Concurrent Programming

What is a Concurrent Program?

A sequential program has a single thread of control.

A concurrent program has multiple threads of control allowing it to perform multiple computations in parallel and to control multiple external activities which occur at the same time.

Why Concurrent Programming?

- Performance gain from multiprocessing hardware
  - parallelism.
- Increased application throughput
  - an I/O call need only block one thread.
- Increased application responsiveness
  - high priority thread for user requests.
- More appropriate structure
  - for programs which interact with the environment, control multiple activities and handle multiple events.

Do I need to know about concurrent programming?

Concurrency is widespread but error prone.

- Therac - 25 computerised radiation therapy machine
  Concurrent programming errors contributed to accidents causing deaths and serious injuries.
- Mars Rover
  Problems with interaction between concurrent tasks caused periodic software resets reducing availability for exploration.

A Cruise Control System

When the car ignition is switched on and the on button is pressed, the current speed is recorded and the system is enabled: it maintains the speed of the car at the recorded setting.

Pressing the brake, accelerator or off button disables the system. Pressing resume re-enables the system.

- Is the system safe?
- Would testing be sufficient to discover all errors?

Models

A model is a simplified representation of the real world. Engineers use models to gain confidence in the adequacy and validity of a proposed design.

- focus on an aspect of interest - concurrency
- model animation to visualise a behaviour
- mechanical verification of properties (safety & progress)

Models are described using state machines, known as Labelled Transition Systems LTS. These are described textually as finite state processes (FSP) and displayed and analysed by the LTSA analysis tool.
modeling the Cruise Control System

Later chapters will explain how to construct models such as this so as to perform animation and verification.

programming practice in Java

Java is

- widely available, generally accepted and portable
- provides sound set of concurrency features

Hence Java is used for all the illustrative examples, the demonstrations and the exercises. Later chapters will explain how to construct Java programs such as the Cruise Control System.

"Toy" problems are also used as they crystallize particular aspects of concurrent programming problems!

course objective

This course is intended to provide a sound understanding of the concepts, models and practice involved in designing concurrent software.

The emphasis on principles and concepts provides a thorough understanding of both the problems and the solution techniques. Modeling provides insight into concurrent behavior and aids reasoning about particular designs. Concurrent programming in Java provides the programming practice and experience.

Learning outcomes...

After completing this course, you will know

- how to model, analyze, and program concurrent object-oriented systems.
- the most important concepts and techniques for concurrent programming.
- what are the problems which arise in concurrent programming.
- what techniques you can use to solve these problems.

Book

Concurrency: State Models & Java Programs, 2nd Edition

Jeff Magee & Jeff Kramer

WILEY

Course Outline

- Processes and Threads
- Concurrent Execution
- Shared Objects & Interference
- Monitors & Condition Synchronization
- Deadlock
- Safety and Liveness Properties
- Model-based Design
- Dynamic systems
- Concurrent Software Architectures
- Message Passing
- Timed Systems
Web based course material

[staff.city.ac.uk/c.kloukinas/concurrency](staff.city.ac.uk/c.kloukinas/concurrency)

◆ Java examples and demonstration programs
◆ State models for the examples
◆ Labelling Transition System Analyser (LTS) for modeling concurrency, model animation and model property checking.

Summary

◆ Concepts
  - we adopt a model-based approach for the design and construction of concurrent programs
◆ Models
  - we use finite state models to represent concurrent behavior.
◆ Practice
  - we use Java for constructing concurrent programs.

*Examples are used to illustrate the concepts, models and demonstration programs.*
concurrent processes
We structure complex systems as sets of simpler activities, each represented as a sequential process. Processes can overlap or be concurrent, so as to reflect the concurrency inherent in the physical world, or to offload time-consuming tasks, or to manage communications or other devices. Designing concurrent software can be complex and error prone. A rigorous engineering approach is essential.

Concepts:
- processes - units of sequential execution.
- finite state processes (FSP)
- labelled transition systems (LTS)

Practice:
- Java threads

modelling processes
A process is the execution of a sequential program. It is modelled as a finite state machine which transits from state to state by executing a sequence of atomic actions.

ONESHOT = (once -> STOP).

Can finite state models produce infinite traces?

FSP - action prefix
If x is an action and P a process then (x -> P) describes a process that initially engages in the action x and then behaves exactly as described by P.

ONESHOT state machine (terminating process)

Convention: actions begin with lowercase letters
- PROCESSES begin with uppercase letters
Repellive behaviour uses **recursion**:

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

Substituting to get a more succinct definition:

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

And again:

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

**FSP - action prefix & recursion**

**FSP - action prefix**

**FSP - choice**

If \(x\) and \(y\) are actions then \((x \rightarrow P | y \rightarrow Q)\) describes a process which initially engages in either of the actions \(x\) or \(y\). After the first action has occurred, the subsequent behavior is described by \(P\) if the first action was \(x\) and \(Q\) if the first action was \(y\).

Who or what makes the choice? Is there a difference between input and output actions?

**animation using LTSA**

The LTSA animator can be used to produce a trace.

Ticked actions are eligible for selection.

In the LTS, the last action is highlighted in red.

**Non-deterministic choice**

Process \((x \rightarrow P | x \rightarrow Q)\) describes a process which engages in \(x\) and then behaves as either \(P\) or \(Q\).

**FSP model of a drinks machine**:

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

**LTS generated using LTSA**:

Possible traces?

**animation using LTSA**

**FSP model of a traffic light**:

\[
\text{TRAFFICLIGHT} = (\text{red} \rightarrow \text{orange} \rightarrow \text{green} \rightarrow \text{orange} \rightarrow \text{TRAFFICLIGHT}).
\]

**LTS generated using LTSA**:

Trace:

red \rightarrow orange \rightarrow green \rightarrow orange \rightarrow red \rightarrow orange \rightarrow green ...

**Non-deterministic choice**

Process \((x \rightarrow P | x \rightarrow Q)\) describes a process which engages in \(x\) and then behaves as either \(P\) or \(Q\).

**Tossing a coin**

\[
\text{COIN} = (\text{toss} \rightarrow \text{HEADS} | \text{toss} \rightarrow \text{TAILS}), \\
\text{HEADS} = (\text{heads} \rightarrow \text{COIN}), \\
\text{TAILS} = (\text{tails} \rightarrow \text{COIN}).
\]

**Possible traces?**

**Could we make this deterministic and trace equivalent?**

**Would it really have equivalent behaviour?**
Modelling failure

How do we model an unreliable communication channel which accepts *in* actions and if a failure occurs produces no output, otherwise performs an *out* action?

Use non-determinism...

CHAN = (in->CHAN
       |in->out->CHAN
       ).

Deterministic?

FSP - indexed processes and actions

Local indexed process definitions are equivalent to process definitions for each index value

index expressions to model calculation:

const N = 1
range T = 0..N
range R = 0..2*N

SUM = (in[a:T][b:T]->TOTAL[a+b]),
      TOTAL[s:R] = (out[s]->SUM).

FSP - guarded actions

The choice (when B x -> P | y -> Q) means that when the guard B is true then the actions x and y are both eligible to be chosen, otherwise if B is false then the action x cannot be chosen.

COUNT (N=3) = COUNT[0],
COUNT[i:0..N] = (when (i<N) inc->COUNT[i+1]
                    |when (i>0) dec->COUNT[i-1]
                    ).

FSP - indexed processes and actions

Single slot buffer that inputs a value in the range 0 to 3 and then outputs that value:

BUFF = (in[i:0..3]->out[i]->BUFF).

equivalent to

BUFF = (in[0]->out[0]->BUFF
        |in[1]->out[1]->BUFF
        |in[2]->out[2]->BUFF
        |in[3]->out[3]->BUFF
       ).

or using a process parameter with default value:

BUFF(N=3) = (in[i:0..N]->out[i]->BUFF).

Indexed actions generate labels of the form: action.index

FSP - guarded actions

A countdown timer which, once started, beeps after N ticks, or can be stopped.

COUNTDOWN (N=3) = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] = (when (i>0) tick->COUNTDOWN[i-1]
                          |when (i==0) beep->STOP
                          |stop->STOP
                         ).
A countdown timer which, once started, beeps after N ticks, or can be stopped.

\[
\text{COUNTDOWN (N=3) } = \{ \text{start} \rightarrow \text{COUNTDOWN}[N]\}, \\
\text{COUNTDOWN}[i:0..N] = \\
(\text{when}(i>0) \text{tick} \rightarrow \text{COUNTDOWN}[i-1] \\
| \text{when}(i==0) \text{beep} \rightarrow \text{STOP} \\
| \text{stop} \rightarrow \text{STOP} ).
\]

A countdown timer which, once started, beeps after N ticks, or can be stopped.

const False = 0

P = (when (False) doanything \rightarrow P).

What is the following FSP process equivalent to?

\[
\text{FILTER} = (\text{in}[v:0..5] \rightarrow \text{DECIDE}[v]), \\
\text{DECIDE}[v:0..5] = ( \ ? ).
\]

Process alphabets are implicitly defined by the actions in the process definition.

The alphabet of a process can be displayed using the LTSA alphabet window.

Note: to avoid confusion, we use the term process when referring to the models, and thread when referring to the implementation in Java.
### Implementing processes - the OS view

A (heavyweight) process in an operating system is represented by its code, data and the state of the machine registers, given in a descriptor. In order to support multiple (lightweight) threads of control, it has multiple stacks, one for each thread.

### threads in Java

A Thread class manages a single sequential thread of control. Threads may be created and deleted dynamically.

The Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class.

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

public class MyThread extends Thread {
    public void run() {
        //......
    }
}
```

Creating and starting a thread object:

```java
Thread a = new MyThread();
a.start();
```

### thread alive states in Java

Once started, an alive thread has a number of substates:

- **Runnable**
  - start() causes the thread to call its run() method.
  - run() returns

- **Running**
  - yield() timeslice
  - dispatch
  - sleep()

- **Non-Runnable**
  - wait()

### thread life-cycle in Java

An overview of the life-cycle of a thread as state transitions:

- **Created**
  - new Thread()

- **Alive**
  - start()
  - run() returns

- **Terminated**
  - The predicate `isAlive()` can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted (cf. mortals).

### Java thread lifecycle - an FSP specification

```
THREAD = CREATED, CREATED = {start ->RUNNABLE},
RUNNABLE = {dispatch ->RUNNING},
RUNNING = {{sleep,wait} ->NON_RUNNABLE |
|{yield,timeslice} ->RUNNABLE |end ->TERMINATED |
|run ->RUNNING},
NON_RUNNABLE = {{timeout,notify} ->RUNNABLE},
TERMINATED = STOP.
```

**Dispatch, timeslice, end, run, and timeout** are not methods of class Thread, but model the thread execution and scheduler.
Java thread lifecycle - an LTS specification

States 0 to 4 correspond to CREATED, RUNNABLE, RUNNING, TERMINATED and NON-RUNNABLE respectively.

CountDown class

```java
public class CountDown extends Applet implements Runnable {
    Thread counter; int i;
    final static int N = 10;
    AudioClip beepSound, tickSound;
    NumberCanvas display;

    public void init() {...}
    public void start() {...}
    public void stop() {...}
    public void run() {...} // private
    private void tick() {...} // private
    private void beep() {...} // private
}
```

CountDown timer example

```java
COUNTDOWN (N=3) = (start->COUNTDOWN[N]),
COUNTDOWN[1:0..N] =
    (when(i>0) tick->COUNTDOWN[i-1]
     |when(i==0)beep->STOP
     |stop->STOP ),
```

Implementation in Java?

CountDown class - start(), stop() and run()

```java
public void start() {
    counter = new Thread(this);
    i = N; counter.start();
}
public void stop() {
    counter = null; }
public void run() {
    while(true) {
        if (counter == null) return;
        if (i>0) { tick(); --i; }
        if (i==0) { beep(); return; }
    }
}
```

CountDown timer - class diagram

The class CountDown derives from Applet and contains the implementation of the run() method which is required by Thread.

Summary

◆ Concepts
  ● process - unit of concurrency, execution of a program

◆ Models
  ● LTS to model processes as state machines - sequences of atomic actions
  ● FSP to specify processes using prefix "->", choice "|" and recursion.

◆ Practice
  ● Java threads* to implement processes.
  ● Thread lifecycle - created, running, runnable, non-runnable, terminated.

* see also java.util.concurrent
* cf. POSIX pthreads in C
Chapter 3

Concurrent Execution

Concepts: processes - concurrent execution and interleaving, process interaction.

Models: parallel composition of asynchronous processes - interleaving interaction - shared actions process labeling, and action relabeling and hiding structure diagrams

Practice: Multithreaded Java programs

3.1 Modeling Concurrency

◆ How should we model process execution speed?
  ● arbitrary speed (we abstract away time)

◆ How do we model concurrency?
  ● arbitrary relative order of actions from different processes (interleaving but preservation of each process order)

◆ What is the result?
  ● provides a general model independent of scheduling (asynchronous model of execution)

Definitions

◆ Concurrency
  ● Logically simultaneous processing. Does not imply multiple processing elements (PEs). Requires interleaved execution on a single PE.

◆ Parallelism
  ● Physically simultaneous processing. Involves multiple PEs and/or independent device operations.

Both concurrency and parallelism require controlled access to shared resources. We use the terms parallel and concurrent interchangeably and generally do not distinguish between real and pseudo-parallel execution.

parallel composition - action interleaving

If P and Q are processes then \( (P||Q) \) represents the concurrent execution of P and Q. The operator \(||\) is the parallel composition operator.

\[
\begin{align*}
\text{ITCH} & = (\text{scratch}>\text{STOP}). \\
\text{CONVERSE} & = (\text{think}>\text{talk}>\text{STOP}). \\
|\text{CONVERSE_ITCH} & = (\text{ITCH} \ || \ \text{CONVERSE}).
\end{align*}
\]

parallel composition - action interleaving

Possible traces as a result of action interleaving.

\[
\begin{align*}
\text{ITCH} \quad 2 \text{ states} & \quad \text{CONVERSE} \quad 3 \text{ states} \\
\text{CONVERSE_ITCH} \quad 2 \times 3 \text{ states}
\end{align*}
\]
Concurrency: concurrent execution

parallel composition - algebraic laws

Commutative: \((P || Q) = (Q || P)\)

Associative: \((P || (Q || R)) = ((P || Q) || R)\)

Clock radio example:

\[
\begin{align*}
\text{CLOCK} &= (\text{tick} \rightarrow \text{CLOCK}). \\
\text{RADIO} &= (\text{on} \rightarrow \text{off} \rightarrow \text{RADIO}). \\
\text{CLOCK} || \text{RADIO} &= (\text{CLOCK} || \text{RADIO}).
\end{align*}
\]

LTS? Traces? Number of states?

modeling interaction - shared actions

If processes in a composition have actions in common, these actions are said to be shared. Shared actions are the way that process interaction is modeled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in the shared action.

MAKER = (make \rightarrow \text{ready} \rightarrow \text{MAKER}).

USER = (\text{ready} \rightarrow \text{use} \rightarrow \text{USER}).

\text{CLOCK} \text{ RADIO} = (\text{CLOCK} || \text{RADIO}).

LTS? Traces? Number of states?

composite processes

A composite process is a parallel composition of primitive processes. These composite processes can be used in the definition of further compositions.

\[
\begin{align*}
\text{MAKER} &= (\text{make} \rightarrow \text{ready} \rightarrow \text{MAKER}). \\
\text{USER} &= (\text{ready} \rightarrow \text{use} \rightarrow \text{USER}). \\
\text{CLOCK} \text{ RADIO} &= (\text{CLOCK} || \text{RADIO}).
\end{align*}
\]

LTS? Traces? Number of states?

process labeling

A handshake is an action acknowledged by another:

\[
\begin{align*}
\text{MAKER} &= (\text{make} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKER}). \\
\text{USER} &= (\text{ready} \rightarrow \text{use} \rightarrow \text{used} \rightarrow \text{USER}). \\
\text{MAKER} || \text{USER} &= (\text{MAKER} || \text{USER}).
\end{align*}
\]

Interaction constrains the overall behaviour.

An array of instances of the switch process:

\[
\begin{align*}
\text{SWITCH} &= (\text{on} \rightarrow \text{off} \rightarrow \text{SWITCH}). \\
\text{TWO SWITCH} &= (\text{a:SWITCH} || \text{b:SWITCH}). \\
\text{SWITCHES} &= (\forall i : [1..N] \text{ a}[i]:\text{SWITCH}). \\
\text{SWITCHES} &= (\forall i : [1..N] \text{ a}[i]:\text{SWITCH}).
\end{align*}
\]

Multi-party synchronization:

\[
\begin{align*}
\text{MAKE A} &= (\text{makeA} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE A}). \\
\text{MAKE B} &= (\text{makeB} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE B}). \\
\text{ASSEMBLE} &= (\text{ready} \rightarrow \text{assemble} \rightarrow \text{used} \rightarrow \text{ASSEMBLE}). \\
\|\text{FACTORY} &= (\text{MAKE A} || \text{MAKE B} || \text{ASSEMBLE}).
\end{align*}
\]

Process labeling

\[
\begin{align*}
\text{a:P} \text{ prefixes each action label in the alphabet of P with a.}
\end{align*}
\]

Two instances of a switch process:

\[
\begin{align*}
\text{SWITCH} &= (\text{on} \rightarrow \text{off} \rightarrow \text{SWITCH}). \\
\|\text{TWO SWITCH} &= (\text{a:SWITCH} || \text{b:SWITCH}). \\
\end{align*}
\]

An array of instances of the switch process:

\[
\begin{align*}
\text{SWITCHES} &= (\forall i : [1..N] \text{ a}[i]:\text{SWITCH}). \\
\|\text{SWITCHES} &= (\forall i : [1..N] \text{ a}[i]:\text{SWITCH}).
\end{align*}
\]
process labeling by a set of prefix labels

(a1,...,ax)::P replaces every action label n in the alphabet of P with the labels a1.n,...,ax.n. Further, every transition (n->X) in the definition of P is replaced with the transitions ((a1.n,...,ax.n)->X).

Process prefixing is useful for modeling shared resources:
- RESOURCE = (acquire->release->RESOURCE).
- USER = (acquire->use->release->USER).
- RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).

An alternative formulation of the client server system is described below using qualified or prefixed labels:

CLIENTv2 = (call->request)->continue->SERVERv2.

action relabeling

Relabeling functions are applied to processes to change the names of action labels. The general form of the relabeling function is:

/(newlabel_1/oldlabel_1, ..., newlabel_n/oldlabel_n).

Relabeling to ensure that composed processes synchronize on particular actions.

CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).

action hiding - abstraction to reduce complexity

When applied to a process P, the hiding operator \(\{a1.ax\}\) removes the action names a1.ax from the alphabet of P and makes these concealed actions "silent". These silent actions are labeled "tau". Silent actions in different processes are not shared.

Sometimes it is more convenient to specify the set of labels to be exposed.... (like defining an interface)

When applied to a process P, the interface operator @\(\{a1.ax\}\) hides all actions in the alphabet of P not labeled in the set a1.ax.
action hiding

The following definitions are equivalent:

\[ \text{USER} = (\text{acquire}\rightarrow\text{use}\rightarrow\text{release}\rightarrow\text{USER}) \backslash \{\text{use}\}. \]

\[ \text{USER} = (\text{acquire}\rightarrow\text{use}\rightarrow\text{release}\rightarrow\text{USER}) \oplus \{\text{acquire, release}\}. \]

Minimization removes hidden \( \tau \) actions to produce an LTS with equivalent observable behavior.

---

structure diagrams

Process \( P \) with alphabet \( \{a, b\} \).

Parallel Composition
\( \langle P \mid Q \rangle / \{m/a, m/b, c/d\} \)

Composite process
\( \langle S = (P \mid Q) \otimes \{x, y\} \rangle \)

---

structure diagrams

We use structure diagrams to capture the structure of a model expressed by the static combinators: parallel composition, relabeling and hiding.

\[ \text{range } T = 0..3 \]

\[ \text{BUFF} = (\text{in}[i:T] \rightarrow \text{out}[i] \rightarrow \text{BUFF}). \]

\[ \langle \langle \text{TWOBUF} = ? \rangle \rangle \]

---

3.2 Multi-threaded Programs in Java

Concurrency in Java occurs when more than one thread is alive. ThreadDemo has two threads which rotate displays.
ThreadDemo model

Interpret run, pause, stop as inputs, rotate as an output.

ThreadDemo implementation in Java - class diagram

Rotator class

class Rotator implements Runnable {
  public void run() { Runnable {
    try {
      while(true) ThreadPanel.rotate();
    } catch(InterruptedException e) {exit
      }
    }
  run() finishes if an exception is raised by Thread.interrupt().
}

ThreadPanel class

public class ThreadPanel extends Panel {
  // construct display with title and segment color c
  public ThreadPanel(String title, Color c) { }
  // rotate display of currently running thread 6 degrees
  // return value not used in this example
  public static boolean rotate()
    throw InterruptedException { } Try
  // create a new thread with target r and start it running
  public void start(Runnable r) { ThreadPanel.rotate() to move the display.
    thread = new DisplayThread(canvas,r, ...);
    thread.start();
  }
  // stop the thread using Thread.interrupt()
  public void stop() { ThreadPanel.rotate() to move the display.
    thread.interrupt();
  }
  // calls to rotate() are delegated to DisplayThread.
  Calls to rotate() are delegated to DisplayThread.
  Threads are created by the start() method, and terminated by the stop() method.
}

ThreadDemo class

public class ThreadDemo extends Applet {
  ThreadPanel A; ThreadPanel B;
  ThreadDemo creates two ThreadPanel displays when initialized.
  ThreadPanel manages the display and control buttons, and delegates calls to
  rotate() in DisplayThread. Rotator implements the runnable interface.
  public void init() { A = new ThreadPanel("Thread A", Color.blue);
    B = new ThreadPanel("Thread B", Color.blue);
    add(A); add(B);
  }
  public void start() { A.start(new Rotator());
    B.start(new Rotator());
  }
  public void stop() { A.stop();
    B.stop();
  }
  ThreadDemo creates two ThreadPanel displays when initialized and two threads when started.
  Threads are created by the start() method, and terminated by the stop() method.
}

Summary
◆ Concepts
  concurrent processes and process interaction
◆ Models
  Asynchronous (arbitrary speed) & so interleaving (arbitrary order).
  Parallel composition as a finite state process with action interleaving.
  Process interaction by shared actions.
  Process labeling and action relabeling and hiding.
  Structure diagrams
◆ Practice
  Multiple threads in Java.

Concurrency: concurrent execution
©Magee/Kramer
ThreadDemo model
Interpret run, pause, stop as inputs, rotate as an output.

Concurrency: concurrent execution
©Magee/Kramer
ThreadDemo implementation in Java - class diagram

Concurrency: concurrent execution
©Magee/Kramer
Rotator class

Concurrency: concurrent execution
©Magee/Kramer
ThreadDemo class

Concurrency: concurrent execution
©Magee/Kramer
ThreadPanel class

Concurrency: concurrent execution
©Magee/Kramer
Summary
◆ Concepts
  concurrent processes and process interaction
◆ Models
  Asynchronous (arbitrary speed) & so interleaving (arbitrary order).
  Parallel composition as a finite state process with action interleaving.
  Process interaction by shared actions.
  Process labeling and action relabeling and hiding.
  Structure diagrams
◆ Practice
  Multiple threads in Java.
Chapter 4

Shared Objects & Mutual Exclusion

Concepts: process interference, mutual exclusion.

Models: model checking for interference modeling mutual exclusion

Practice: thread interference in shared Java objects mutual exclusion in Java (synchronized objects/methods).

4.1 Interference

Ornamental garden problem:
People enter an ornamental garden through either of two turnstiles. Management wish to know how many are in the garden at any time.

The concurrent program consists of two concurrent threads and a shared counter object.

The Turnstile thread simulates the periodic arrival of a visitor to the garden every second by sleeping for a second and then invoking the increment() method of the counter object.

The Turnstile class extends Thread:

```java
private void go() {
    counter = new Counter(counterD);
    west = new Turnstile(westD,counter);
    east = new Turnstile(eastD,counter);
    west.start();
    east.start();
}
```

Note that counterD, westD and eastD are objects of NumberCanvas used in chapter 2.

The run() method exits and the thread terminates after Garden.MAX visitors have entered.
Counter class

```java
class Counter {
    int value = 0;

    Counter(NumberCanvas n) {
        display=n;
        display.setValue(value);
    }

    void increment() {
        int temp = value; //read value
        value = temp; //write value
        display.setValue(value);
    }
}
```

Hardware interrupts can occur at arbitrary times.

The `increment` method at the same time.

The `counter` simulates a hardware interrupt during an increment, between reading and writing to the shared counter value. Interrupt randomly calls `Thread.yield()` to force a thread switch.

```java
data=ReadFromDB(query);
newData = Compute(data);
WriteToDB(newData);
```

after the East and West turnstile threads have each incremented its counter 20 times, the garden people counter is not the sum of the counts displayed. Counter increments have been lost. Why?

The alphabet of process `VAR` is declared explicitly as a `const` constant, `VarAlpha` to ensure no unintended free actions in `VAR` ie: all actions in `VAR` must be controlled by a `TURNSTILE`.

```
read value
write value + 1
```

After the East and West turnstile threads have each incremented its counter 20 times, the garden people counter is not the sum of the counts displayed. Counter increments have been lost. Why?

```
const N = 4
range T = 0..N
set VarAlpha = {value.read[T].write[T]} 

VAR = VAR[0]
   write[newv:T] ->VAR[newv]) // input

TURNSTILE = (go -> RUN)
RUN = (arrive -> INCREMENT 
   |end -> TURNSTILE)
INCREMENT = (value.read[x:T] // input 
   -> value.write[x+1:T] // output 
   +VarAlpha)
```

```
||GARDEN = (east.TURNSTILE || west.TURNSTILE || (east.west.display ::value:VAR) 
   //go/(east.west).go).end //
```

Process `VAR` models read and write access to the shared counter value.

Increment is modeled inside `TURNSTILE` since Java method activations are not atomic i.e. thread objects `east` and `west` may interleave their read and write actions.
checking for errors - exhaustive analysis

Exhaustive checking - compose the model with a TEST process which sums the arrivals and checks against the display value:

```
TEST = TEST[0],
TEST[v:T] =
  (when (v<N) (east.arrive,west.arrive)->TEST[v+1]
   |end->CHECK[v]
  )
CHECK[v:T] =
  (display.value.read[u:T] ->
    (when (u==v) right -> TEST[v]
     |when (u!=v) wrong -> ERROR)
  )
{display.VarAlpha}.
```

ornamental garden model - checking for errors

```
||TESTGARDEN = (GARDEN || TEST).

Use LTS4 to perform an exhaustive search for ERROR.

Trace to property violation in TEST:
  go
east.arrive
  east.value.read.0
  west.arrive
  west.value.read.0
  east.value.write.1
  west.value.write.1
  end
  display.value.read.1
  wrong
```

Interference and Mutual Exclusion

Destructive update, caused by the arbitrary interleaving of read and write actions, is termed interference. (aka a "data race")

Interference bugs are extremely difficult to locate. The general solution is to give methods mutually exclusive access to shared objects.

Mutual exclusion can be modeled as atomic actions. (functional programming: no updates \(\rightarrow\) no interference)

The Java™ Tutorials: Concurrency

Immutable Objects

"An object is considered immutable if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.

Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state."

docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

(The fewer moving things when juggling, the better - code "more functional")

4.2 Mutual exclusion in Java

Concurrent activations of a method in Java can be made mutually exclusive by prefixing the method with the keyword synchronized.

We correct COUNTER class by deriving a class from it and making the increment method synchronized:

```
  class SynchronizedCounter extends Counter {
    SynchronizedCounter(NumberCanvas n)
    {super(n);}
    synchronized void increment()
    {
      super.increment();
    }
  }
```

mutual exclusion - the ornamental garden

Java associates a lock with every object. The Java compiler inserts code to acquire the lock before executing the body of the synchronized method and code to release the lock before the method returns. Concurrent threads are blocked until the lock is released.
Access to an object may also be made mutually exclusive by using the `synchronized` statement:

```java
class SynchronizedCounter {
  synchronized (object) { statements }
}
```

A less elegant way to correct the example would be to modify the `Turnstile.run()` method:

```java
synchronized (counter) {counter.increment();}
```

Why is this “less elegant”?

To ensure mutually exclusive access to an object, all object methods should be synchronized.

### 4.3 Modeling mutual exclusion

To add locking to our model, define a `LOCK`, compose it with the shared `VAR` in the garden, and modify the alphabet set:

```plaintext
LOCK = (acquire->release->LOCK).
set VarAlpha = {value.(read[T],write[T],
    acquire, release)}
```

Modify `TURNSTILE` to acquire and release the lock:

```plaintext
TURNSTILE = (go -> RUN),
RUN = (arrive -> INCREMENT
    | end -> TURNSTILE),
INCREMENT = (value.acquire
    -> value.read[x:T] -> value.write[x+1]
    -> value.release->RUN
    )+VarAlpha.
```

### Revised ornamental garden model - checking for errors

A sample animation

To model shared objects directly in terms of their synchronized methods, we can abstract the details by hiding.

For `SynchronizedCounter` we hide `read`, `write`, `acquire`, `release` actions.

```plaintext
\[ COUNTER = \text{INCREMENT} \cup \text{LOCK} \cup \text{VAR} \]
```

### COUNTER: Abstraction using action hiding

```plaintext
const N = 4
range T = 0..N
VAR = VAR[0],
LOCK = (acquire->release->LOCK).
INCREMENT = (acquire->read[x:T]
    -> (when (x<N) write[x+1]
    -> release->increment->INCREMENT
    )
    )+{read[T],write[T]}.
||COUNTER = (INCREMENT | LOCK | VAR ||\{increment\}).
```

We can give a more abstract, simpler description of a `COUNTER` which generates the same LTS:

```plaintext
COUNTER = COUNTER[0],
COUNTER[v:T] = (when (v<N) increment -> COUNTER[v+1]).
```

This therefore exhibits "equivalent" behavior i.e. has the same observable behavior.
Summary

◆ Concepts
  ◆ process interference
  ◆ mutual exclusion

◆ Models
  ◆ model checking for interference
  ◆ modeling mutual exclusion

◆ Practice
  ◆ thread interference in shared Java objects
  ◆ mutual exclusion in Java (synchronized objects/methods).
Chapter 5

Monitors & Condition Synchronization

Concepts: monitors:
- encapsulated data + access procedures
- mutual exclusion + condition synchronization
- single access procedure active in the monitor
- nested monitors

Models: guarded actions

Practice: private data and synchronized methods (exclusion), wait(), notify() and notifyAll() for condition synch.
- single thread active in the monitor at a time

5.1 Condition synchronization

A controller is required for a carpark, which only permits cars to arrive when the carpark is not full and does not permit cars to depart when there are no cars in the carpark. Car arrival and departure are simulated by separate threads.

carpark model

Events or actions of interest?
- arrive and depart

Identify processes.
- arrivals, departures and carpark control

Define each process alphabet

Define each process and interactions (structure).

Guarded actions are used to control arrive and depart.
carpark program

- Model - all entities are processes interacting by actions
- Program - need to identify threads and monitors
  - thread - active entity which initiates (output) actions
  - monitor - passive entity which responds to (input) actions.

For the carpark?

```java
class CarParkControl {
    protected int spaces;
    protected int capacity;
    CarParkControl(int n) {
        capacity = spaces = n;
    }
    synchronized void arrive() {
        if (spaces < capacity) {
            spaces++; // block if full?
        }
    }
    synchronized void depart() {
        if (spaces > 0) {
            spaces--; // block if empty?
        }
    }
}
```

How do we implement the control of CarParkControl?

### carpark program - Arrivals and Departures threads

```java
class Arrivals implements Runnable {
    CarParkControl carpark;
    Arrivals(CarParkControl c) {carpark = c;}
    public void run() {
        while (true) {
            try {
                ThreadPanel.rotate(330);
                Similarly Departures that calls
carpark.depart().
                carpark.arrive();
                ThreadPanel.rotate(30);
            } catch (InterruptedException e) {
                // Arrivals = the Subject of the Verb “arrive”
            }
        }
    }
}
```

### Carpark program - class diagram

```
class CarParkControl {
    protected int spaces;
    protected int capacity;
    CarParkControl(int n) {
        capacity = spaces = n;
    }
    synchronized void arrive() {
        if (spaces < capacity) {
            spaces++; // block if full?
        }
    }
    synchronized void depart() {
        if (spaces > 0) {
            spaces--; // block if empty?
        }
    }
}
```

```
carpark program - CarParkControl monitor

Instances of these are created by the start() method of the CarPark applet:

```java
public void start() {
    CarParkControl c =
        new DisplayCarPark(carDisplay,Places);
    arrivals.start(new Arrivals(c));
    departures.start(new Departures(c));
}
```

```java
class Arrivals implements Runnable {
    CarParkControl carpark;
    Arrivals(CarParkControl c) {carpark = c;}
    public void run() {
        while (true) {
            try {
                ThreadPanel.rotate(330);
                Similarly Departures that calls
carpark.depart().
                carpark.arrive();
                ThreadPanel.rotate(30);
            } catch (InterruptedException e) {
                // Arrivals = the Subject of the Verb “arrive”
            }
        }
    }
}
```

```
class CarParkControl {
    protected int spaces;
    protected int capacity;
    CarParkControl(int n) {
        capacity = spaces = n;
    }
    synchronized void arrive() {
        if (spaces < capacity) {
            spaces++; // block if full?
        }
    }
    synchronized void depart() {
        if (spaces > 0) {
            spaces--; // block if empty?
        }
    }
}
```

Carpark program

- Arrivals and Departures implement Runnable,
  CarParkControl provides the control (condition synchronization).

```
class CarPark {
    CarPark(int n) {
        capacity = spaces = n;
    }
    synchronized void arrive() {
        if (spaces < capacity) {
            spaces++; // block if full?
        }
    }
    synchronized void depart() {
        if (spaces > 0) {
            spaces--; // block if empty?
        }
    }
}
```

```
class Departures implements Runnable {
    CarParkControl carpark;
    Departures(CarParkControl c) {carpark = c;}
    public void run() {
        while (true) {
            try {
                ThreadPanel.rotate(330);
                Similarly Arrivals that calls
carpark.arrive().
                carpark.depart();
                ThreadPanel.rotate(30);
            } catch (InterruptedException e) {
                // Departures = the Subject of the Verb “depart”
            }
        }
    }
}
```

```
class CarParkControl {
    protected int spaces;
    protected int capacity;
    CarParkControl(int n) {
        capacity = spaces = n;
    }
    synchronized void arrive() {
        if (spaces < capacity) {
            spaces++; // block if full?
        }
    }
    synchronized void depart() {
        if (spaces > 0) {
            spaces--; // block if empty?
        }
    }
}
```

```
class CarPark {
    CarPark(int n) {
        capacity = spaces = n;
    }
    synchronized void arrive() {
        if (spaces < capacity) {
            spaces++; // block if full?
        }
    }
    synchronized void depart() {
        if (spaces > 0) {
            spaces--; // block if empty?
        }
    }
}
```

```
class Departures implements Runnable {
    CarParkControl carpark;
    Departures(CarParkControl c) {carpark = c;}
    public void run() {
        while (true) {
            try {
                ThreadPanel.rotate(330);
                Similarly Arrivals that calls
carpark.arrive().
                carpark.depart();
                ThreadPanel.rotate(30);
            } catch (InterruptedException e) {
                // Departures = the Subject of the Verb “depart”
            }
        }
    }
}
```

```
class CarParkControl {
    protected int spaces;
    protected int capacity;
    CarParkControl(int n) {
        capacity = spaces = n;
    }
    synchronized void arrive() {
        if (spaces < capacity) {
            spaces++; // block if full?
        }
    }
    synchronized void depart() {
        if (spaces > 0) {
            spaces--; // block if empty?
        }
    }
}
```

```
class CarPark {
    CarPark(int n) {
        capacity = spaces = n;
    }
    synchronized void arrive() {
        if (spaces < capacity) {
            spaces++; // block if full?
        }
    }
    synchronized void depart() {
        if (spaces > 0) {
            spaces--; // block if empty?
        }
    }
}
```

```
class Departures implements Runnable {
    CarParkControl carpark;
    Departures(CarParkControl c) {carpark = c;}
    public void run() {
        while (true) {
            try {
                ThreadPanel.rotate(330);
                Similarly Arrivals that calls
carpark.arrive().
                carpark.depart();
                ThreadPanel.rotate(30);
            } catch (InterruptedException e) {
                // Departures = the Subject of the Verb “depart”
            }
        }
    }
}
```

```
class CarParkControl {
    protected int spaces;
    protected int capacity;
    CarParkControl(int n) {
        capacity = spaces = n;
    }
    synchronized void arrive() {
        if (spaces < capacity) {
            spaces++; // block if full?
        }
    }
    synchronized void depart() {
        if (spaces > 0) {
            spaces--; // block if empty?
        }
    }
}
```

```
class CarPark {
    CarPark(int n) {
        capacity = spaces = n;
    }
    synchronized void arrive() {
        if (spaces < capacity) {
            spaces++; // block if full?
        }
    }
    synchronized void depart() {
        if (spaces > 0) {
            spaces--; // block if empty?
        }
    }
}
```

```
class Departures implements Runnable {
    CarParkControl carpark;
    Departures(CarParkControl c) {carpark = c;}
    public void run() {
        while (true) {
            try {
                ThreadPanel.rotate(330);
                Similarly Arrivals that calls
carpark.arrive().
                carpark.depart();
                ThreadPanel.rotate(30);
            } catch (InterruptedException e) {
                // Departures = the Subject of the Verb “depart”
            }
        }
    }
}
```
Concurrent monitors & condition synchronization

Java provides a thread wait set per monitor (actually per object) with the following methods:

**public final void notify()**
- Wakes up a single thread that is waiting on this object’s set.

**public final void notifyAll()**
- Wakes up all threads that are waiting on this object’s set.

**public final void wait()**
- Throws InterruptedException
- Waits to be notified by another thread. The waiting thread releases the synchronization lock associated with the monitor. When notified, the thread must wait to reacquire the monitor before resuming execution.

### CarParkControl - condition synchronization

```java
class CarParkControl {
    // CARPARKCONTROLS(N=4) = SPACES[N].
    protected int spaces; // SPACES[0..N] = (when i=0) arrive->SPACES[i-1] 
    protected int capacity; // when (i=N) depart->SPACES[i+1] 
    CarParkControl(int n) {
        capacity = spaces = n;
    }
    synchronized void arrive() throws InterruptedException {
        --spaces;
        notifyAll();
    }
    synchronized void depart() throws InterruptedException {
        ++spaces;
        notifyAll();
    }
}
```

### models to monitors - summary

**Active** entities (that initiate actions) are implemented as threads.
**Passive** entities (that respond to actions) are implemented as monitors.

Each guarded action in the model of a monitor is implemented as a synchronized method, which uses a while loop and wait() to implement the guard. The while loop condition is the negation of the model guard condition.

Changes in the state of the monitor are signaled to waiting threads using notify() or notifyAll().

Watch out for transactions!

(what happens if an exception occurs after your method?)

Java provides a thread wait set per monitor (actually per object) with the following methods:

**public synchronized void act()**
- Throws InterruptedException
- While (cond) wait();
- // modify monitor data // NO EXCEPTIONS!
- notifyAll();

The while loop is necessary to retest the condition cond to ensure that cond is indeed satisfied when it re-enters the monitor.

notifyAll() is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed.

Watch out for transactions!

(what happens if an exception occurs after your method?)
5.2 Semaphores

Semaphores are widely used for dealing with inter-process synchronization in operating systems. A semaphore `s` is an integer variable that can take only non-negative values.

The only operations permitted on `s` are `up(s)` and `down(s)`. Blocked processes are held in a FIFO queue.

```
| `s` > 0 then |
| decrement `s` |
| else |
| block execution of the calling process |

| `s` = 0 then |
| awaken one of them |
| else |
| increment `s` |
```

semaphore demo - model

Three processes `{1..3}` use a shared semaphore `mutex` to ensure mutually exclusive access (action `critical`) to some resource.

```
LOOP = (mutex.down -> critical -> mutex.up -> LOOP).
|| SEMADEMO = (p[1..3]: LOOP
|| || p[1..3]: :: mutex: Semaphore(1))
```

For mutual exclusion, the semaphore initial value is 1. Why?

Is the ERROR state reachable for SEMADEMO?

Is a binary semaphore sufficient (i.e. Max=1)?

LTS?

modeling semaphores

To ensure analyzability, we only model semaphores that take a finite range of values. If this range is exceeded then we regard this as an ERROR. `N` is the initial value.

```
const Max = 3 const TRUE = 1
range Int = 0..Max
SEMAPHORE(N=0) = SEMA[N],
SEM[ v:Int ] = (when(TRUE) up->SEMA[v+1] |
when( v>0 ) down->SEMA[v-1]
),
SEM[Max+1] = ERROR.
```

semaphore demo - model

modeling semaphores

```
| up |
| up |
| up |
| up |
| up |
| -1 |
| 0 |
| 1 |
| 2 |
| 3 |
```

Action down is only accepted when value `v` of the semaphore is greater than 0.

Action up is not guarded.

Trace to a violation:

```
up → up → up → up
```

semaphores in Java

Semaphores are passive objects, therefore implemented as monitors.

```
public class Semaphore {
  private int value;
  public Semaphore (int initial) {
    value = initial;
  }
  synchronized public void up() {
    if (true) wait(); //????
    ++value;
    notifyAll();
  }
  synchronized public void down() {
    while (value== 0) wait();
    --value;
    notifyAll(); //????
  }
}
```

(Note: In practice, semaphores are a low-level mechanism often used for implementing the higher-level monitor construct.

Java SE5 provides general counting semaphores.)
Concurrency: monitors & condition synchronization

What if we adjust the time that each thread spends in its critical section?

- Large resource requirement - more conflict?
  (eg. more than 67% of a rotation)?
- Small resource requirement - no conflict?
  (eg. less than 33% of a rotation)?

Hence the time a thread spends in its critical section should be kept as short as possible.

Part III

5.3 Bounded Buffer

A bounded buffer consists of a fixed number of slots. Items are put into the buffer by a producer process and removed by a consumer process. It can be used to smooth out transfer rates between the producer and consumer.

(see car park example)
Some *System* Design Patterns

- Smooth out spikes:
  - Buffers (trade space for time)

- Increase throughput:
  - Parallelism:
    - SIMD (e.g., GPUs)
    - MIMD (e.g., Pipeline, threads)
  - Play the odds:
    - Pre-fetching (trade space for time)
    - Caching (trade space for time)

- Make changes easier:
  - Add indirection (pointers)

bounded buffer - a data-independent model

The behaviour of BOUNDEDBUFFER is independent of the actual data values, and so can be modelled in a data-independent manner.

LTS:

```
  0  1  2  3  4  5
  put put put put  put
  get  get  get  get  get
```

bounded buffer program - buffer monitor

```java
public interface Buffer {
    public synchronized void put(Object o) throws InterruptedException {
        while (count==size) wait();
        buf[in] = o; ++count; in=(in+1)%size;
        notify(); // notifyAll() ?
    }
    public synchronized Object get() throws InterruptedException {
        while (count==0) wait();
        Object o =buf[out];
        if (count>0
        buf[out]=null; --count; out=(out+1)%size;
        notify(); // notifyAll() ?
        return (o);
    }
}
```

bounded buffer program - producer process

```java
class Producer implements Runnable {
    Buffer buf;
    String alphabet= "abcdefghijklmnopqrstuvwxyz";
    Producer(Buffer b) {buf = b;}
    public void run() {
        try {
            int ai = 0;
            while(true) {
                ThreadPanel.rotate(12);
                buf.put(new Character(alphabet.charAt(ai)));
                ai=(ai+1) % alphabet.length();
                ThreadPanel.rotate(348);
            }
        } catch (InterruptedException e){}
    }
}
```

Part IV
Each Java object has a thread `wait` set and the following methods:

```java
public final void wait()
    throws InterruptedException
    Waits to be notified by another thread. The waiting thread releases the synchronization lock associated with the monitor. When notified, the thread must wait to reacquire the monitor before resuming execution.

public final void notify()
    notifies a single thread that is waiting on this object's set.
public final void notifyAll()
    wakes up all threads that are waiting on this object's set.

Notifying threads have no idea what the others are waiting for.
```

### 5.4 Nested Monitors!

Suppose that, in place of using the `count` variable and condition synchronization directly, we instead use two semaphores `full` and `empty` to reflect the state of the buffer.

```java
class SemaBuffer implements Buffer {
    Semaphore full; //counts number of items
    Semaphore empty; //counts number of spaces

    SemaBuffer(int size) {
        this.size = size; buf = new Object[size];
        full = new Semaphore(0);
        empty = new Semaphore(size);
    }

    // Semaphore's value = # available resources
}
```

We signal only those who care about our signal!

```java
synchronized public void put(Object o) throws InterruptedException {
    empty.down();
    buf[in] = o; ++count; in = (in+1)%size;
    full.up();
}
```

We signal only those who care about our signal!

```java
synchronized public Object get() throws InterruptedException{
    full.down();
    Object o =buf[out]; buf[out]=null;
    --count; out=(out+1)%size;
    empty.up();
    return (o);
}
```

### nested monitors - bounded buffer program

The bounded buffer model uses `buffer` to hold the characters, with a `producer` thread that tries to put characters into the buffer and a `consumer` thread that tries to get characters from the buffer. When the buffer is full, the producer thread is blocked until the consumer thread consumes a character.

```java
SemaBuffer(int size) {
    this.size = size; buf = new Object[size];
    full = new Semaphore(0);
    empty = new Semaphore(size);
    }
```

The `SemaBuffer` class implements the `Buffer` interface with its methods `put` and `get`. The `empty` semaphore is decremented during a `put` operation, which is blocked if `empty` is zero; `full` is decremented by a `get` operation, which is blocked if `full` is zero.

### nested monitors - bounded buffer model

The nested monitor problem occurs when a thread is blocked on a `wait` operation and another thread is blocked on a `notify` operation. This can occur in the bounded buffer program when the producer thread is blocked on `empty.down()` and the consumer thread is blocked on `full.up()`.

```java
class SemaBuffer implements Buffer {
    Semaphore full; //counts number of items
    Semaphore empty; //counts number of spaces

    SemaBuffer(int size) {
        this.size = size; buf = new Object[size];
        full = new Semaphore(0);
        empty = new Semaphore(size);
    }

    // Semaphore's value = # available resources
}
```

The `SemaBuffer` class is used to implement the bounded buffer model. It uses two semaphores, `full` and `empty`, to control access to the buffer.

### nested monitors - revised bounded buffer program

The only way to avoid it in Java is by careful design. In this example, the deadlock can be removed by ensuring that the monitor lock for the buffer is not acquired until after semaphores are decremented.

```java
public void put(Object o) throws InterruptedException {
    empty.down(); /* do I have the resources I need to proceed? */
    synchronized(this){
        // monitor starts here!
        buf[in] = o; ++count; in = (in+1)%size;
    }
    full.up();/* not inside the monitor; must keep
    critical region as short as possible.*/
}
```
The semaphore actions have been moved to the producer and consumer. This is exactly as in the implementation where the semaphore actions are outside the monitor.

**Does this behave as desired?**

**Minimized LTS?**

**Class Invariant Properties**

**Class constructor role:**
Establish the class invariant property.

*You don't know the class invariant?*
Then you don't know what the class is supposed to do.

Each method assumes that the invariant holds when it starts.

Each method must guarantee the invariant holds when it ends.

*You don't know the class invariant?*
Then you don't know what the class is supposed to do.

*Invariant hard to define?*
Maybe you’ve chosen the wrong fields…
(or you don’t know what the class is supposed to do)

**5.5 Monitor invariants**

An **invariant** for a monitor is an assertion on its fields. Invariants must hold (=non-variant) whenever no thread executes inside the monitor, i.e., on thread entry to and exit from a monitor.

- **CarParkControl Invariant:** $0 \leq \text{spaces} \leq N$
- **Semaphore Invariant:** $0 \leq \text{value}$
- **Buffer Invariant:** $0 \leq \text{count} \leq \text{size}$ and $0 \leq \text{in} < \text{size}$ and $0 \leq \text{out} < \text{size}$ and $\text{in} = (\text{out} + \text{count}) \ mod \ \text{size}$

Invariants can be helpful in reasoning about correctness of monitors using a logical proof-based approach. Generally, we prefer to use a model-based approach, as it’s amenable to mechanical checking.
Deadlock

Concepts: system deadlock: no further progress
four necessary & sufficient conditions

Models: deadlock - no eligible actions

Practice: blocked threads

Aim: deadlock avoidance - to design systems where deadlock cannot occur.

Deadlock: four necessary and sufficient conditions

- Serially reusable resources:
  the processes involved share resources which they use under mutual exclusion.

- Incremental acquisition:
  processes hold on to resources already allocated to them while waiting to acquire additional resources.

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

- Wait-for cycle:
  a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.

Wait-for cycle

A
B
C
D
E

Has A awaits B
Has B awaits C
Has C awaits D
Has D awaits E

6.1 Deadlock analysis - primitive processes

- deadlocked state is one with no outgoing transitions
- in FSP: STOP process

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

\[
\begin{array}{c}
0 \\
1 \\
2 \\
\end{array}
\]

- animation to produce a trace.
- analysis using LTSA:
  Trace to DEADLOCK:
  (shortest trace to STOP) north
  north

deadlock analysis - parallel composition

- in systems, deadlock may arise from the parallel composition of interacting processes.

\[
\begin{align*}
\text{RESOURCE} &= (\text{get} \rightarrow \text{put} \rightarrow \text{RESOURCE}). \\
\text{P} &= (\text{printer}.\text{get} \rightarrow \text{copy} \rightarrow \text{printer}.\text{put} \rightarrow \text{P}). \\
\text{Q} &= (\text{scanner}.\text{get} \rightarrow \text{printer}.\text{get} \rightarrow \text{copy} \rightarrow \text{scanner}.\text{put} \rightarrow \text{printer}.\text{put} \rightarrow \text{Q}). \\
\text{SYS} &= (\text{p:P} || \text{q:Q}).
\end{align*}
\]

Trace to DEADLOCK:

north

Avoidance?

Deadlock Trace?
6.2 Dining Philosophers

Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.

One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.

Trace to DEADLOCK:
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

This is the situation where all the philosophers become hungry at the same time, sit down at the table and each philosopher picks up the fork to his right. The system can make no further progress since each philosopher is waiting for a fork held by his neighbor i.e. a wait-for cycle exists!

Deadlock is easily detected in our model.

How easy is it to detect a potential deadlock in an implementation?
Concurrent Dining Philosophers - implementation in Java

```
philosophers: active entities - implement as threads
forks: shared passive entities - implement as monitors
display
```

```
class Fork {// FORK = (get -> put -> FORK).
    private boolean taken=false;
    private PhilCanvas display;
    private int identity;
    Fork(PhilCanvas disp, int id)
    { display = disp; identity = id;}
    synchronized void put()
    { taken=false; // WHY ?
        display.setFork(identity.taken);
        notify(); // WHY ?
    }
    synchronized void get()
    throws java.lang.InterruptedIOException {
        while (taken) wait(); // WHY ?
        taken=true;
        display.setFork(identity.taken);
    }
}
```

```
for (int i =0; i<N; ++i)
    fork[i] = new Fork(display,i);
for (int i =0; i<N; ++i){
    phil[i] = new Philosopher (this,i,fork[(i-1+N)%N],fork[i]);
    phil[i].start();
}
```

Dining Philosophers - Fork monitor

```
Guarded actions may be hidden in a model!

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

Actions get & put cannot happen at all times - they're guarded!

Encode the state of the LTS as an explicit variable to expose them:
FORK = TAKEN[0];
TAKEN[b:0..1] = (when (b) get -> TAKEN[b])
| when ( b) put -> TAKEN[b]).
```

```
To ensure deadlock occurs eventually, the slider control may be moved to the left. This reduces the time each philosopher spends thinking and eating. This "speedup" increases the probability of deadlock occurring.
```

```
Follows from the model (sitting down and leaving the table have been counted).
```
Deadlock-free Philosophers

Deadlock can be avoided by ensuring that a wait-for cycle cannot exist. How? Introduce an asymmetry into our definition of philosophers.

Use the identity I of a philosopher to make even numbered philosophers get their left forks first, odd their right first.

Other strategies?

Maze example - shortest path to “deadlock”

We can exploit the shortest path trace produced by the deadlock detection mechanism of LTSA to find the shortest path out of a maze to the STOP process!

We must first model the MAZE. Each position can be modelled by the moves that it permits. The MAZE parameter gives the starting position.

eg. MAZE(Start=8) = P[Start], P[0] = (north->STOP|east->P[1]),...

Summary

◆ Concepts
  ● deadlock: no further progress
  ● four necessary and sufficient conditions:
    ◦ serially reusable resources
    ◦ incremental acquisition
    ◦ no preemption
    ◦ wait-for cycle

◆ Models
  ● no eligible actions (analysis gives shortest path trace)

◆ Practice
  ● blocked threads

Aim: deadlock avoidance
- to design systems where deadlock cannot occur.
Chapter 7

Safety & Liveness Properties

Safety & liveness properties

Concepts:
- **safety**: true for every possible execution
- **liveness**: something good eventually happens

Models:
- **safety**: no reachable **ERROR**/**STOP** state
- **progress**: an action is eventually executed
- **fair choice** and **action priority**

Practice: threads and monitors

Aim: property satisfaction.

7.1 Safety

A safety property asserts that nothing bad happens.

- **STOP** or deadlocked state (no outgoing transitions)
- **ERROR** process (-1) to detect erroneous behaviour

Analysis using LTSA:
- Trace to **ERROR**
- Trace of shortest trace

Safety properties

Property that it is polite to knock before entering a room.

Traces: 
- **knock**→**enter**
- **enter**
- **knock**→**knock**

In all states, all the actions in the alphabet of a property are eligible choices.

Safety property **P** defines a deterministic process that asserts that any trace including actions in the alphabet of **P**, is accepted by **P**.

Thus, if **P** is composed with **S**, then traces of actions in the alphabet of **S** and alphabet of **P** must also be valid traces of **P** otherwise **ERROR** is reachable.

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. However, 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 behaviors in its alphabet.

Safety - mutual exclusion

How do we check that this does indeed ensure mutual exclusion in the critical section?

property MUX = \(\{p[1..3]\}.\)\{enter\} -> p[i].\{exit\} -> MUX .

\[ | \| \text{CHECK} = (\text{SEMADEMO} | | \text{MUTEX}). \]

What happens if semaphore is initialized to 2?

Part II – Single Lane Bridge

7.2 Single Lane Bridge problem

A bridge over a river is only wide enough to permit a single lane of traffic. Consequently, cars can only move concurrently if they are moving in the same direction. A safety violation occurs if two cars moving in different directions enter the bridge at the same time.

Single Lane Bridge - model

Events or actions of interest?

- enter and exit
- Identify processes, cars and bridge
- Define each process and interactions (structure)

property ONEWAY

The property focuses on system actions ONLY! Property doesn't care about the mechanism used to achieve it (here mutex down/up)
**Single Lane Bridge - CARS model**

```
const N = 3    // number of each type of car
range T = 0..N // type of car count
range ID= 1..N // car identities
CAR = (enter->exit->CAR).
```

To model the fact that cars cannot pass each other on the bridge, we model a CONVOY of cars in the same direction. We will have a red and a blue convoy of up to N cars for each direction:

```
||CARS = (red:CONVOY || blue:CONVOY).
```

**Single Lane Bridge - CONVOY model**

```
NOPASS1 = C[1],   //preserves entry order
C[i:ID] = ([i].enter-> C[i+N+1]).
NOPASS2 = C[1],   //preserves exit order
C[i:ID] = ([i].exit-> C[i+N+1]).

||CONVOY = ([ID]:CAR|NOPASS1|NOPASS2).
```

**Single Lane Bridge - model analysis**

```
||SingleLaneBridge = (CARS|| BRIDGE||ONEWAY).
```

**Single Lane Bridge - safety property ONEWAY**

```
property ONEWAY = (red[1].enter -> RED[1])
| blue[1].enter -> BLUE[1] |
RED[i:ID] = (red[ID].enter -> RED[i+1])
| when(i=1)red[ID].exit -> ONEWAY |
| when(i>1) red[ID].exit -> RED[i-1] |
// i is a count of red cars on the bridge
BLUE[i:ID]= (blue[ID].enter -> BLUE[i+1])
| when(i=1)blue[ID].exit -> ONEWAY |
| when( i>1)blue[ID].exit -> BLUE[i-1] |
// i is a count of blue cars on the bridge
```

**Single Lane Bridge - BRIDGE model**

Cars can move concurrently on the bridge only if in the same direction. The bridge maintains counts of blue and red cars on the bridge. Red cars are only allowed to enter when the blue count is zero and vice-versa.

```
Bridge = Bridge[0][0],   // initially empty
Bridge[0:T][nb:T] = (when (nb==0)  
| red[ID].enter -> BRIDGE[nr+1][nb] //nb==0  
| blue[ID].enter -> BRIDGE[nr][nb+1] //nr==0  
| blue[ID].exit -> BRIDGE[nr][nb-1]  
| blue[ID].exit -> BRIDGE[nr][nb-1]  
)
```

**Single Lane Bridge - implementation in Java**

Active entities (cars) are implemented as threads. Passive entity (bridge) is implemented as a monitor. BridgeCanvas enforces no overtaking.
Single Lane Bridge - BridgeCanvas

An instance of BridgeCanvas class is created by SingleLaneBridge applet - ref is passed to each newly created RedCar and BlueCar object.

class BridgeCanvas extends Canvas {
    public void init(int ncars) {...} // set number of cars
    public boolean moveRed(int i) {...} // move red car with the identity i a step
    public boolean moveBlue(int i) {...} // move blue car with the identity i a step
    public synchronized void freeze(){...} // freeze display
    public synchronized void thaw(){...} // unfreeze display
}

Single Lane Bridge - RedCar

class RedCar implements Runnable {
    BridgeCanvas display; Bridge control; int id;
    RedCar(Bridge b, BridgeCanvas d, int id) {
        display = d; this.id = id; control = b;
    }
    public void run() {
        try {
            while (true) {
                while (!display.moveRed(id)); // not on bridge
                control.redEnter(); // request access to bridge
                while (!display.moveRed(id)); // move over bridge
                control.redExit(); // release access to bridge
            }
        } catch (InterruptedException e) {} 
    }
}

Similarly for the BlueCar

Single Lane Bridge - class Bridge

class Bridge {
    public synchronized void redEnter() {...} throws InterruptedException 
    public synchronized void redExit() {...} 
    public synchronized void blueEnter() {...} 
    public synchronized void blueExit() {...} 
}

Class Bridge provides a null implementation of the access methods i.e. no constraints on the access to the bridge.

Result:...........?

Single Lane Bridge - SafeBridge

class SafeBridge extends Bridge {
    private int nred = 0; // number of red cars on bridge
    private int nblue = 0; // number of blue cars on bridge
    synchronized void redEnter() {...} throws InterruptedException 
        while (nred>0) wait();
        ++nred;
        synchronized void redExit() {...} 
        --nred;
    synchronized void blueEnter() {...} 
        --nblue;
        if (nblue==0) notifyAll();
}

This is a direct translation from the BRIDGE model.

To avoid unnecessary thread switches, we use conditional notification to wake up waiting threads only when the number of cars on the bridge is zero i.e. when the last car leaves the bridge.

But does every car eventually get an opportunity to cross the bridge? This is a liveness property.
7.3 Liveness

A safety property asserts that nothing bad happens.
A liveness property asserts that something good eventually happens.

Single Lane Bridge: Does every car eventually get an opportunity to cross the bridge?

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.

Suppose that there were two possible coins that could be picked up:
a trick coin and a regular coin......

progress $P = \{a_1, a_2, \ldots, a_N\}$ defines a progress property $P$ which asserts that in an infinite execution of a target system, at least one of the actions $a_1, a_2, \ldots, a_N$ will be executed infinitely often.

COIN system:
progress HEADS = \{heads\}✓
progress TAILS = \{tails\}✓

LTSA check progress: No progress violations detected.

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 heads would be chosen infinitely often and that tails would be chosen infinitely often. This requires Fair Choice!

Note: $\exists * \emptyset = \emptyset$.
Part IV – Checking Progress in the Single Lane Bridge

Progress analysis
A terminal set of states is one in which every state is reachable from every other state in the set via one or more transitions, and there is no transition from within the set to any state outside the set.

Terminal sets for TWOCOIN:
(1,2) and (3,4,5)

Given fair choice, each terminal set represents an execution in which each action used in a transition in the set is executed infinitely often. Since there is no transition out of a terminal set, any action that is not used in the set cannot occur infinitely often in all executions of the system - and hence represents a potential progress violation!

Progress - single lane bridge
The Single Lane Bridge implementation can permit progress violations. However, if default progress analysis is applied to the model then no violations are detected!

Progress - action priority
Action priority expressions describe scheduling properties:

Fair choice means that eventually every possible execution occurs, including those in which cars do not starve. To detect progress problems we must superimpose some scheduling policy for actions, which models the situation in which the bridge is congested.
Concurrency: safety & liveness properties

Progress - action priority

Action priority simplifies the resulting LTS by discarding lower priority actions from choices.

\[ ||HIGH = (NORMAL) << \{work\}. \]

\[ ||LOW = (NORMAL) >> \{work\}. \]

---

7.4 Congested single lane bridge

Progress violation: REDCROSS
Path to terminal set of states:
red.1.enter
red.2.enter
red.2.exit, red.3.enter, red.3.exit

Actions in terminal set:
\{red.1.enter, red.1.exit, red.2.enter, red.2.exit, red.3.enter, red.3.exit\}

Progress violation: BLUECROSS
Path to terminal set of states:
blue.1.enter
blue.2.enter
blue.2.exit, blue.3.enter, blue.3.exit

Actions in terminal set:
\{blue.1.enter, blue.1.exit, blue.2.enter, blue.2.exit, blue.3.enter, blue.3.exit\}

Congestion using action priority?

Could give red cars priority over blue (or vice versa)? In practice neither has priority over the other. Instead we merely encourage congestion by lowering the priority of the exit actions of both cars from the bridge.

\[ ||CongestedBridge = (SingleLaneBridge) >> \{red[ID].exit, blue[ID].exit\}. \]

---

congested single lane bridge model

Progress Analysis? LTS?

This corresponds with the observation that, with more than one car, it is possible that whichever color car enters the bridge first will continuously occupy the bridge preventing the other color from ever crossing.

---

Lecture 9 stopped here!

---

Progress - revised single lane bridge model

The bridge needs to know whether or not cars are waiting to cross.

Modify CAR:

\[ CAR = (request->enter->exit->CAR). \]

Modify BRIDGE:

\[ Red \text{ cars are only allowed to enter the bridge if there are no blue cars on the bridge and there are no blue cars waiting} \text{ to enter the bridge.} \]

\[ Blue \text{ cars are only allowed to enter the bridge if there are no red cars on the bridge and there are no red cars waiting} \text{ to enter the bridge.} \]

---

congested single lane bridge model

Will the results be the same if we model congestion by giving car entry to the bridge high priority?

Can congestion occur if there is only one car moving in each direction?
Concurrency: safety & liveness properties

Progress - revised single lane bridge model

/* nr–number of red cars on the bridge  
wr – number of red cars waiting to enter  
wb – number of blue cars waiting to enter  
*/  
BRIDGE = BRIDGE[0][0][0][0], 
BRIDGE[nr:T][nb:T][wr:T][wb:T] =  
{red[ID].request -> BRIDGE[nr][nb][wr+1][wb]  
|when (nb==0 && wb==0)  
  red[ID].enter -> BRIDGE[nr+1][nb][wr-1][wb]  
|red[ID].exit -> BRIDGE[nr-1][nb][wr][wb]  
|blue[ID].request -> BRIDGE[nr][nb][wr][wb+1]  
|when (nr==0 && wb==0)  
  blue[ID].enter -> BRIDGE[nr][nb+1][wr][wb-1]  
|blue[ID].exit -> BRIDGE[nr][nb-1][wr][wb]  
}).

Analysis - revised single lane bridge model

The trace is the scenario in which there are cars waiting at both ends, and consequently, the bridge does not allow either red or blue cars to enter.

Solution?

Introduce some asymmetry in the problem (cf. Dining philosophers).

This takes the form of a boolean variable (bt) which breaks the deadlock by indicating whether it is the turn of blue or red cars to enter the bridge.

Arbitrarily set bt to true initially giving blue initial precedence.

Revised single lane bridge implementation - FairBridge

```java
class FairBridge extends Bridge {
    private int nred = 0; // count of red cars on the bridge
    private int nblue = 0; // count of blue cars on the bridge
    private int waitred = 0; // count of waiting red cars
    private int waitblue = 0; // count of waiting blue cars
    private boolean blueturn = true;

    synchronized void redRequest() {  
        ++waitred;  
        while (nblue>0 && (waitblue>0 || blueturn)) wait();  
        --waitred;  
        ++nred;
    }

    synchronized void redEnter()  
        throws InterruptedException {
        ++waitred;  
        while (nblue>0 || (waitblue>0 && blueturn))  
            wait(); 
        --waitred;  
        if (nred==0) notifyAll();
    }

    synchronized void redExit()  
        throws InterruptedException {
        ++waitred;  
        while (nbr>0 || (waitred>0 && blueturn))  
            wait(); 
        --waitred;  
        if (nblue==0) notifyAll();
    }

    synchronized void blueRequest()  
        throws InterruptedException {
        ++waitblue;  
        while (nred>0 && (waitred>0 || blueturn))  
            wait(); 
        --waitblue;  
        ++nblue;
    }

    synchronized void blueEnter()  
        throws InterruptedException {  
        ++waitblue;  
        while (nred>0 || (waitred>0 && blueturn))  
            wait(); 
        --waitblue;  
        ++nblue;
    }

    synchronized void blueExit()  
        throws InterruptedException {  
        ++waitblue;  
        while (nred>0 || (waitred>0 && blueturn))  
            wait(); 
        --waitblue;  
        if (nblue==0) notifyAll();
    }
}
```

Concurrence: safety & liveness properties
7.5 Readers and Writers

A shared database is accessed by two kinds of processes: Readers and Writers. Readers execute transactions that examine and update the database while Writers both examine and update the database. A Writer must have exclusive access to the database; any number of Readers may concurrently access it.

**readers/writers model - READER & WRITER**

- Set actions
  - `set Actions = {acquireRead, releaseRead, acquireWrite, releaseWrite}`
- Reader
  - `READER = (acquireRead -> examine -> releaseRead -> READER)`
- Writer
  - `WRITER = (acquireWrite -> modify -> releaseWrite -> WRITER)`
- Readonly
  - `const Nread = 2` // Maximum readers
- Writeonly
  - `const Nwrite = 2` // Maximum writers
- RW_Lock
  - `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].`
An ERROR occurs if a reader or writer is badly behaved (release before acquire or more than two readers).

We can now compose the READWRITELOCK with READER and WRITER processes according to our structure...

We define an interface that identifies the monitor methods that must be implemented, and develop a number of alternative implementations of this interface. Firstly, the safe READWRITELOCK.

interface ReadWrite
{
    public void acquireRead();
    public void acquireWrite();
    public void releaseRead()
    throws InterruptedException;
    public void releaseWrite()
    throws InterruptedException;
    public void releaseWrite();
}

We concentrate on the monitor implementation:

class ReadWriteSafe implements ReadWrite
{
    private int readers = 0;
    private boolean writing = false;
    public void acquireRead();
    throws InterruptedException
    {
        while (writing) wait();
        ++readers;
    }
    public synchronized void acquireWrite();
    throws InterruptedException
    {
        while (writing) wait();
        ++writers;
    }
    public synchronized void releaseRead()
    {
        --readers;
        if (readers == 0) notify(); // notifyAll() ?
    }
    public synchronized void releaseWrite()
    {
        writing = false;
        notifyAll();
    }
}

We can now compose the READWRITELOCK.

Progress violation: WRITE
Path to terminal set of states: reader.1.acquireRead, reader.1.releaseRead, writer.2.acquireWrite, writer.2.releaseRead
Actions in terminal set:

Solver?

Unblock a single writer when no more readers.
(How do I know only writers are waiting?)

We lower the priority of the release actions for both readers and writers.

Try the Applet!

Solution?

Unblock all readers and writers!!

However, this monitor implementation suffers from the WRITE progress problem: possible writer starvation if the number of readers never drops to zero.
Part V – Readers & Writers – Priority

readers/writers - writer priority

Strategy:
Block readers if there is a writer waiting.

\[
\text{set } \text{Actions} = \{\text{acquireRead, releaseRead, acquireWrite, releaseWrite, requestWrite}\}
\]

\[
\text{WRITER} = (\text{requestWrite} -> \text{acquireWrite} -> \text{modify} -> \text{releaseWrite} -> \text{WRITER}) + \text{Actions}\{\text{modify}\}.
\]

readers/writers implementation - ReadWritePriority

```java
class ReadWritePriority implements ReadWrite{
    private int readers = 0;
    private boolean writing = false;
    private int waitingW = 0; // no of waiting Writers.

    public synchronized void acquireRead() throws InterruptedException {
        while (writing || waitingW > 0) wait();
        ++readers;
    }

    public synchronized void releaseRead() {
        --readers;
        if (readers == 0) notify(); // notifyAll();
    }

    public synchronized void acquireWrite() throws InterruptedException {
        try {
            // BAIL OUT: Tx strategy 1
            ++waitingW;
            while (readers > 0 || writing) wait();
        }
        finally {
            --waitingW;
        }
        writing = true;
    }

    public synchronized void releaseWrite() {
        writing = false;
        notifyAll();
    }
}
```

Progress violation: READ
Path to terminal set of states:
writer.1.requestWrite
writer.2.requestWrite
Actions in terminal set:
{writer.1.requestWrite, writer.1.acquireWrite, writer.1.releaseWrite, writer.2.requestWrite, writer.2.acquireWrite, writer.2.releaseWrite}

In practice, this may be satisfactory as it is usually more read access than write, and readers generally want the most up to date information.

readers/writers model - writer priority

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

Safety and Progress Analysis?

Readers starvation: if always a writer waiting.

No deadlocks/errors

Both READ and WRITE progress properties can be satisfied by introducing a turn variable as in the Single Lane Bridge.
readers/writers implementation - ReadWritePriority

```java
synchronized public void acquireWrite() {
    ++waitingW;
    while (readers>0 || writing)
        try{ wait(); } catch(InterruptedException e) {}  
    --waitingW;
    writing = true;
}

synchronized public void releaseWrite() {
    writing = false;
    notifyAll();
}
```

Both READ and WRITE progress properties can be satisfied by introducing a turn variable as in the Single Lane Bridge.

Summary

- **Concepts**
  - **properties**: true for every possible execution
  - **safety**: nothing bad happens
  - **liveness**: something good *eventually* happens

- **Models**
  - **safety**: no reachable ERROR/STOP state
  - **progress**: an action is always eventually executed

- **Practice**
  - threads and monitors

Single Lane Bridge problem – NOT ALL PROBLEMS HAVE A CENTRALISED CONTROLLER!!!

Here it’s implied.
But not every problem has one.

In distributed systems this is particularly the case (no centralised solutions desired, to minimise contention on that centralised controller).
Chapter 8

Model-Based Design

**Concurrent model-based design**

**Chapter 8**

**Model-Based Design**

**Concepts:**
- design process: requirements to models to implementations

**Models:**
- check properties of interest:
  - safety on the appropriate (sub)system
  - progress on the overall system

**Practice:**
- model interpretation - to infer actual system behavior
- threads and monitors

**Aim:**
- rigorous design process.

---

**A Cruise Control System - requirements**

When the car ignition is switched on and the on button is pressed, the current speed is recorded and the system is enabled. It maintains the speed of the car at the recorded setting.

Pressing the brake, accelerator or off button disables the system. Pressing resume or on re-enables the system.

**A Cruise Control System - hardware**

Parallel Interface Adapter (PIA) is polled every 100ms. It records the actions of the sensors:
- buttons (on, off, resume)
- brake (pressed)
- accelerator (pressed)
- engine (on, off).

Wheel revolution sensor generates interrupts to enable the car speed to be calculated.

Output: The cruise control system controls the car speed by setting the throttle via the digital-to-analogue converter.
model - design

- Main events, actions and interactions.
  - on, off, resume, brake, accelerator
  - engine on, engine off,
  - speed, setThrottle
  - clearSpeed, recordSpeed,
  - enableControl, disableControl

- Identify main processes.
  - Sensor Scan, Input Speed,
  - Cruise Controller, Speed Control and
  - Throttle

- Identify main properties.
  - safety - disabled when off, brake or accelerator pressed.

- Define and structure each process.

model elaboration - process definitions

```plaintext
// enable speed control when cruising,
// disable when off, brake or accelerator pressed
CRUISECONTROLLER = INACTIVE,
ACTIVE = {engineOn -> clearSpeed -> ACTIVE},
| {recordSpeed,enableControl} -> CRUISING |
| enableControl -> CRUISING |
| disableControl -> STANDBY |
| on->recordSpeed->enableControl->CRUISING |
| off,brake,accelerator |
STANDBY = {engineOff -> INACTIVE
| resume -> enableControl -> CRUISING |
| on->recordSpeed->enableControl->CRUISING |
```

model elaboration - process definitions

```plaintext
SENSORSCAN = {{Sensors} -> SENSORSCAN}.
// monitor speed when engine on
INPUTSPEED = (engineOn -> CHECKSPEED),
CHECKSPEED = {speed -> CHECKSPEED |
| engineOff -> INPUTSPEED |
| zoom when throttle set
THROTTLE = (setThrottle -> zoom -> THROTTLE).

// perform speed control when enabled
SPEEDCONTROL = DISABLED,
DISABLED = (speed,clearSpeed,recordSpeed)->DISABLED |
| enableControl -> ENABLED |
| disableControl -> DISABLED |
```

model elaboration - process definitions

Safety properties should be compositional. If there is no violation at a subsystem level, then there cannot be a violation when the subsystem is composed with other subsystems. This is because, if the ERROR state of a particular safety property is unreachable in the LTS of the subsystem, it remains unreachable in any subsequent parallel composition which includes the subsystem. Hence...

Safety properties should be composed with the appropriate system or subsystem to which the property refers. In order that the property can check the actions in its alphabet, these actions must not be hidden in the system.

```
Animate to check particular traces:
- Is control enabled after the engine is switched on and the on button is pressed?
- Is control disabled when the brake is then pressed?
- Is control re-enabled when resume is then pressed?
```

However, we need to analyse to exhaustively check:

```
Safety: Is the control disabled when off, brake or accelerator is pressed?
Progress: Can every action eventually be selected?
```
Concurrency: model-based design

**model - Safety properties**

```plaintext
property CRUISESAFETY =
  (off,accelerator,brake,disableControl) -> CRUISESAFETY
{on,resume} -> SAFETYCHECK
},
SAFETYCHECK =
  {(on,resume) -> SAFETYCHECK
  (off,accelerator,brake) -> SAFETYACTION
  (disableControl -> CRUISESAFETY
  )},
SAFETYACTION = (disableControl -> CRUISESAFETY).
```

**model analysis**

We can now compose the whole system:

```plaintext
||CONTROL =
  (CRUISECONTROLLER || SPEEDCONTROL || CRUISESAFETY
  )@ (Sensors,speed,setThrottle).
||CRUISECONTROLSYSTEM =
  (CONTROL || SENSORSCAN || INPUTSPEED || THROTTLE).
```

**model - Progress properties**

Progress checks are not compositional. Even if there is no violation at a subsystem level, there may still be a violation when the subsystem is composed with other subsystems.

This is because an action in the subsystem may satisfy progress yet be unreachable when the subsystem is composed with other subsystems which constrain its behavior. Hence...

**cruise control model - minimized LTS**

```plaintext
||CRUISEMINIMIZED = (CRUISECONTROLSYSTEM) @ (Sensors,speed).
```

**model - revised cruise control system**

Modify CRUISECONTROLLER so that control is disabled when the engine is switched off:

```plaintext
CRUISING = (engineoff -> disableControl -> INACTIVE
  | (off,brake,accelerator) -> disableControl -> STANDBY
  | on -> recordSpeed -> enableControl -> CRUISING
  ),
```

Modify the safety property:

```plaintext
property IMPROVEDSAFETY = ((off,accelerator,brake,disableControl, engineOff) -> IMPROVEDSAFETY
  | (on, resume) -> SAFETYCHECK
  ),
SAFETYCHECK = (on, resume) -> SAFETYCHECK
  | (off,accelerator,brake,engineOff) -> SAFETYACTION
  | disableControl -> IMPROVEDSAFETY
  ),
SAFETYACTION = (disableControl -> IMPROVEDSAFETY).
```
The central role of design architecture

Design architecture describes the gross organization and global structure of the system in terms of its constituent components.

We consider that the models for analysis and the implementation should be considered as elaborated views of this basic design structure.

8.2 from models to implementations

- identify the main active entities
  - to be implemented as threads
- identify the main (shared) passive entities
  - to be implemented as monitors
- identify the interactive display environment
  - to be implemented as associated classes
- structure the classes as a class diagram
### Concurrency: Model-based Design

**Summary**

- **Concepts**
  - design process: from requirements to models to implementations
  - design architecture

- **Models**
  - check properties of interest
  - safety: compose safety properties at appropriate (sub)system level

- **Practice**
  - model interpretation - to infer actual system behavior
  - threads and monitors

**Aim:** rigorous design process.

---

**Cruise Control System - Class Controller**

```java
class Controller {
    private static final int INACTIVE = 0;
    private static final int ACTIVE = 1;
    private static final int CRUISING = 2;
    private int controlState = INACTIVE;
    private Thread speedController;
    Controller(CarSpeed cs, CruiseDisplay disp) {
        cs = cs;
        disp = disp;
        if (controlState != INACTIVE) {
            synchronized {
                Thread.sleep(500);
                speedController = null;
            }
        }
    }
}
```

**Cruise Control System - Class SpeedControl**

```java
class SpeedControl implements Runnable {
    final static int DISABLED = 0;
    final static int ENABLED = 1;
    private int state = DISABLED;
    private int setSpeed = 0;
    private Thread speedController;
    private CarSpeed cs;
    private CruiseDisplay disp;
    SpeedControl(CarSpeed cs, CruiseDisplay disp) {
        this.cs = cs;
        this.disp = disp;
        disp.disable();
        disp.record(0);
    }
    synchronized void enableControl() {
        if (state == DISABLED) {
            state = ENABLED;
            disp.enable();
            speedController = new Thread(this);
        }
    }
    synchronized void disableControl() {
        if (state == DISABLED) {
            setSpeed = 0;
            disp.record(setSpeed);
        }
    }
    synchronized void clearSpeed() {
        if (state == DISABLED) {
            cs = cs;
            disp.disable();
            state = DISABLED;
        }
    }
    synchronized void brake() {
        if (controlState == CRUISING) {
            double steady = (double)setSpeed / 12.0;
            cs.setThrottle(steady, error);
        }
    }
    synchronized void engineOn() {
        if (controlState == INACTIVE) {
            sc = sc;
            sc.enableControl();
            controlState = CRUISING;
        }
    }
    synchronized void off() {
        if (controlState == CRUISING) {
            sc = sc;
            sc.disableControl();
            controlState = STANDBY;
        }
    }
    synchronized void resume() {
        if (controlState == STANDBY) {
            sc = sc;
            sc.enableControl();
            controlState = CRUISING;
        }
    }
}
```

### Course Outline

- Processes and Threads
- Concurrent Execution
- Shared Objects & Interference
- Monitors & Condition Synchronization
- Deadlock
- Safety and Liveness Properties
- Model-based Design
- Dynamic Systems
- Concurrent Software Architectures
- Message Passing
- Timed Systems
Chapter 10
Message Passing

**Concepts**: synchronous message passing - channel
asynchronous message passing - port
- send and receive / selective receive
rendezvous bidirectional comms - entry
- call and accept ... reply

**Models**: channel: relabelling, choice & guards
port: message queue, choice & guards
entry: port & channel

**Practice**:
distributed computing (disjoint memory)
threads and monitors (shared memory)

---

10.1 Synchronous Message Passing - channel

A sender communicates with a receiver using a single channel.
The sender sends a sequence of integer values from 0 to 9 and then restarts at 0 again.

```
Channel chan = new Channel();
rx.start(new Receiver(chan, recvdisp));
rx.start(new Receiver(chan, recvdisp));
```

Instances of Sender
Instances of Receiver

---

Java implementation - channel

```
class Channel extends Selectable {
    Object chann = null;
    public synchronized void send(Object v) throws InterruptedException {
        chann = v;
        signal();
        while (chann != null) wait();
    }
    public synchronized Object receive() throws InterruptedException {
        block(); clearReady();
        Object tmp = chann; chann = null;
        notifyAll(); //could be notify()
        return(tmp);
    }
}
```

Selectable is described later.

---

Java implementation - sender

```
class Sender implements Runnable {
    private Channel chan;
    private SlotCanvas display;
    Sender(Channel c, SlotCanvas d) {chan=c; display=d;}
    public void run() {
        try { int ei = 0;
            while(true) {
                display.enter(String.valueOf(ei));
                ThreadPanel.rotate(12);
                chan.send(new Integer(ei));
                display.leave(String.valueOf(ei));
                ei=(ei+1)%10; ThreadPanel.rotate(348);
            }
        } catch (InterruptedException e){}
    }
}
```

---
```java
class Receiver implements Runnable {
    private Channel chan;
    private SlotCanvas display;
    Receiver(Channel c, SlotCanvas d) {
        chan = c; display = d;
    }
    public void run() {
        try { Integer v = null;
            while (true) {
                ThreadPanel.rotate(180);
                v = (Integer) chan.receive();
                display.enter(v.toString());
                ThreadPanel.rotate(180);
            }
        } catch (InterruptedException e) {
        }
    }
}
```

**Java implementation - selective receive**

```java
public void run() {
    try {
        Integer v = null;
        while (true) {
            ThreadPanel.rotate(180);
            v = (Integer) chan.receive();
            display.enter(v.toString());
            ThreadPanel.rotate(180);
        }
    } catch (InterruptedException e) {
    }
}
```

**Java implementation - selective receive**

```java
class MsgCarPark implements Runnable {
    private Channel arrive, depart;
    private int spaces;
    private StringCanvas disp;
    public MsgCarPark(Channel a, Channel l,
        StringCanvas d, int capacity) {
        depart = l; arrive = a; N = spaces = capacity; disp = d;
    }
    public void run() {
        ...
    }
}
```

**Java implementation - selective receive**

```java
public void run() {
    try {
        Integer v = null;
        while (true) {
            ThreadPanel.rotate(120);
            v = (Integer) chan.receive();
            if (v != null) display.leave(v.toString());
            ThreadPanel.rotate(120);
        }
    } catch (InterruptedException e) {
    }
}
```

**Java implementation - selective receive**

```java
SELECT sel = { arrive => S1; depart => S2; }
```

**Java implementation - selective receive**

```java
SELECT sel = { arrive => S1; depart => S2; }
```

**Java implementation - selective receive**

```java
public void run() {
    try {
        Integer v = null;
        while (true) {
            ThreadPanel.rotate(120);
            v = (Integer) chan.receive();
            if (v != null) display.leave(v.toString());
            ThreadPanel.rotate(120);
        }
    } catch (InterruptedException e) {
    }
}
```

**Java implementation - selective receive**

```java
public void run() {
    try {
        Integer v = null;
        while (true) {
            ThreadPanel.rotate(120);
            v = (Integer) chan.receive();
            if (v != null) display.leave(v.toString());
            ThreadPanel.rotate(120);
        }
    } catch (InterruptedException e) {
    }
}
```

**Java implementation - selective receive**

```java
public void run() {
    try {
        Integer v = null;
        while (true) {
            ThreadPanel.rotate(120);
            v = (Integer) chan.receive();
            if (v != null) display.leave(v.toString());
            ThreadPanel.rotate(120);
        }
    } catch (InterruptedException e) {
    }
}
```
10.2 Asynchronous Message Passing - port

**Concurrency: message passing**
©Magee/Kramer

- **send(e,p)** - send the value of the expression e to port p. The process calling the send operation is not blocked. The message is queued at the port if the receiver is not ready.

- **v = receive(p)** - receive a value into local variable v from port p. The process calling the receive operation is blocked if there are no messages queued to the port.

**port model**

- **range** M = 0..9 // messages with values up to 9
- **set** S = {{M}, [M] [M]} // queue of up to three messages
- **PORT** // empty state, only send permitted
  - (send(x:M)->PORT[x]),
  - (send(x:M)->PORT[x][h] // one message queued to port
|receive[h] ->PORT)
  - ( PORT[t:S][h:M] // two or more messages queued to port
|receive[h] ->PORT[t])

// minimise to see result of abstracting from data values
||APORT = PORT/|send/send[M],receive/receive[M]|.

**model of applet**

- **Instances of ThreadPanel**
- **Instances of SlotCanvas**

**asynchronous message passing - applet**

Two senders communicate with a receiver via an "unbounded" port.

Each sender sends a sequence of integer values from 0 to 9 and then restarts at 0 again.

Instances of ThreadPanel

Instances of SlotCanvas

**Java implementation - port**

```java
class Port extends Selectable {
    Vector queue = new Vector();
    public synchronized void send(Object v) {
        queue.addElement(v);
    }
    public synchronized Object receive() throws InterruptedException {
        if (queue.isEmpty()) {
            signal();
            return;
        }
        Object tmp = queue.elementAt(0);
        queue.removeElementAt(0);
        return(tmp);
    }
}
```

**10.3 Rendezvous - entry**

Rendezvous is a form of request-reply to support client server communication. Many clients may request service, but only one is serviced at a time.

**LTS?**

**Safety?**
**Concurrency: message passing**

- `res=call(e,req)` - send the value `req` as a request message which is queued to the entry `e`.
- `req=accept(e)` - receive the value of the request message from the entry `e` into local variable `req`. The calling process is blocked if there are no messages queued to the entry.
- `reply(e,res)` - send the value `res` as a reply message to entry `e`.

**Java implementation - entry**

```java
public class Entry extends Port {
  private CallMsg cm;
  public Object call() throws InterruptedException {
    cm = (CallMsg) receive();
    return cm.request;
  }
  public void reply(Object res) throws InterruptedException {
    cm.replychan.send(res);
  }
  private class CallMsg {
    Object request; Channel replychan;
    CallMsg(Object request, Channel replychan) {
      [request, replychan];
    }
  }
}
```

**Asynchronous message passing - applet**

Two clients call a server which services a request at a time.

![Diagram of applet communication](image)

**Java implementation - entry**

Entries are implemented as extensions of ports, thereby supporting queuing and selective receipt.

The `call` method creates a channel object on which to receive the reply message. It constructs and sends to the entry a message consisting of a reference to this channel and a reference to the `req` object. It then awaits the reply on the channel.

The `accept` method keeps a copy of the channel reference; the `reply` method sends the reply message to this channel.

**Rendezvous**

- Blocks until a reply message is received into the local variable `req`.

**Model of entry and applet**

We reuse the models for ports and channels...

![Diagram of model](image)

**Rendezvous vs Monitor method invocation**

What is the difference?

- From the point of view of the client?
- From the point of view of the server?
- Mutual exclusion?

Which implementation is more efficient?

- In a local context (client and server in same computer)?
- In a distributed context (in different computers)?
Summary

◆ Concepts
  - Synchronous message passing – channel
  - Asynchronous message passing – port
    - Send and receive / selective receive
  - Rendezvous bidirectional comms – entry
    - Call and accept ... reply

◆ Models
  - Channel: relabelling, choice & guards
  - Port: message queue, choice & guards
  - Entry: port & channel

◆ Practice
  - Distributed computing (disjoint memory)
  - Threads and monitors (shared memory)

Course Outline

- Processes and Threads
- Concurrent Execution
- Shared Objects & Interference
- Monitors & Condition Synchronization
- Deadlock
- Safety and Liveness Properties
- Model-based Design
- Dynamic Systems
- Concurrent Software Architectures
- Message Passing
- Timed Systems