) {
            this.action(i);
        }
    }
}

protected void action(int n) {
    String message;
    message = this.getName() + "(" + String.valueOf(n) + ")";
    _slot.put(message);
    System.out.println(getName() + " put " + message);
}
```
Consumer in Java

... and the Consumer gets them:

class Consumer extends Producer {
    // code reuse only!
    Consumer(String name, Buffer slot, int count) {
        super(name, slot, count);
    }
    protected void action(int n) {
        String message;
        message = (String) _slot.get();
        System.out.println(getName() + " got " + message);
    }
}

**Composing Producers and Consumers**

*Multiple* producers and consumers may *share* the buffer:

```java
public static void main(String args[]) {
    Buffer slot = new Slot();
    new Producer("apple ", slot, count).start();
    new Producer("orange", slot, count).start();
    new Producer("banana", slot, count).start();

    new Consumer("asterix", slot, count).start();
    new Consumer("obelix ", slot, 2*count).start();
}
```
What you should know!

✎ How do you model interaction with FSP?
✎ What is a critical section? What is critical about it?
✎ Why don’t sequential programs need synchronization?
✎ How do locks address safety problems?
✎ What primitives do you need to implement the busy-wait mutex protocol?
✎ How can you use FSP to check for safety violations?
✎ What happens if you call wait or notify outside a synchronized method or block?
✎ When is it safe to use notifyAll()?
Can you answer these questions?

- What is an example of an invariant that might be violated by interfering, concurrent threads?
- What constitute atomic actions in Java?
- Can you ensure safety in concurrent programs without using locks?
- When should you use `synchronize(this)` rather than `synchronize(someObject)`?
- Is the busy-wait mutex protocol fair? Deadlock-free?
- How would you implement a `Lock` class in Java?
- Why is the Java Slot class so much more complex than the FSP Slot specification?
4. Safety Patterns

Overview

- Immutability:
  - *avoid* safety problems by avoiding state changes

- Full Synchronization:
  - *dynamically* ensure exclusive access

- Partial Synchronization:
  - restrict synchronization to "critical sections"

- Containment:
  - *structurally* ensure exclusive access
Idioms, Patterns and Architectural Styles

Idioms, patterns and architectural styles express **best practice** in resolving common design problems.

**Idioms**

"an implementation technique"

**Design patterns**

"a commonly-recurring structure of communicating components that solves a general design problem within a particular context"

**Architectural patterns**

"a fundamental structural organization schema for software systems"
**Pattern: Immutable classes**

**Intent:** Bypass safety issues by not changing an object’s state after creation.

**Applicability**
- When objects represent values of simple ADTs
  - colours (java.awt.Color), numbers (java.lang.Integer)
- When classes can be separated into **mutable and immutable versions**
  - java.lang.String vs. java.lang.StringBuffer
- When updating by copying is cheap
  - “hello” + “ ” + “world” → “hello world”
- When **multiple instances** can represent the **same value**
  - i.e., two copies of 712 represent the same integer
Immutability variants

Variants

*Stateless methods*
- methods that do not access an object’s state do not need to be synchronized (can be declared static)
- any temporary state should be local to the method

*Stateless objects*
- an object whose “state” is *dynamically* computed needs no synchronization!

“Hardening”
- object becomes immutable after a mutable phase
- expose to concurrent threads only *after* hardening
Immutable classes — design steps

Declare a class with instance variables that are never changed after construction.

class Relay { // helper for some Server class
    private final Server server_

    Relay(Server s) { // blank finals must be
        server_ = s; // initialized in all
    } // constructors

    void doIt() {
        server_.doIt();
    }
}
Design steps …

- Especially if the class represents an immutable data abstraction (such as `String`), consider **overriding** `Object.equals` and `Object.hashCode`.

- Consider writing **methods that generate new objects** of this class.
  (e.g., `String` concatenation)

- Consider declaring the class as **final**.

- If only some variables are immutable, use synchronization or other techniques for the methods that are not stateless.
Pattern: Fully Synchronized Objects

Intent: Maintain consistency by fully synchronizing all methods. At most one method will run at any point in time.

Applicability

- You want to eliminate all possible read/write and write/write conflicts, regardless of the context in which it the object is used.

- All methods can run to completion without waits, retries, or infinite loops.

- You do not need to use instances in a layered design in which other objects control synchronization of this class.
You can avoid or deal with liveness failures, by:
- Exploiting partial immutability
- Removing synchronization for accessors
- Removing synchronization in invocations
- Arranging per-method concurrency
- ...

Applicability ...
Full Synchronization — design steps

❑ Declare all methods as synchronized

❖ Do not allow any direct access to state (i.e., no public instance variables; no methods that return references to instance variables).

❖ Constructors cannot be marked as synchronized in Java. *Use a synchronized block* in case a constructor passes this to multiple threads.

❖ Methods that access static variables must either do so via static synchronized methods or within blocks of the form synchronized(getClass()) { ... }.
Design steps …

- Ensure that every public method exits leaving the object in a consistent state, even if it exits via an exception.

- Keep methods short so they can atomically run to completion.
Design steps …

- State-dependent actions must rely on *balking*:
  - Return failure (i.e., exception) to client if preconditions fail
  - If the precondition does not depend on state (e.g., just on the arguments), then check outside synchronized code
  - Provide *public accessor methods* so that clients can check conditions before making a request
Example: a BalkingBoundedCounter

```java
public class BalkingBoundedCounter {
    protected long count_ = BoundedCounter.MIN; // between MIN and MAX
    public synchronized long value() { return count_; }
    public synchronized void inc() throws CannotIncrementException {
        if (count_ >= BoundedCounter.MAX) { // if pre fails
            throw new CannotIncrementException(); // throw exception
        } else {
            ++count_;
        }
    }
    public synchronized void dec() { ... } // analogous
}
```

What safety problems could arise if this class were not fully synchronized?
Example: an ExpandableArray

A simplified variant of java.util.Vector:

```java
import java.util.NoSuchElementException;
public class ExpandableArray {
    protected Object[] data_; // the elements
    protected int size_; // the number of slots used
    public ExpandableArray(int cap) {
        data_ = new Object[cap]; // reserve some space
        size_ = 0;
    }
    public synchronized int size() { return size_; }
    public synchronized Object at(int i) // array indexing
        throws NoSuchElementException {
        if (i < 0 || i >= size_ )
            throw new NoSuchElementException();
        else
            return data_[i];
    }
    ...
public synchronized void append(Object x) { // add at end
    if (size_ >= data_.length) { // need a bigger array
        Object[] olddata = data_; // so increase ~50%
        data_ = new Object[3 * (size_ + 1) / 2];
        for (int i = 0; i < size_; ++i)
            data_[i] = olddata[i];
    }
    data_[size_++] = x;
}

public synchronized void removeLast() throws NoSuchElementException {
    if (size_ == 0)
        throw new NoSuchElementException();
    else
        data_[--size_] = null;
}
Bundling Atomicity

Consider adding synchronized methods that perform sequences of actions as a single atomic action

```java
public interface Procedure { // apply an operation to an object
    public void apply(Object x);
}

public class ExpandableArrayV2 extends ExpandableArray {
    public ExpandableArrayV2(int cap) { super(cap); }

    public synchronized void applyToAll(Procedure p) {
        for (int i = 0; i < size_; ++i) {
            p.apply(data_[i]);
        }
    }
}
```

What possible liveness problems does this introduce?
Using inner classes

Use anonymous inner classes to pass procedures:

class ExpandableArrayUser {
    public static void main(String[] args) {
        ExpandableArrayV2 a = new ExpandableArrayV2(100);
        for (int i = 0; i < 100; ++i) // fill it up
            a.append(new Integer(i));
        a.applyToAll(new Procedure () { // print all elements
            public void apply(Object x) {
                System.out.println(x);
            }
        });
    }
}

NB: Any variables shared with the host object must be declared final (immutable).
**Pattern: Partial Synchronization**

**Intent:** Reduce overhead by synchronizing only within “critical sections”.

**Applicability**

- When objects have both mutable and immutable instance variables.

- When methods can be split into a “critical section” that deals with mutable state and a part that does not.
Partial Synchronization — design steps

- Fully synchronize all methods

- Remove synchronization for accessors to atomic or immutable values

- Remove synchronization for methods that access mutable state through a single other, already synchronized method

- Replace method synchronization by block synchronization for methods where access to mutable state is restricted to a single, critical section
Example: LinkedCells

```java
public class LinkedCell {
    protected double value_; // NB: doubles are not atomic!
    protected final LinkedCell next_; // fixed

    public LinkedCell (double val, LinkedCell next) {
        value_ = val; next_ = next;
    }

    public synchronized double value() { return value_; }
    public synchronized void setValue(double v) { value_ = v; }

    public synchronized LinkedCell next() { // not synched!
        return next_; // next_ is immutable
    }

    ...
```
Example ...


```java
public double sum() { // add up all element values
double v = value(); // get via synchronized accessor
if (next() != null)
    v += next().sum();
return v;
}

public boolean includes(double x) { // search for x
    synchronized(this) { // synch to access value
        if (value_ == x) return true;
    }
    if (next() == null) return false;
    else return next().includes(x);
}
```
Pattern: Containment

Intent: Achieve safety by avoiding shared variables. Unsynchronized objects are “contained” inside other objects that have at most one thread active at a time.

Applicability
- There is no need for shared access to the embedded objects.
- The embedded objects can be conceptualized as exclusively held resources.
Applicability …

- Embedded objects must be structured as islands — communication-closed sets of objects reachable only from a single unique reference. They cannot contain methods that reveal their identities to other objects.

- You are willing to hand-check designs for compliance.

- You can deal with or avoid indefinite postponements or deadlocks in cases where host objects must transiently acquire multiple resources.
Contained Objects — design steps

- Define the *interface* for the outer host object.

  The host could be, e.g., an Adaptor, a Composite, or a Proxy, that provides synchronized access to an *existing, unsynchronized class*.

- Ensure that the host is *either* fully synchronized, or is in turn a contained object.
Design steps …

- Define instances variables that are unique references to the contained objects.

- Make sure that these references cannot leak outside the host!

- Establish policies and implementations that ensure that acquired references are really unique!

- Consider methods to duplicate or clone contained objects, to ensure that copies are unique.
Managed Ownership

- Model contained objects as *physical resources*:
  - If you have one, then *you can do something* that you couldn't do otherwise.
  - If you have one, then *no one else has it*.
  - If you give one to someone else, then *you no longer have it*.
  - If you *destroy* one, then *no one will ever have it*. 
Managed Ownership …

- If contained objects can be passed among hosts, define a transfer protocol.

- Hosts should be able to acquire, give, take, exchange and forget resources

- Consider using a dedicated class to manage transfer
A minimal transfer protocol class

A simple buffer for transferring objects between threads:

```java
public class ResourceVariable {
    protected Object ref_; 
    public ResourceVariable(Object res) { ref_ = res; }
    public synchronized Object resource() { return ref_; }
    public synchronized Object exchange(Object r) {
        Object old = ref_; 
        ref_ = r; 
        return old;
    }
}
```

✎ What are the weaknesses of this class?
✎ How would you fix them?
What you should know!

✎ Why are immutable classes inherently safe?
✎ Why doesn’t a “relay” need to be synchronized?
✎ What is “balking”? When should a method balk?
✎ When is partial synchronization better than full synchronization?
✎ How does containment avoid the need for synchronization?
Can you answer these questions?

✎ When is it all right to declare only some methods as synchronized?
✎ When is an inner class better than an explicitly named class?
✎ What could happen if any of the ExpandableArray methods were not synchronized?
✎ What liveness problems can full synchronization introduce?
✎ Why is it a bad idea to have two separate critical sections in a single method?
✎ Does it matter if a contained object is synchronized or not?
5. Liveness and Deadlock

Overview

- Safety revisited
  - ERROR conditions
- Liveness
  - Progress Properties
- Deadlock
  - The Dining Philosophers problem
  - Detecting and avoiding deadlock

Selected material © Magee and Kramer
Safety revisited

A safety property asserts that nothing bad happens

ERROR process (-1) to detect erroneous behaviour

\[
\text{command} \
\begin{array}{c}
\text{-1} \\
\downarrow \quad \uparrow \\
\text{0} \\
\downarrow \\
\text{1}
\end{array}
\]

ACTUATOR
\[
= (\text{command} \rightarrow \text{ACTION}),
\]
ACTION
\[
= (\text{respond} \rightarrow \text{ACTUATOR}
\mid \text{command} \rightarrow \text{ERROR}).
\]

Trace to ERROR: command command
Safety — property specification

ERROR conditions state what is \textit{not} required.

In complex systems, it is usually better to specify directly what \textit{is} required.

Trace to property violation in SAFE_ACTUATOR:

\begin{verbatim}
command command
-1 0 1
\end{verbatim}

property SAFE_ACTUATOR
= (command
  -> respond
  -> SAFE_ACTUATOR
).

Trace to property violation in SAFE_ACTUATOR:

\begin{verbatim}
command command
\end{verbatim}
Safety properties

A safety property $P$ defines a deterministic process that asserts that any trace including actions in the alphabet of $P$ is accepted by $P$.

Transparency of safety properties:

- Since all actions in the alphabet of a property are eligible choices, composing a property with a set of processes does not affect their correct behaviour.
- If a behaviour can occur which violates the safety property, then ERROR is reachable.

Properties must be deterministic to be transparent.
Safety properties

How can we specify that some action, disaster, never occurs?

property CALM = STOP + \{disaster\}.

A safety property must be specified so as to include all the acceptable, valid behaviours in its alphabet.
Liveness

A **liveness property** asserts that something good **eventually** happens.

A **progress property** asserts that it is **always** the case that an action is **eventually** executed.

Progress is the opposite of *starvation*, the name given to a concurrent programming situation in which an action is never executed.
Liveness Problems

A program may be “safe”, yet suffer from various kinds of liveness problems:

Starvation: (AKA “indefinite postponement”)
- The system as a whole makes progress, but some individual processes don’t

Dormancy:
- A waiting process fails to be woken up

Premature termination:
- A process is killed before it should be

Deadlock:
- Two or more processes are blocked, each waiting for resources held by another
Progress properties — fair choice

**Fair Choice:** If a choice over a set of transitions is executed infinitely often, then *every* transition in the set will be executed infinitely often.

If a coin were tossed an infinite number of times, we would expect that both heads and tails would each be chosen infinitely often.

*This assumes fair choice!*

\[
\text{COIN} = (\text{toss} \rightarrow \text{heads} \rightarrow \text{COIN} \mid \text{toss} \rightarrow \text{tails} \rightarrow \text{COIN}).
\]
Progress properties

\[ \text{progress } P = \{a_1, a_2 \ldots a_n\} \]

asserts that in an infinite execution of a target system, \textit{at least one} of the actions \(a_1, a_2 \ldots a_n\) will be executed \textit{infinitely often}.

COIN system:

\[ \text{progress HEADS } = \{\text{heads}\} \]
\[ \text{progress TAILS } = \{\text{tails}\} \]

\[ \ldots \]

No progress violations detected.
Progress properties

Suppose we have both a normal coin and a *trick coin*

\[
\text{TWO\text{COIN} } = (\text{pick} \rightarrow \text{COIN} \mid \text{pick} \rightarrow \text{TRICK}), \\
\text{TRICK } = (\text{toss} \rightarrow \text{heads} \rightarrow \text{TRICK}), \\
\text{COIN } = (\text{toss} \rightarrow \text{heads} \rightarrow \text{COIN} \mid \text{toss} \rightarrow \text{tails} \rightarrow \text{COIN}).
\]

progress \text{HEADS} = \{\text{heads}\}

progress \text{TAILS} = \{\text{tails}\}

progress \text{HEADSorTAILS} = \{\text{heads}, \text{tails}\}
Progress analysis

Progress violation: TAILS
Trace to terminal set of states: pick
Actions in terminal set: \{toss, heads\}

A terminal set of states is one in which every state is mutually reachable but no transitions leads out of the set.

The terminal set \{1, 2\} violates progress property TAILS
Deadlock

Four necessary and sufficient conditions:

Serially reusable resources: the deadlocked processes share resources under mutual exclusion.

Incremental acquisition: processes hold on to acquired resources while waiting to obtain additional ones.

No pre-emption: once acquired by a process, resources cannot be pre-empted but only released voluntarily.

Wait-for cycle: a cycle of processes exists in which each process holds a resource which its successor in the cycle is waiting to acquire.
Waits-for cycle

Has A awaits B

Has E awaits A

Has B awaits C

Has D awaits E

Has C awaits D
Deadlock analysis - primitive processes

- A deadlocked state is one with *no outgoing transitions*
- In FSP: STOP process

\[
\text{MOVE} = (\text{north} \rightarrow (\text{south} \rightarrow \text{MOVE} | \text{north} \rightarrow \text{STOP})).
\]

Progress violation for actions: \{north, south\}
Trace to terminal set of states: north north
Actions in terminal set: {}

© O. Nierstrasz — U. Berne
The Dining Philosophers Problem

- Philosophers alternate between *thinking* and *eating*.
- A philosopher needs *two forks* to eat.
- No two philosophers may hold the same fork simultaneously.
- There must be *no deadlock* and *no starvation*.
- Want efficient behaviour under absence of contention.
Deadlocked diners

A deadlock occurs if a \textit{waits-for cycle} arises in which each philosopher grabs one fork and waits for the other.
Dining Philosophers, Safety and Liveness

Dining Philosophers illustrate many classical safety and liveness issues:

<table>
<thead>
<tr>
<th>Mutual Exclusion</th>
<th>Each fork can be used by one philosopher at a time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Condition synchronization</strong></td>
<td>A philosopher needs two forks to eat</td>
</tr>
<tr>
<td><strong>Shared variable communication</strong></td>
<td>Philosophers share forks ...</td>
</tr>
<tr>
<td><strong>Message-based communication</strong></td>
<td>... or they can pass forks to each other</td>
</tr>
</tbody>
</table>
## Dining Philosophers ...

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busy-waiting</td>
<td>A philosopher can poll for forks ...</td>
</tr>
<tr>
<td>Blocked waiting</td>
<td>... or can sleep till woken by a neighbour</td>
</tr>
<tr>
<td>Livelock</td>
<td>All philosophers can grab the left fork and busy-wait for the right ...</td>
</tr>
<tr>
<td>Deadlock</td>
<td>... or grab the left one and wait (sleep) for the right</td>
</tr>
<tr>
<td>Starvation</td>
<td>A philosopher may starve if the left and right neighbours are always faster at grabbing the forks</td>
</tr>
</tbody>
</table>
Modeling Dining Philosophers

PHIL = ( sitdown
  -> right.get -> left.get -> eat
  -> left.put -> right.put
  -> arise -> PHIL).

FORK = ( get -> put -> FORK ).

||DINERS(N=5)=
  forall [i:0..N-1]
  (phil[i]:PHIL
   ||{phil[i].left,phil[((i-1)+N)%N].right}::FORK).

⚠️ Is this system safe? Is it live?
Dining Philosophers Analysis

Trace to terminal set of states:
- phil.0.sitdown
- phil.0.right.get
- phil.1.sitdown
- phil.1.right.get
- phil.2.sitdown
- phil.2.right.get
- phil.3.sitdown
- phil.3.right.get
- phil.4.sitdown
- phil.4.right.get

Actions in terminal set: {}

No further progress is possible due to the waits-for cycle
Eliminating Deadlock

There are two fundamentally different approaches to eliminating deadlock.

Deadlock detection:

❑ *Repeatedly check for waits-for cycles.* When detected, choose a victim and force it to release its resources.

❖ Common in transactional systems; the victim should “roll-back” and try again

Deadlock avoidance:

❑ Design the system so that *a waits-for cycle cannot possibly arise.*
Dining Philosopher Solutions

There are countless solutions to the Dining Philosophers problem that use various concurrent programming styles and patterns, and offer varying degrees of liveness guarantees:

Number the forks
- Philosophers grab the lowest numbered fork first.

Philosophers queue to sit down
- allow no more than four at a time to sit

Do these solutions avoid deadlock?
What about starvation?
Are they “fair”?
What you should know!

- What are safety properties? How are they modelled in FSP?
- What kinds of liveness problems can occur in concurrent programs?
- Why is progress a liveness rather than a safety issue?
- What is fair choice? Why do we need it?
- What is a terminal set of states?
- What are necessary and sufficient conditions for deadlock?
- How can you detect deadlock? How can you avoid it?
Can you answer these questions?

- How would you manually check a safety property?
- Why must safety properties be deterministic to be transparent?
- How would you manually check a progress property?
- What is the difference between starvation and deadlock?
- How would you manually detect a waits-for cycle?
- What is fairness?
6. Liveness and Guarded Methods

Overview

- Guarded Methods
  - Checking guard conditions
  - Handling interrupts
  - Structuring notification
    - Encapsulating assignment
    - Tracking state
    - Tracking state variables
    - Delegating notifications
Achieving Liveness

There are various strategies and techniques to ensure liveness:

- Start with safe design and selectively remove synchronization
- Start with live design and selectively add safety
- Adopt design patterns that limit the need for synchronization
- Adopt standard architectures that avoid cyclic dependencies
Pattern: Guarded Methods

Intent: Temporarily suspend an incoming thread when an object is not in the right state to fulfil a request, and wait for the state to change rather than balking (raising an exception).
Guarded Methods — applicability

- Clients can *tolerate indefinite postponement*. (Otherwise, use a *balking design.*)

- You can guarantee that the *required states are eventually reached* (via other requests), or if not, that it is acceptable to block forever.

- You can arrange that *notifications occur after all relevant state changes*. (Otherwise consider a design based on a *busy-wait spin loop.*)

- You can *avoid* or cope with liveness problems due to waiting threads retaining all synchronization locks.
Applicability …

- You can *construct computable predicates* describing the state in which actions will succeed. (Otherwise consider an *optimistic design*.)

- Conditions and actions are managed *within a single object*. (Otherwise consider a *transactional form*.)
Guarded Methods — design steps

The basic recipe is to use wait in a conditional loop to block until it is safe to proceed, and use notifyAll to wake up blocked threads.

```java
public synchronized Object service() {
    while (wrong State) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    // fill request and change state ...
    notifyAll();
    return result;
}
```
Step: Separate interface from policy

- Define *interfaces* for the methods, so that classes can implement guarded methods according to different *policies*.

```java
public interface BoundedCounter {
    public static final long MIN = 0; // min value
    public static final long MAX = 10; // max value
    public long value(); // inv’t: MIN <= value() <= MAX
                           // init: value() == MIN
    public void inc();    // pre: value() < MAX
    public void dec();    // pre: value() > MIN
}
```
Step: Check guard conditions

- Define a *predicate* that precisely describes the conditions under which actions may proceed. (This can be encapsulated as a helper method.)

- Precede the conditional actions with a *guarded wait loop* of the form:

  ```java
  while (!condition)
      try { wait(); }
      catch (InterruptedException ex) { ... }
  ```

  Optionally, encapsulate this code as a helper method.
Step: Check guard conditions ...

- If there is only one possible condition to check in this class (and all plausible subclasses), and notifications are issued only when the condition is true, then there is no need to re-check the condition after returning from `wait()`

- Ensure that the object is in a consistent state (i.e., the class invariant holds) before entering any `wait` (since `wait` releases the synchronization lock). The easiest way to do this is to perform the guards before taking any actions.
Step: Handle interrupts

- Establish a **policy** to deal with `InterruptedException`s. Possibilities include:
  - **Ignore interrupts** (i.e., an empty `catch` clause), which preserves safety at the possible expense of liveness.
  - **Terminate** the current thread (`stop`). This preserves safety, though brutally! (**Not recommended.**)
  - **Exit** the method, possibly raising an exception. This preserves liveness but may require the caller to take special action to preserve safety.
  - **Cleanup and restart.**
  - **Ask for user intervention** before proceeding.

Interrupts can be useful to signal that the guard can never become true because, for example, the collaborating threads have terminated.
Step: Signal state changes

- Add notification code to each method of the class that changes state in any way that can affect the value of a guard condition. Some options are:

  - use notifyAll to wake up all threads that are blocked in waits for the host object.

...
Notify() vs notifyall()

... use notify to wake up only one thread (if any exist). This is best treated as an optimization where:

- all blocked threads are necessarily waiting for conditions signalled by the same notifications,
- only one of them can be enabled by any given notification, and
- it does not matter which one of them becomes enabled.

You build your own special-purpose notification methods using notify and notifyAll. (For example, to selectively notify threads, or to provide certain fairness guarantees.)
Step: Structure notifications

- Ensure that *each wait is balanced by at least one notification*. Options include:

| Blanket Notifications | Place a *notification at the end of every method* that can cause any state change (i.e., assigns any instance variable). Simple and reliable, but may cause performance problems ...
| Encapsulating Assignment | *Encapsulate assignment* to each variable mentioned in any guard condition *in a helper method* that performs the notification after updating the variable. |
| **Tracking State** | Only issue notifications for the *particular state changes* that could actually unblock waiting threads. May improve performance, at the cost of flexibility (i.e., subclassing becomes harder.) |
| **Tracking State Variables** | Maintain an *instance variable that represents control state*. Whenever the object changes state, invoke a helper method that re-evaluates the control state and will issue notifications if guard conditions are affected. |
| **Delegating Notifications** | Use *helper objects to maintain aspects of state* and have these helpers issue the notifications. |
Encapsulating assignment

Guards and assignments are encapsulated in helper methods:

```java
public class BoundedCounterV1 implements BoundedCounter {
    protected long count_ = MIN;
    public synchronized long value() { return count_; }
    public synchronized void inc() {
        awaitIncrementable();
        setCount(count_ + 1);
    }
    public synchronized void dec() {
        awaitDecrementable();
        setCount(count_ - 1);
    }
}
```
protected synchronized void awaitIncrementable() {
    while (count_ >= MAX)
    try {
        wait();
    } catch (InterruptedException ex) {
    }
}

protected synchronized void awaitDecrementable() {
    while (count_ <= MIN)
    try {
        wait();
    } catch (InterruptedException ex) {
    }
}

protected synchronized void setCount(long newValue) {
    count_ = newValue;
    notifyAll();
}
Tracking State

The only transitions that can possibly affect waiting threads are those that step away from logical states top and bottom:

```java
public class BoundedCounterVST
    implements BoundedCounter {
    protected long count_ = MIN; // ...
    public synchronized void inc() {
        while (count_ == MAX)
            try { wait(); } catch(InterruptedException ex) {};
        if (count_++ == MIN)
            notifyAll(); // just left bottom state
    }
    ...
}
```
Tracking State Variables

public class BoundedCounterVSV implements BoundedCounter {
    static final int BOTTOM = 0; // logical states
    static final int MIDDLE = 1;
    static final int TOP = 2;
    protected int state_ = BOTTOM; // state variable
    protected long count_ = MIN;
    public synchronized void inc() {
        while (state_ == TOP) // consult logical state
            try { wait(); }
            catch(InterruptedException ex) {};
        ++count_; // modify actual state
        checkState(); // sync logical state
    ...
}
...  
public synchronized void dec() { ... }  
public synchronized long value() { return count_; }  

protected synchronized void checkState() {  
    int oldState = state_;  
    if (count_ == MIN) state_ = BOTTOM;  
    else if (count_ == MAX) state_ = TOP;  
    else state_ = MIDDLE;  
    if (state_ != oldState  
        && (oldState == TOP  
            || oldState == BOTTOM))  
        notifyAll();  
  }  
}
Delegating notifications

```java
public class NotifyingLong {
    private long value_; 
    private Object observer_; 
    public NotifyingLong(Object o, long v) {
        observer_ = o; value_ = v;
    }
    public synchronized long value() { return value_; }
    public void setValue(long v) {
        synchronized(this) { value_ = v; }
        synchronized(observer_) {
            observer_.notifyAll(); // NB: must be synched!
        }
    }
}
```
Delegating notifications ...

Notification is delegated to the helper object:

```java
public class BoundedCounterVNL implements BoundedCounter {
    private NotifyingLong c_ =
        new NotifyingLong(this, MIN);
    public synchronized void inc() {
        while (c_.value() >= MAX)
            try { wait(); }
            catch(InterruptedException ex) {};
        c_.setValue(c_.value()+1);
    }
    ...
}
```
What you should know!

✎ When can you apply the *Guarded Methods* pattern?
✎ When should methods *recheck guard conditions after waking from a* `wait()`?
✎ Why should you usually *prefer* `notifyAll()` *to* `notify()`?
✎ When and where should you issue *notification*?
✎ Why must you *re-establish the class invariant before calling* `wait()`?
✎ What should you do when you receive an `InterruptedException`?
✎ What is the difference between *tracking state* and using *state-tracking variables*?
Can you answer these questions?

✎ When are **guarded methods** better than **balking**?
✎ When should you use **helper methods** to implement guarded methods?
✎ What is the best way to **structure guarded methods** for a class if you would like it to be easy for others to define correctly functioning **subclasses**?
✎ When is the complexity of **delegating notifications** worthwhile?
7. **Lab session I**

The lab exercises will be available on the course web page:

www.iam.unibe.ch/~scg/Teaching/CP/
8. Liveness and Asynchrony

Overview

- Asynchronous invocations
  - Simple Relays
    - Direct Invocations
    - Thread-based messages; Gateways
  - Command-based messages
  - Tail calls
  - Early replies
  - Futures
Pattern: Asynchronous Invocations

Intent: Avoid waiting for a request to be serviced by decoupling sending from receiving.

Applicability

❑ When a host object can distribute services amongst multiple helper objects.
❑ When an object does not immediately need the result of an invocation to continue doing useful work.
❑ When invocations that are logically asynchronous, regardless of whether they are coded using threads.
❑ During refactoring, when classes and methods are split in order to increase concurrency and reduce liveness problems.
Asynchronous Invocations — form

Asynchronous invocation typically looks like this:

class Host {
    public service() {
        pre(); // code to run before invocation
        invokeHelper(); // the invocation
        during(); // code to run in parallel
        post(); // code to run after completion
    }
}
Asynchronous Invocations — design steps

Consider the following issues:

<p>| Does the Host need results back from the Helper? | Not if, e.g., the Helper returns results directly to the Host’s caller! |
| Can the Host process new requests while the Helper is running? | Might depend on the kind of request ... |
| Can the Host do something while the Helper is running? | i.e., in the during() code |
| Does the Host need to synchronize pre-invocation processing? | i.e., if service() is guarded or if pre() updates the Host’s state |</p>
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does the Host need to synchronize post-invocation processing?</td>
<td>i.e., if <code>post()</code> updates the Host’s state</td>
</tr>
<tr>
<td>Does post-invocation processing only depend on the Helper’s result?</td>
<td>... or does the host have to wait for other conditions?</td>
</tr>
<tr>
<td>Is the same Helper always used?</td>
<td>Is a new one generated to help with each new service request?</td>
</tr>
</tbody>
</table>

Liveness and Asynchrony
Simple Relays — three variants

A relay method obtains all its functionality by delegating to the helper, without any pre(), during(), or post() actions.

Direct invocations: Invoke the Helper directly, but without synchronization

Thread-based messages: Create a new thread to invoke the Helper

Command-based messages: Pass the request to another object that will run it

Relays are commonly seen in Adaptors.
Variant: Direct invocations

Asynchrony is achieved by avoiding synchronization.

class Host {
    protected Helper helper_ = new Helper();
    public void service() { // unsynchronized!
        invokeHelper(); // (stateless method)
    }
    protected void invokeHelper() {
        helper_.help(); // unsynchronized!
    }
}

The Host is free to accept other requests, while the Host's caller must wait for the reply.
Direct invocations ...

If helper_ is mutable, it can be protected with an accessor:

class Host2 extends Host {
    protected Helper helper_ = new Helper();
    protected synchronized Helper helper() {
        return helper_;  
    }
    public void service() { // unsynchronized
        helper().help(); // partially synchronized
    }
}
Variant: Thread-based messages

The invocation can be performed *within a new thread:*

```java
protected void invokeHelper() {
    new Thread() {
        final Helper h_ = helper_; // Must be final!
        public void run() { h_.help(); }
    }.start();
}
```
Thread-based messages …

The cost of evaluating Helper.help() should outweigh the overhead of creating a thread!

- If the Helper is a *daemon* (loops endlessly)
- If the Helper does *I/O*
- Possibly, if *multiple* helper methods are invoked
Thread-per-message Gateways

The Host may construct a new Helper to service each request.

```java
public class FileIO {
    public void writeBytes(String file, byte[] data) {
        new Thread(new FileWriter(file, data)).start();
    }
    public void readBytes(...) { ... }
}
class FileWriter implements Runnable {
    private String nm_; // hold arguments
    private byte[] d_;
    public FileWriter(String name, byte[] data) { ... }
    public void run() { ... } // write to file ...
}
```
Variant: Command-based messages

The Host can also put a Command object in a queue for another object that will invoke the Helper:

```java
protected EventQueue q_;  
protected invokeHelper() {
   q_.put(new HelperMessage(helper_));
}
```

Command-based forms are especially useful for:
- scheduling of helpers
- undo and replay capabilities
- transporting messages over networks
Tail calls

Applies when the helper method is the last statement of a method. Only pre() code is synchronized.

class Subject {
    protected Observer obs_ = new ...;
    protected double state_;
    public void updateState(double d) {
        // not synched
        doUpdate(d);
        // synched
        sendNotification();
        // not synched
    }
    protected synchronized doUpdate(double d) {
        // synched
        state_ = d;
    }
    protected void sendNotification() {
        // not synched
        obs_.changeNotification(this);
    }
}

The host is immediately available to accept new requests
Tail calls with new threads

Alternatively, the tail call may be made in a separate thread:

```java
public synchronized void updateState(double d) {
    state_ = d;
    new Thread() {
        final Observer o_ = obs_
        public void run() {
            o_.changeNotification(Subject.this);
        }
    }.start();
}
```
Early Reply

Early reply allows a host to perform useful activities after returning a result to the client:

- Early reply is a built-in feature in some programming languages.
- It can be easily simulated when it is not a built-in feature.
Simulating Early Reply

A one-slot buffer can be used to pick up the reply from a helper thread:

A one-slot buffer is a simple abstraction that can be used to implement many higher-level concurrency abstractions ...
Early Reply in Java

```java
public class Host {
    ...

    public Object service() { // unsynchronized
        final Slot reply = new Slot();
        final Host host = this;
        new Thread() {
            // Helper
            public void run() {
                synchronized (host) {
                    reply.put(host.compute());
                    host.cleanup(); // retain lock
                }
            }
        }.start();
        return reply.get(); // early reply
    }
    ...
}
```
Futures

Futures allow a client to continue in parallel with a host until the future value is needed:

```
Client

put() value()
```

```
Host

service()

new

Future

returns future

value()

returns value

put()
A Future Class

Futures can be implemented as a layer of abstraction around a shared Slot:

```java
class Future {
    private Object val_;  // initially null
    private Slot slot_;   // shared with some worker
    public Future(Slot slot) {
        slot_ = slot;
    }
    public Object value() {
        if (val_ == null)
            val_ = slot_.get();
        return val_;
    }
}
```
Using Futures in Java

Without special language support, the client must explicitly request a value() from the future object.

```java
public Future service () {
    // unsynchronized
    final Slot slot = new Slot();
    new Thread() {
        public void run() {
            slot.put(compute());
        }
    }.start();
    return new Future(slot);
}

protected synchronized Object compute() { ... }
```
What you should know!

- What general form does an asynchronous invocation take?
- When should you consider using asynchronous invocations?
- In what sense can a direct invocation be “asynchronous”?
- Why (and how) would you use inner classes to implement asynchrony?
- What is “early reply”, and when would you use it?
- What are “futures”, and when would you use them?
- How can implement futures and early replies in Java?
Can you answer these questions?

✎ Why might you want to increase concurrency on a single-processor machine?
✎ Why are servers commonly structured as thread-per-message gateways?
✎ Which of the concurrency abstractions we have discussed till now can be implemented using one-slot-buffers as the only synchronized objects?
✎ When are futures better than early replies? Vice versa?
9. Condition Objects

Overview

- Condition Objects
  - Simple Condition Objects
  - The “Nested Monitor Problem”
  - Permits and Semaphores
  - Using Semaphores
Pattern: Condition Objects

Intent: Condition objects encapsulate the waits and notifications used in guarded methods.

Applicability

- To simplify class design by off-loading waiting and notification mechanics.
- Because of the limitations surrounding the use of condition objects in Java, in some cases the use of condition objects will increase rather than decrease design complexity!
Condition Objects — applicability

- As an efficiency manoeuvre.
  By isolating conditions, you can often avoid notifying waiting threads that could not possibly proceed given a particular state change.

- As a means of encapsulating special scheduling policies surrounding notifications, for example to impose fairness or prioritization policies.

- In the particular cases where conditions take the form of "permits" or "latches".
Condition Objects

Condition objects implement this interface:

```java
public interface Condition {
    public void await(); // wait for some condition
    public void signal(); // signal that condition
}
```

A client that awaits a condition blocks until another object signals that the condition now *may* hold.
A Simple Condition Object

We can encapsulate guard conditions with this class:

```java
public class SimpleConditionObject
    implements Condition
{
    public synchronized void await()
    {
        try {
            wait();
        }
        catch (InterruptedException ex) {}
    }
    public synchronized void signal()
    {
        notifyAll();
    }
}
```

Careless use can lead to the “Nested Monitor Problem”
The Nested Monitor problem

We want to avoid waking up the wrong threads by separately notifying the conditions notMin and notMax:

```java
public class BoundedCounterVBAD implements BoundedCounter {
    protected long count_ = MIN;
    protected Condition notMin_ = new SimpleConditionObject();
    protected Condition notMax_ = new SimpleConditionObject();
    public synchronized long value() {
        return count_;  
    }
    ...
```
The Nested Monitor problem ...

```java
public synchronized void dec() {
    while (count_ == MIN)
        notMin_.await(); // wait till count not MIN
    if (count_-- == MAX)
        notMax_.signal();
}
public synchronized void inc() { // can’t get in!
    while (count_ == MAX)
        notMax_.await();
    if (count_++ == MIN)
        notMin_.signal(); // we never get here!
}
```
The Nested Monitor problem ...

Nested monitor lockouts occur whenever a blocked thread holds the lock for an object containing the method that would otherwise provide a notification to unblock the wait.
Nested Monitors in FSP

Nested Monitors typically arise when one synchronized object is implemented using another.

Recall our one Slot buffer in FSP:

\[
\text{const } N = 2 \\
\text{Slot} = (\text{put}[v:0..N] \rightarrow \text{get}[v] \rightarrow \text{Slot}).
\]

Suppose we try to implement a call/reply protocol using a private instance of Slot:

\[
\text{ReplySlot} = \\
(\text{put}[v:0..N] \rightarrow \text{my.put}[v] \rightarrow \text{ack} \rightarrow \text{ReplySlot} \\
| \text{get} \rightarrow \text{my.get}[v] \rightarrow \text{ret}[v] \rightarrow \text{ReplySlot}).
\]
Nested Monitors in FSP …

Our producer/consumer chain obeys the new protocol:

Producer = ( put[0] -> ack
            -> put[1] -> ack

Consumer = ( get-> ret[x:0..N] -> Consumer ).

||Chain = (Producer||ReplySlot||my:Slot||Consumer).
Nested Monitors in FSP ...

But now the chain may deadlock:

Progress violation for actions: \{put.0, ack, put.1, put.2, my.put.0, my.put.1, my.put.2, get, my.get.2, ret.2\}

Trace to terminal set of states:
  get
  ret.0

Actions in terminal set: {}
Solving the Nested Monitors problem

You must ensure that:

- **Waits** do not occur while *synchronization* is held on the *host* object.
  - This leads to a guard loop that reverses the synchronization seen in the faulty version.

- **Notifications** are never missed.
  - The entire guard wait loop should be enclosed within *synchronized* blocks on the *condition* object.

...
Solving Nested Monitors ...

... 

- **Notifications** do not **deadlock**.
  - All **notifications** should be performed **only upon release of all synchronization** (except for the notified condition object).

- Helper and host state must be **consistent**.
  - If the helper object maintains any state, it must always be consistent with that of the host, and if it **shares** any state with the host, that access is properly **synchronized**.
Example solution

```java
public class BoundedCounterVCV implements BoundedCounter {
    public void dec() {
        // not synched!
        boolean wasMax = false;
        // record notification condition
        synchronized(notMin_) {
            // synch on condition object
            while (true) {
                synchronized(this) {
                    if (count_ > MIN) {
                        // check and act
                        wasMax = (count_ == MAX);
                        count_--;
                        break;
                    }
                }
            }
            notMin_.await();
            // release host synch before wait
        }
        if (wasMax) notMax_.signal();
        // first release all synchs!
    }
}
```
**Pattern: Permits and Semaphores**

**Intent:** Bundle synchronization in a condition object when synchronization depends on the value of a counter.

**Applicability**

- When any given `await` may proceed only if there have been *more signals than awaits*.
  - I.e., when `await` decrements and `signal` increments the number of available “permits”.

- You need to guarantee the *absence of missed signals*.
  - Unlike simple condition objects, semaphores work even if one thread enters its `await` after another thread has signalled that it may proceed.

- The host classes can arrange to invoke `Condition` methods *outside* of synchronized code.
Define a class implementing `Condition` that maintains a permit count, and *immediately* releases `await` if there are already enough permits.

* e.g., `BoundedCounter`

```java
public class CountCondition implements Condition {
    protected BoundedCounter counter_ = new BoundedCounterV0();
    public void await() { counter_.dec(); }
    public void signal() { counter_.inc(); }
}
```
Design steps ...

- As with all kinds of condition objects, their clients must avoid invoking `await` inside of synchronized code.
- You can use a *before/after design* of the form:

```java
class Host {
    Condition aCondition_; ...
    public method m1() {
        aCondition_.await();    // not synched
        doM1();                 // synched
        for each Condition c enabled by m1()
            c.signal();         // not synched
    }
    protected synchronized doM1() { ... }
}
```
Variants

Permit Counters: (Counting Semaphores)
- Just keep track of the number of “permits”
- Can use notify instead of notifyAll if class is final

Fair Semaphores:
- Maintain FIFO queue of threads waiting on a SimpleCondition

Locks and Latches:
- Locks can be acquired and released in separate methods
- Keep track of thread holding the lock so locks can be reentrant!
- A latch is set to true by signal, and always stays true

See the On-line supplement for details!
Semaphores in Java

```java
public class Semaphore { // simple version
    private int value;
    public Semaphore (int initial) { value = initial; }
    synchronized public void up() { // AKA V
        ++value;
        notify(); // wake up just one thread!
    }
    synchronized public void down() { // AKA P
        while (value==0)
            try { wait(); } catch(InterruptedException ex) { };
        --value;
    }
}
```
Using Semaphores

public class BoundedCounterVSem
    implements BoundedCounter {
    protected long count_ = MIN;
    protected Semaphore mutex;
    protected Semaphore full;  // number of items
    protected Semaphore empty;  // number of slots

    BoundedCounterVSem() {
        mutex = new Semaphore(1);
        full = new Semaphore(0);
        empty = new Semaphore(MAX-MIN);
    }

    ...
Using Semaphores ...

```java
public long value() {
    mutex.down(); // grab the resource
    long val = count_;
    mutex.up(); // release it
    return val;
}

public void inc() {
    empty.down(); // grab a slot
    mutex.down(); // sequence is important!
    count_ ++;
    mutex.up();
    full.up(); // release an item
}
...```

...
Using Semaphores ...

These would cause a nested monitor problem!

```java
public void BADinc() {
    mutex.down(); empty.down(); // locks out BADdec!
    count_ ++;
    full.up(); mutex.up();
}

public void BADdec() {
    mutex.down(); full.down(); // locks out BADinc!
    count_ --;
    empty.up(); mutex.up();
}
```
What you should know!

What are “condition objects”? How can they make your life easier? Harder?
What is the “nested monitor problem”?
How can you avoid nested monitor problems?
What are “permits” and “latches”? When is it natural to use them?
How does a semaphore differ from a simple condition object?
Why (when) can semaphores use notify() instead of notifyAll()?
Can you answer these questions?

✎ Why doesn’t SimpleConditionObject need any instance variables?
✎ What is the easiest way to avoid the nested monitor problem?
✎ What assumptions do nested monitors violate?
✎ How can the obvious implementation of semaphores (in Java) violate fairness?
✎ How would you implement fair semaphores?
10. Fairness and Optimism

- Concurrently available methods
  - Priority
  - Interception
  - Readers and Writers

- Optimistic methods

Selected material © Magee and Kramer
Pattern: Concurrency Available Methods

Intent: Non-interfering methods are made concurrently available by implementing policies to enable and disable methods based on the current state and running methods.

Applicability

- Host objects are accessed by many different threads.
- Host services are not completely interdependent, so need not be performed under mutual exclusion.
- You need to improve throughput for some methods by eliminating nonessential blocking.
- You want to prevent various accidental or malicious starvation due to some client forever holding its lock.
- Full synchronization would needlessly make host objects prone to deadlock or other liveness problems.
Concurrent Methods — design steps

Layer concurrency control policy over mechanism by:

Policy Definition:
- When may methods run concurrently?
- What happens when a disabled method is invoked?
- What priority is assigned to waiting tasks?

Instrumentation:
- Define state variables to detect and enforce policy.

Interception:
- Have the host object intercept public messages and then relay them under the appropriate conditions to protected methods that actually perform the actions.
Priority

Priority may depend on any of:

- **Intrinsic attributes** of tasks (class & instance variables).
- **Representations** of task *priority, cost, price, or urgency*.
- The **number** of tasks **waiting** for some condition.
- The **time** at which each task is added to a queue.
- **Fairness** — guarantees that each waiting task will eventually run.
- **Expected duration** or time to completion of each task.
- The **desired completion time** of each task.
- Termination **dependencies** among tasks.
- The **number** of tasks that have **completed**.
- The **current time**.
Fairness

There are subtle differences between definitions of fairness:

**Weak fairness:** If a process *continuously* makes a request, *eventually* it will be granted.

**Strong fairness:** If a process makes a request *infinitely often*, *eventually* it will be granted.

**Linear waiting:** If a process makes a request, it will be granted *before* any other process is granted the request *more than once*.

**FIFO (first-in first out):** If a process makes a request, it will be *granted before* that of any process making a *later* request.
Interception

Interception strategies include:

Pass-Throughs: The host maintains a set of immutable references to helper objects and simply relays all messages to them within unsynchronized methods.

Lock-Splitting: Instead of splitting the class, split the synchronization locks associated with subsets of the state.

Before/After methods: Public methods contain before/after processing surrounding calls to non-public methods in the host that perform the services.
Concurrent Reader and Writers

“Readers and Writers” is a family of concurrency control designs in which “Readers” (non-mutating accessors) may concurrently access resources while “Writers” (mutative, state-changing operations) require exclusive access.
Readers/Writers Model

We are interested only in capturing who gets access:

set Actions = \{ acquireRead, releaseRead, 
acquireWrite, releaseWrite \}

READER = ( acquireRead
-> examine
-> releaseRead -> READER )
+Actions \{examine\}.

WRITER = ( acquireWrite
-> modify
-> releaseWrite -> WRITER )
+Actions \{modify\}.
A Simple RW Protocol

```plaintext
const Nread = 2     // Maximum readers
const Nwrite = 2    // Maximum writers

RW_LOCK = RW[0][False],
RW[readers:0..Nread][writing:Bool] =
  ( when (!writing)
    acquireRead     -> RW[readers+1][writing]
    releaseRead    -> RW[readers-1][writing]
    when (readers==0 && !writing)
    acquireWrite   -> RW[readers][True]
    releaseWrite   -> RW[readers][False]
  ).
```
Safety properties

We specify the safe interactions:

property SAFE_RW =
  ( acquireRead -> READING[1]
  | acquireWrite -> WRITING ),
READING[i:1..Nread] =
  ( acquireRead -> READING[i+1]
  | when(i>1) releaseRead -> READING[i-1]
  | when(i==1) releaseRead -> SAFE_RW
  ),
WRITING = ( releaseWrite -> SAFE_RW ).
Safety properties ...

And compose them with RW_LOCK:

\[ \text{READWRITELOCK} = (\text{RW\_LOCK} \ || \ \text{SAFE\_RW}) \].
**Composing the Readers and Writers**

We compose the READERS and WRITERS with the protocol and check for safety violations:

\[
\| \| \text{READERS}_\text{WRITERS} = \\
( \text{reader}[1..N\text{read}]:\text{READER} \\
| | \text{writer}[1..N\text{write}]:\text{WRITER} \\
| | \{\text{reader}[1..N\text{read}], \\
\quad \text{writer}[1..N\text{write}]\}::\text{READWRITELOCK}).
\]

No deadlocks/errors
**Progress properties**

We similarly specify liveness properties:

\[
\|
\|
RW\_PROGRESS = READERS\_WRITERS
\]

\[
\gg\{\text{reader}[1..Nread].releaseRead, \\
\text{writer}[1..Nread].releaseWrite\}.
\]

progress WRITE\[i:1..Nwrite\] = writer\[i\].acquireWrite

progress READ\[i:1..Nwrite\] = reader\[i\].acquireRead

Progress violation: WRITE.1 WRITE.2

Trace to terminal set of states:

- reader.1.acquireRead tau

Actions in terminal set:

- \{reader.1.acquireRead, reader.1.releaseReadRead, \\
reader.2.acquireRead, reader.2.releaseRead\}
Readers and Writers Policies

Individual policies must address:

- Can new Readers join already active Readers even if a Writer is waiting?
  - if yes, Writers may starve
  - if not, the throughput of Readers decreases

- If both Readers and Writers are waiting for a Writer to finish, which should you let in first?
  - Similar choices exist after Readers finish.

- Can Readers upgrade to Writers without having to give up access?
Policies …

A typical set of choices:

- **Block incoming Readers** if there are waiting Writers.

- “Randomly” choose among incoming threads (i.e., let the scheduler choose).

- **No upgrade** mechanisms.

*Before/after methods* are the simplest way to implement Readers and Writers policies.
Readers and Writers example

Implement state tracking variables

```java
public abstract class RWVT {
    protected int activeReaders_ = 0; // zero or more
    protected int activeWriters_ = 0; // zero or one
    protected int waitingReaders_ = 0;
    protected int waitingWriters_ = 0;

    protected abstract void read_(); // define in subclass
    protected abstract void write_(); // define in subclass
}
```
Readers and Writers example

Public methods call protected before/after methods

... 

public void read() {  // unsynchronized
    beforeRead();  // obtain access
    read_();  // perform service
    afterRead();  // release access
}

public void write() {
    beforeWrite();
    write_();
    afterWrite();
}

...
Readers and Writers example

*Synchronized before/after methods maintain state variables*

...  
protected synchronized void beforeRead() {
  ++waitingReaders_;  // available to subclasses
  while (!allowReader())
    try {
      wait();
    } catch (InterruptedException ex) {}  
    --waitingReaders_; ++activeReaders_;  
}

protected synchronized void afterRead() {
  --activeReaders_; notifyAll();
}

...
Readers and Writers example

Different policies can use the same state variables ...

... protected boolean allowReader() { // default policy
    return waitingWriters_ == 0 && activeWriters_ == 0;
}
...
Pattern: Optimistic Methods

Intent: Optimistic methods attempt actions, but rollback state in case of interference. After rollback, they either throw failure exceptions or retry the actions.

Applicability

- Clients can tolerate either failure or retries.
- If not, consider using guarded methods.
- You can avoid or cope with livelock.
- You can undo actions performed before failure checks.

Rollback/Recovery: undo effects of each performed action. If messages are sent to other objects, they must be undone with “anti-messages”.

Provisional action: “pretend” to act, delaying commitment until interference is ruled out.
Optimistic Methods — design steps

Collect and encapsulate all mutable state so that it can be tracked as a unit:

- Define an immutable helper class holding values of all instance variables.
- Define a representation class, but make it mutable (allow instance variables to change), and additionally include a version number (or transaction identifier) field or even a sufficiently precise time stamp.
- Embed all instance variables, plus a version number, in the host class, but define commit to take as arguments all assumed values and all new values of these variables.
- Maintain a serialized copy of object state.
- Various combinations of the above ...
Detect failure …

Provide an operation that simultaneously detects version conflicts and performs updates via a method of the form:

```java
class Optimistic {
    // code sketch
    private State currentState_; // immutable values
    synchronized boolean commit(State assumed, State next)
    {
        boolean success = (currentState_ == assumed);
        if (success)
            currentState_ = next;
        return success;
    }
}
```
Detect failure ...

Structure the main actions of each public method as follows:

State assumed = currentState();
State next = ... // compute optimistically
if (!commit(assumed, next))
    rollback();
else
    otherActionsDependingOnNewStateButNotChangingIt();
Handle conflicts ...

Choose and implement a policy for dealing with commit failures:

- **Throw an exception** upon commit failure that tells a client that it may retry.
- **Internally retry** the action until it succeeds.
- **Retry some bounded number of times**, or until a timeout occurs, finally throwing an exception.
- **Pessimistically synchronize selected methods** which should not fail.
Ensure progress ...

Ensure progress in case of internal retries

- **Immediately retrying** may be counterproductive!

- **Yielding** may only be effective if all threads have reasonable priorities and the Java scheduler at least approximates fair choice among waiting tasks (which it is not guaranteed to do)!

- **Limit retries** to avoid livelock
An Optimistic Bounded Counter

public class BoundedCounterVOPT
    implements BoundedCounter
{
    protected Long count_ = new Long(MIN);
    protected synchronized boolean commit(Long oldc, Long newc)
    {
        boolean success = (count_ == oldc);
        if (success) count_ = newc;
        return success;
    }
    ...
}
An Optimistic Bounded Counter

... public long value() { return count_.longValue(); } public void inc() {
    for (;;) { // thinly disguised busy-wait!
        Long c = count_; long v = c.longValue();
        if (v < MAX && commit(c, new Long(v+1)))
            break;
        Thread.currentThread().yield();
        // is there another thread?!
    }
} ...

What you should know!

- What criteria might you use to prioritize threads?
- What are different possible definitions of fairness?
- What are readers and writers problems?
- What difficulties do readers and writers pose?
- When should you consider using optimistic methods?
- How can an optimistic method fail? How do you detect failure?
Can you answer these questions?

✎ When does it make sense to *split locks*? How does it work?
✎ When should you provide a policy for *upgrading* readers to writers?
✎ What are the dangers in letting the (Java) *scheduler* choose which writer may enter a critical section?
✎ What are advantages and disadvantages of encapsulating synchronization conditions as *helper methods*?
✎ How can optimistic methods *livelock*?
11. *Lab session II*

The lab exercises will be available on the course web page:

www.iam.unibe.ch/~scg/Teaching/CP/
12. Architectural Styles for Concurrency

Overview

- What is Software Architecture?
- Three-layered application architecture
- Flow architectures
  - Active Prime Sieve
- Blackboard architectures
  - Fibonacci with Linda
Sources

Software Architecture

A **Software Architecture** defines a system in terms of computational **components** and **interactions** amongst those components.

An **Architectural Style** defines a family of systems in terms of a pattern of structural organization.

— cf. Shaw & Garlan, Software Architecture, pp. 3, 19
Architectural style

Architectural styles typically entail four kinds of properties:

- **A vocabulary of design elements**
  - e.g., “pipes”, “filters”, “sources”, and “sinks”

- **A set of configuration rules that constrain compositions**
  - e.g., pipes and filters must alternate in a linear sequence

- **A semantic interpretation**
  - e.g., each filter reads bytes from its input stream and writes bytes to its output stream

- **A set of analyses that can be performed**
  - e.g., if filters are “well-behaved”, no deadlock can occur, and all filters can progress in tandem
Communication Styles

Shared Variables
Processes communicate *indirectly*.
Explicit synchronization mechanisms are needed.

Message-Passing
Communication and synchronization are *combined*. 
Simulated Message-Passing

Most concurrency and communication styles can be simulated by one another:

Unsynchronized objects

Synchronized objects

Message-passing can be modelled by associating message queues to each process.
Three-layered Application Architectures

Interaction with external world
Generating threads

Concurrency control
Locking, waiting, failing

Basic mechanisms

This kind of architecture avoids nested monitor problems by restricting concurrency control to a single layer.
Problems with Layered Designs

Hard to extend beyond three layers because:

- Control may depend on unavailable information
  - Because it is not safely accessible
  - Because it is not represented (e.g., message history)

- Synchronization policies of different layers may conflict
  - E.g., nested monitor lockouts

- Ground actions may need to know current policy
  - E.g., blocking vs. failing
Flow Architectures

Many synchronization problems can be avoided by arranging things so that information only flows in one direction from sources to filters to sinks.

Unix “pipes and filters”: Processes are connected in a linear sequence.

Control systems: events are picked up by sensors, processed, and generate new events.

Workflow systems: Electronic documents flow through workflow procedures.
Unix Pipes

Unix pipes are *bounded buffers* that *connect producer* and *consumer* processes (*sources, sinks and filters*):

```
cat file      # send file contents to output stream
| tr -c 'a-zA-Z' '\012' # put each word on one line
| sort         # sort the words
| uniq -c      # count occurrences of each word
| sort -rn     # sort in reverse numerical order
| more         # and display the result
```
Unix Pipes

Processes should *read* from standard input and *write* to standard output streams:

- Misbehaving processes give rise to "broken pipes"!

*Process creation* and *scheduling* are handled by the O/S.

*Synchronization* is handled implicitly by the I/O system (through buffering).
Flow Stages

Every flow stage is a *producer* or *consumer* or both:

- **Splitters** (Multiplexers) have multiple successors
  - *Multicasters* clone results to multiple consumers
  - *Routers* distribute results amongst consumers

- **Mergers** (Demultiplexers) have multiple predecessors
  - *Collectors* interleave inputs to a single consumer
  - *Combiners* process multiple input to produce a single result

- **Conduits** have both multiple predecessors and consumers
Flow Policies

Flow can be pull-based, push-based, or a mixture:

- **Pull-based flow**: Consumers *take* results from Producers
- **Push-based flow**: Producers *put* results to Consumers
- **Buffers**:
  - Put-only buffers (*relays*) connect push-based stages
  - Take-only buffers (*pre-fetch buffers*) connect pull-based stages
  - Put-Take buffers connect (adapt) push-based stages to pull-based stages

![Diagram of flow policies with a producer, buffer, and consumer]
Limiting Flow

**Unbounded buffers**: If producers are faster than consumers, buffers may exhaust available memory.

**Unbounded threads**: Having too many threads can exhaust system resources more quickly than unbounded buffers.

**Bounded buffers**: Tend to be either always full or always empty, depending on relative speed of producers and consumers.

**Bounded thread pools**: Harder to manage than bounded buffers.
Example: a Pull-based Prime Sieve

Primes are agents that reject non-primes, pass on candidates, or instantiate new prime agents:

- ActivePrime(2)
- ActivePrime(3)
- ActivePrime(5)
- ActivePrime(7)
Using Put-Take Buffers

Each ActivePrime uses a one-slot buffer to feed values to the next ActivePrime.

The first ActivePrime *holds* the seed value 2, *gets* values from a TestForPrime, and *creates* new ActivePrime instances whenever it detects a prime value.
The PrimeSieve

The main PrimeSieve class creates the initial configuration

```java
public class PrimeSieve {
    public static void main(String args[]) {
        genPrimes(1000);
    }
    public static void genPrimes(int n) {
        try {
            ActivePrime firstPrime =
                new ActivePrime(2, new TestForPrime(n));
        } catch (Exception e) {
        }
    }
}
```
Pull-based integer sources

Active primes get values to test from an `IntSource`:

```java
interface IntSource { int getInt(); }

class TestForPrime implements IntSource {
    private int nextValue;
    private int maxValue;
    public TestForPrime(int max) {
        this.nextValue = 3; this.maxValue = max;
    }
    public int getInt() { // not synched!
        if (nextValue < maxValue) { return nextValue++; }
        else { return 0; }
    }
}
```
The ActivePrime Class

ActivePrimes themselves implement IntSource

class ActivePrime
    extends Thread implements IntSource {
        private static IntSource lastPrime; // shared
        private int value; // this prime
        private int square; // its square
        private IntSource intSrc; // ints to test
        private Slot slot; // to pass values on
    ...

The ActivePrime Class

... public ActivePrime(int value, IntSource intSrc)
    throws ActivePrimeFailure
{
    this.value = value;
...
    slot = new Slot();  // NB: private
    lastPrime = this;   // unsynchronized (safe!)
    this.start();      // become active
}
...

It is impossible for primes to be discovered out of order!
The ActivePrime Class …

...  

public int value() {  
    return this.value;  
}  

private void putInt(int val) {    // may block  
    slot.put()(new Integer(val));  
}  

public int getInt() {    // may block  
    return ((Integer) slot.get()).intValue();  
}  

...

The only synchronization is hidden in the Slot class.
The ActivePrime Class ...

```java
public void run() {
    int testValue = intSrc.getInt(); // may block
    while (testValue != 0) { // stop
        if (this.square > testValue) { // got a prime
            try {
                new ActivePrime(testValue, lastPrime);
            } catch (Exception e) { break; } // exit loop
        } else if ((testValue % this.value) > 0) {
            this.putInt(testValue); // may block
        }
        testValue = intSrc.getInt(); // may block
    }
    putInt(0); // stop next
}
```
Blackboard Architectures

Blackboard architectures put all synchronization in a “coordination medium” where agents can exchange messages.

Agents do not exchange messages directly, but post messages to the blackboard, and retrieve messages either by reading from a specific location (i.e., a channel), or by posing a query (i.e., a pattern to match).
Result Parallelism

*Result parallelism* is a blackboard architectural style in which *workers* produce *parts* of a more complex whole.

Workers may be arranged hierarchically ...
Agenda Parallelism

*Agenda parallelism* is a blackboard style in which workers retrieve tasks to perform from a blackboard, and may generate new tasks to perform.

Workers repeatedly retrieve tasks until everything is done. Workers are typically able to perform *arbitrary tasks.*
**Specialist Parallelism**

*Specialist parallelism* is a style in which each worker is *specialized* to perform a particular task.

Specialist designs are equivalent to message-passing, and are often organized as *flow architectures*, with each specialist producing results for the next specialist to consume.
Linda

Linda is a *coordination medium*, with associated primitives for coordinating concurrent processes, that can be *added to an existing programming language*.

The coordination medium is a *tuple-space*, which can contain:

- *data tuples* — tuples of primitives values (numbers, strings ...)
- *active tuples* — expressions which are evaluated and eventually turn into data tuples
## Linda primitives

Linda’s coordination primitives are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>out(T)</code></td>
<td>output a tuple <code>T</code> to the medium (non-blocking)</td>
<td><code>out(&quot;employee&quot;, &quot;pingu&quot;, 35000)</code></td>
</tr>
<tr>
<td><code>in(S)</code></td>
<td>destructively input a tuple matching <code>S</code> (blocking)</td>
<td><code>in(&quot;employee&quot;, &quot;pingu&quot;, ?salary)</code></td>
</tr>
<tr>
<td><code>rd(S)</code></td>
<td>non-destructively input a tuple (blocking)</td>
<td></td>
</tr>
<tr>
<td><code>inp(S)</code></td>
<td>try to input a tuple</td>
<td></td>
</tr>
<tr>
<td><code>rdp(S)</code></td>
<td>report success or failure (non-blocking)</td>
<td></td>
</tr>
<tr>
<td><code>eval(E)</code></td>
<td>evaluate <code>E</code> in a new process</td>
<td></td>
</tr>
<tr>
<td></td>
<td>leave the result in the tuple space</td>
<td></td>
</tr>
</tbody>
</table>
Example: Fibonacci

A (convoluted) way of computing Fibonacci numbers with Linda:

```c
int fib(int n) {
    if (rdp("fib", n, ?fibn)) // non-blocking
        return fibn;
    if (n<2) {
        out("fib", n, 1); // non-blocking
        return 1;
    }
    eval("fib", n, fib(n-1) + fib(n-2)); // async
    rd("fib", n, ?fibn); // blocks
    return(fibn);
}
```

// Post-condition: rdp(“fib”, n, ?fibn) == True
Evaluating Fibonacci

\[
\text{fib}(5) \rightarrow \text{rdp fails, so start eval} \\
\text{eval(“fib”,5,fib(4)+fib(3))}
\]
Evaluating Fibonacci

\[
\begin{align*}
&\text{fib(5)} \\
&\text{blocks for result} \\
&\text{fib(4)} + \text{fib(3)} \\
&\text{rd("fib",5,?fn)} \\
&\text{eval("fib",5,fib(4)+fib(3))} \\
&\text{eval("fib",4,fib(3)+fib(2))}
\end{align*}
\]
Evaluating Fibonacci

``` Scheme
(fib 5)
```

``` Scheme
(fib 4) + (fib 3)
```

``` Scheme
(fib 3) + (fib 2)
```

``` Scheme
(fib 2) + (fib 1)
```

``` Scheme
(fib 1) + (fib 0)
```

``` Scheme
(eval "fib", 5, (fib 4) + (fib 3))
```

``` Scheme
(eval "fib", 4, (fib 3) + (fib 2))
```

``` Scheme
(eval "fib", 3, (fib 2) + (fib 1))
```

``` Scheme
(eval "fib", 2, (fib 1) + (fib 0))
```

``` Scheme
("fib", 1, 1)
```

Base level succeeds
Evaluating Fibonacci

\[
\begin{align*}
\text{fib}(5) & : \text{rd(“fib”,5,?fn)} \\
\text{fib}(4) + \text{fib}(3) & : \text{eval(“fib”,5,fib(4)+fib(3))} \\
\text{fib}(3) + \text{fib}(2) & : \text{eval(“fib”,4,fib(3)+fib(2))} \\
\text{fib}(2) + \text{fib}(1) & : \text{eval(“fib”,3,fib(2)+fib(1))} \\
\text{eval yields passive tuple} & : \text{(“fib”,2,2)} \\
\text{fib}(2) & : \text{(“fib”,1,1)} \\
\text{fib}(1) & : \text{(“fib”,0,1)} \\
\end{align*}
\]
Evaluating Fibonacci

```
fib(5)
```

Cached values are reused

```
rd(“fib”, 5, ?fn)
eval(“fib”, 5, fib(4) + fib(3))
eval(“fib”, 4, fib(3) + fib(2))
eval(“fib”, 3, fib(2) + fib(1))
(fib”, 2, 2)  
(fib”, 1, 1)  
(fib”, 0, 1)
```

```java
fib(2) + fib(1)
fib(3) + fib(2)
fib(4) + fib(3)
```
Evaluating Fibonacci

\[ \text{fib}(5) \]

\[
\begin{align*}
(“\text{fib”},5,8) \\
(“\text{fib”},4,5) \\
(“\text{fib”},3,3) \\
(“\text{fib”},2,2) \\
(“\text{fib”},1,1) &\quad ("\text{fib”},0,1)
\end{align*}
\]
What you should know!

✎ What is a **Software Architecture**?
✎ What are advantages and disadvantages of **Layered Architectures**?
✎ What is a **Flow Architecture**? What are the options and tradeoffs?
✎ What are **Blackboard Architectures**? What are the options and tradeoffs?
✎ How does **result parallelism** differ from **agenda parallelism**?
✎ How does **Linda** support **coordination** of concurrent agents?
Can you answer these questions?

✎ How would you model message-passing agents in Java?
✎ How would you classify Client/Server architectures?
✎ Are there other useful styles we haven’t yet discussed?
✎ How can we prove that the Active Prime Sieve is correct? Are you sure that new Active Primes will join the chain in the correct order?
✎ Which Blackboard styles are better when we have multiple processors?
✎ Which are better when we just have threads on a monoprocessor?
✎ What will happen if you start two concurrent Fibonacci computations?
13. Petri Nets

Overview

- **Definition:**
  - places, transitions, inputs, outputs
  - firing enabled transitions

- **Modelling:**
  - concurrency and synchronization

- **Properties of nets:**
  - liveness, boundedness

- **Implementing Petri net models:**
  - centralized and decentralized schemes

Petri nets: a definition

A Petri net $C = \langle P,T,I,O \rangle$ consists of:

1. A finite set $P$ of places
2. A finite set $T$ of transitions
3. An input function $I: T \rightarrow N^P$ (maps to bags of places)
4. An output function $O: T \rightarrow N^P$

A marking of $C$ is a mapping $\mu: P \rightarrow N$

Example:

$P = \{ x, y \}$
$T = \{ a, b \}$
$I(a) = \{ x \}$, $I(b) = \{ x, x \}$
$O(a) = \{ x, y \}$, $O(b) = \{ y \}$
$\mu = \{ x, x \}$
Firing transitions

To fire a transition $t$:
1. There must be enough input tokens: $\mu \geq I(t)$
2. Consume inputs and generate output: $\mu' = \mu - I(t) + O(t)$
Modelling with Petri nets

Petri nets are good for modelling:
- concurrency
- synchronization

Tokens can represent:
- resource availability
- jobs to perform
- flow of control
- synchronization conditions...
Concurrency

Independent inputs permit “concurrent” firing of transitions
Conflict

Overlapping inputs put transitions in conflict

Only one of a or b may fire
Mutual Exclusion

The two subnets are forced to synchronize
Fork and Join
Producers and Consumers

producer

consumer
Bounded Buffers

occupied slots

free slots
Reachability and Boundedness

Reachability:
- The reachability set $R(C, \mu)$ of a net $C$ is the set of all markings $\mu'$ reachable from initial marking $\mu$.

Boundedness:
- A net $C$ with initial marking $\mu$ is safe if places always hold at most 1 token.
- A marked net is $(k)$-bounded if places never hold more than $k$ tokens.
- A marked net is conservative if the number of tokens is constant.
Liveness and Deadlock

Liveness:

- A transition is *deadlocked* if it can never fire.
- A transition is *live* if it can never deadlock.

This net is both *safe* and *conservative.*
Transition a is *deadlocked.*
Transitions b and c are *live.*
The reachability set is \{\{y\}, \{z\}\}.

Are the examples we have seen *bounded? Are they live?*
Related Models

Finite State Processes
- Equivalent to regular expressions
- Can be modelled by one-token conservative nets

The FSA for: a(b|c)*d
Some Petri nets can be modelled by FSPs

Precisely which nets can (cannot) be modelled by FSPs?
Zero-testing Nets

Petri nets are not computationally complete
- Cannot model “zero testing”
- Cannot model priorities

A zero-testing net:
An equal number of a and b transitions may fire as a sequence during any sequence of matching c and d transitions.
(#a ≥ #b, #c ≥ #d)
Other Variants

There exist countless variants of Petri nets

Coloured Petri nets: Tokens are “coloured” to represent different kinds of resources

Augmented Petri nets: Transitions additionally depend on external conditions

Timed Petri nets: A duration is associated with each transition
Applications of Petri nets

Modelling information systems:

- Workflow
- Hypertext (*possible transitions*)
- Dynamic aspects of OODB design
Implementing Petri nets

We can implement Petri net structures in either centralized or decentralized fashion:

Centralized:
- A single "net manager" monitors the current state of the net, and fires enabled transitions.

Decentralized:
- Transitions are processes, places are shared resources, and transitions compete to obtain tokens.
Centralized schemes

In one possible centralized scheme, the Manager selects and fires enabled transitions.

Concurrently enabled transitions can be fired in parallel.

What liveness problems can this scheme lead to?
Decentralized schemes

In decentralized schemes transitions are processes and tokens are resources held by places:

Transitions can be implemented as thread-per-message gateways so the same transition can be fired more than once if enough tokens are available.
Transactions

Transitions attempting to fire must grab their input tokens as an *atomic transaction*, or the net may deadlock even though there are enabled transitions!

If \(a\) and \(b\) are implemented by independent processes, and \(x\) and \(y\) by shared resources, this net can deadlock even though \(b\) is enabled if \(a\) (incorrectly) grabs \(x\) and waits for \(y\).
Coordinated interaction

A simple solution is to treat the state of the entire net as a single, shared resource:

After a transition fires, it notifies waiting transitions.

How could you refine this scheme for a distributed setting?
What you should know!

✎ How are Petri nets formally specified?
✎ How can nets model concurrency and synchronization?
✎ What is the “reachability set” of a net? How can you compute this set?
✎ What kinds of Petri nets can be modelled by finite state processes?
✎ How can a (bad) implementation of a Petri net deadlock even though there are enabled transitions?
✎ If you implement a Petri net model, why is it a good idea to realize transitions as “thread-per-message gateways”?
Can you answer these questions?

✎ What are some simple conditions for guaranteeing that a net is bounded?
✎ How would you model the Dining Philosophers problem as a Petri net? Is such a net bounded? Is it conservative? Live?
✎ What could you add to Petri nets to make them Turing-complete?
✎ What constraints could you put on a Petri net to make it fair?