The Concurrent Programming Abstraction

Terminology

Ben-Ari — “control pointer”
Hardware — “program counter”

Each process has its own PC

Process P: p1, p2, p3, ...
Process Q: q1, q2, q3, ...
Process R: r1, r2, r3, ...

Example with two processes,
each with two statements

Process P: p1, p2
Process Q: q1, q2

Possible Interleavings

p1→q1→p2→q2,

p1→q1→q2→p2,

p1→p2→q1→q2,

q1→p1→q2→p2,

q1→p1→p2→q2,

q1→q2→p1→p2.

Algorithm 2.1: Trivial concurrent program	
integer n ← 0	
p	**q**
integer k1 ← 1	integer k2 ← 2
p1: n ← k1	q1: n ← k2

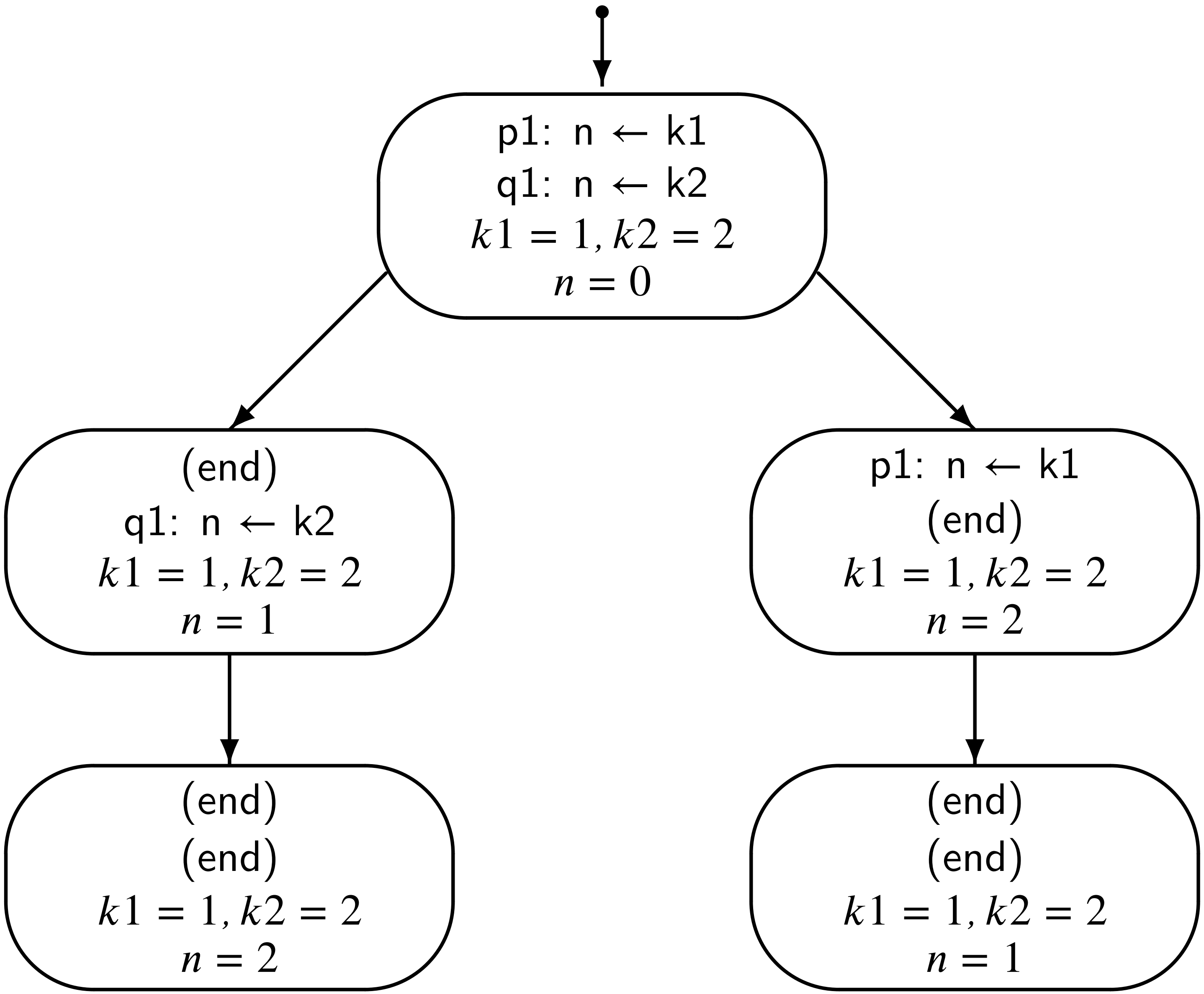
Algorithm 2.1

n is a global variable
k1 and k2 are local variables

What are the possible final values of n?

Can analyze with a state transition diagram.

State Diagram for a Concurrent Program



Homework problem

Devise an interleaving such that Algorithm 2.9 (next slide) terminates with a value of 10 for n.

Algorithm 2.9: Concurrent counting algorithm	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: do 10 times	q1: do 10 times
p2: temp ← n	q2: temp ← n
p3: n ← temp + 1	q3: n ← temp + 1

The compiler

The compiler translates a single statement of a high-order language to multiple machine language statements.

```
n = k + 1;
```

The compiler

The compiler translates a single statement of a high-order language to multiple machine language statements.

```
n = k + 1;
```

→

```
LDA  k,s
ADDA 1,i
STA  n,d
```

Fact

In practice, the interleaving takes place at the machine level, not the high-order language level. To do the analysis correctly, you must analyze Algorithm 2.1 as follows (Pep/8 assembly language).

integer n ← 0

p	q
integer n ← k1	integer n ← k2
p1: `LDA k1,s`	q1: `LDA k2,s`
p2: `STA n,d`	q2: `STA n,d`

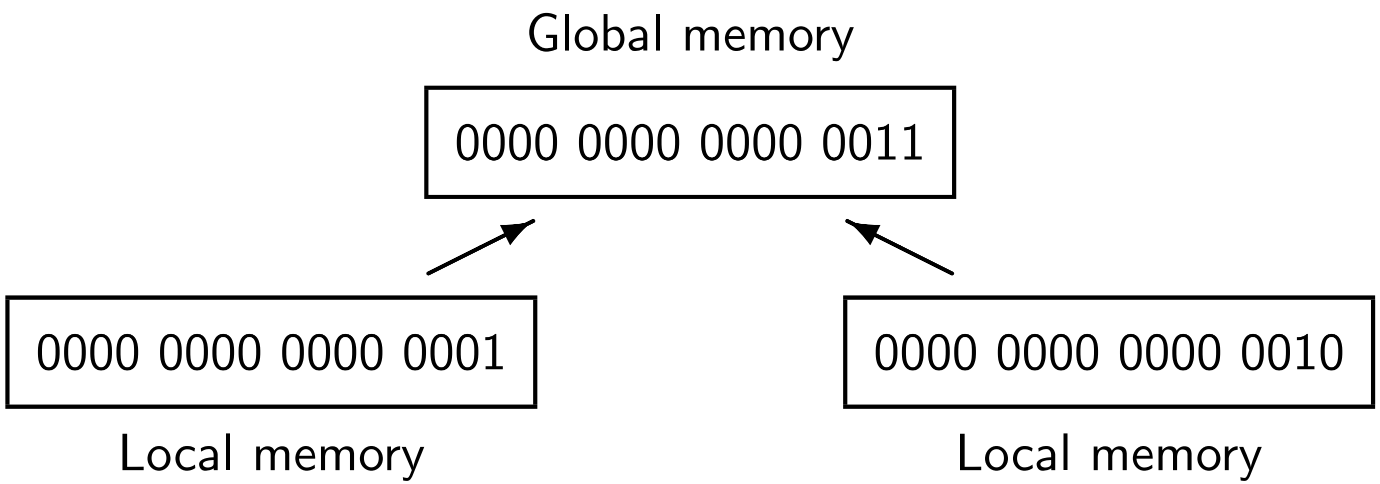
Justification of The Concurrency Theorem

Suppose in a multiprocessing system, one CPU tries to execute p2: `STA n,d` at the same time another CPU tries to execute q2: `STA n,d`.

See next slide.

The hardware will force one to go first, so the corruption in the next slide will not occur.

Inconsistency Caused by Overlapped Execution



Justification of The Concurrency Theorem

Conclusion:

The effect is the <u>same</u> as if an arbitrary interleaving happens in a <u>multiprogramming</u> system.

Atomic statements

A statement is atomic if it cannot be interleaved at a lower level of abstraction.

The atomic assumption in Ben-Ari's text:
All statements in the algorithms of Ben-Ari's text are assumed to be atomic.

Justification of the atomic assumption

It can make a difference in the analysis if you make the atomic assumption.

The following scenarios for Algorithm 2.3 makes the atomic assumption for the assignment statement.

Conclusion: The final value of n must be 2, regardless of which scenario occurs.

Algorithm 2.3: Atomic assignment statements	
integer n $\leftarrow$ 0	
p	**q**
p1: n $\leftarrow$ n + 1	q1: n $\leftarrow$ n + 1

Scenario for Atomic Assignment Statements

Process p	Process q	n
p1: n←n+1	q1: n←n+1	0
(end)	**q1: n←n+1**	1
(end)	(end)	2

Process p	Process q	n
p1: n←n+1	**q1: n←n+1**	0
p1: n←n+1	(end)	1
(end)	(end)	2

Justification of the atomic assumption

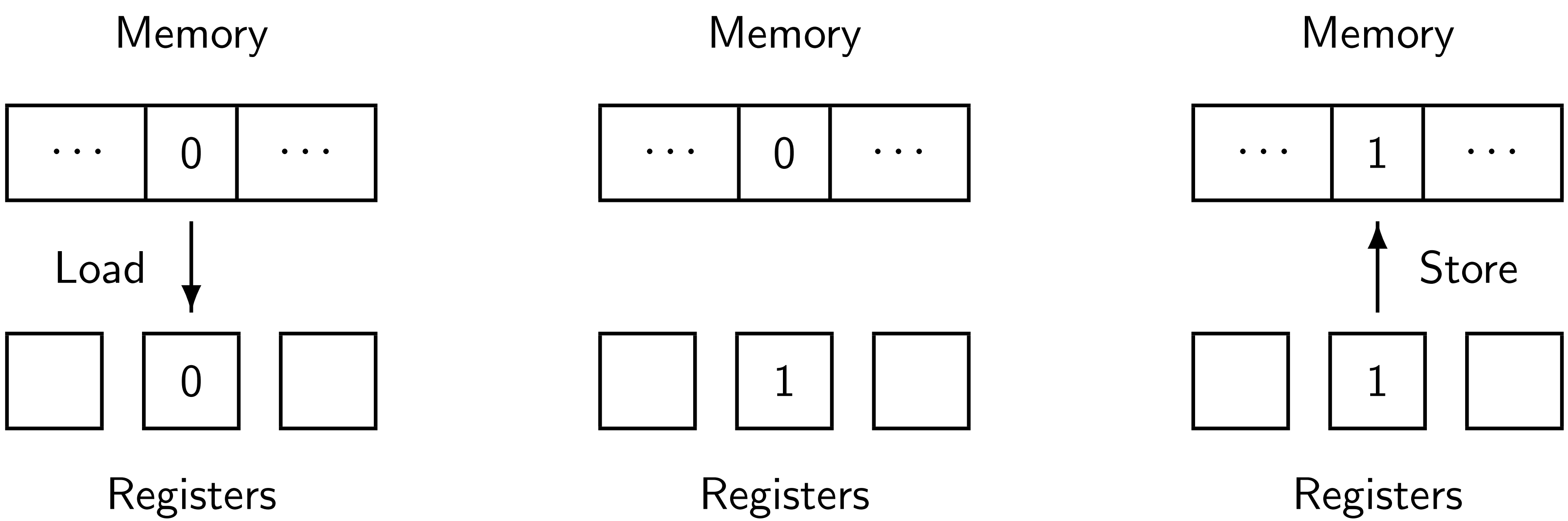
The following scenarios for Algorithm 2.3 do not make the atomic assumption.

(R1 corresponds to the accumulator of Pep8, # is immediate addressing, and direct addressing is default.)

Conclusion: The final value of n could be 1 or 2, depending on which scenario occurs.

Algorithm 2.6: Assignment statement for a register machine	
integer n ← 0	
p	**q**
p1: load R1,n	q1: load R1,n
p2: add R1,#1	q2: add R1,#1
p3: store R1,n	q3: store R1,n

Register Machine



Scenario for a Register Machine

Process p	Process q	n	p.R1	q.R1
p1: load R1,n	q1: load R1,n	0	?	?
p2: add R1,#1	**q1: load R1,n**	0	0	?
p2: add R1,#1	q2: add R1,#1	0	0	0
p3: store R1,n	**q2: add R1,#1**	0	1	0
p3: store R1,n	q3: store R1,n	0	1	1
(end)	**q3: store R1,n**	1	1	1
(end)	(end)	1	1	1

Justification of the atomic assumption

Even though the results are different depending on whether we make the atomic assumption, we can still <u>model</u> the nonatomic assumption with atomic assignment statements.

Algorithm 2.4 uses a temp variable that corresponds to the accumulator.

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Correct Scenario for Assignment Statements

Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	q1: temp←n	0	0	?
(end)	**q1: temp←n**	1	0	?
(end)	**q2: n←temp+1**	1	0	1
(end)	(end)	2	0	1

Incorrect Scenario for Assignment Statements

Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	**q1: temp←n**	0	0	?
p2: n←temp+1	q2: n←temp+1	0	0	0
(end)	**q2: n←temp+1**	1	0	0
(end)	(end)	1	0	0

Justification of the atomic assumption

Conclusion: The final value of n could be 1 or 2, depending on which scenario occurs.

But, this is the outcome of the more realistic analysis.

So, the atomic assumption is justified if you design the algorithm to mimic the lower level (usually with a temp variable).

Definitions

Computation: A directed path through a graph starting from the initial state and ending in a halt state.

Scenario: A table representation of a computation.

Concurrency analysis

Specify which statements are atomic.
Assume arbitrary interleaving of atomic statements.
Is the algorithm correct for all interleavings?

Correctness

Correctness must be proved.
Exhaustive testing is difficult, if not impossible.
Some concurrent algorithms are designed to be non-terminating.

Safety property, P

“Always”
P must be true in every state in every computation.

Safety property, P

"Always"
P must be true in <u>every state</u> in every computation.

Liveness property, P

"Eventually"
In every computation, there is <u>some state</u> in which P is true.

Duality

If P is a safety property, then ¬P is a liveness property.

Duality

If P is a safety property, then ¬P is a liveness property.

$$\neg(\forall x \mid R : P) \equiv (\exists x \mid R : \neg P)$$

Duality

If P is a safety property, then ¬P is a liveness property.

$$\neg(\forall x \mid R : P) \equiv (\exists x \mid R : \neg P)$$

If P is a liveness property, then ¬P is a safety property.

Duality

If P is a safety property, then ¬P is a liveness property.

$$\neg(\forall x \mid R : P) \equiv (\exists x \mid R : \neg P)$$

If P is a liveness property, then ¬P is a safety property.

$$\neg(\exists x \mid R : P) \equiv (\forall x \mid R : \neg P)$$

Safety examples

Vending machine: It is <u>always</u> true that if no money is inserted, no drink is dispensed.

Safety examples

Vending machine: It is <u>always</u> true that if no money is inserted, no drink is dispensed.

Star wars defense: It is <u>always</u> true that a missile is never launched unless the launch button is pressed.

Safety examples

Vending machine: It is <u>always</u> true that if no money is inserted, no drink is dispensed.

Star wars defense: It is <u>always</u> true that a missile is never launched unless the launch button is pressed.

Safety usually rules out bad behavior.

Liveness examples

Vending machine: If enough money is in the machine, a drink will <u>eventually</u> be dispensed.

Liveness examples

Vending machine: If enough money is in the machine, a drink will <u>eventually</u> be dispensed.

Star wars defense: If the launch button is pressed, the missile will <u>eventually</u> be launched.

Liveness examples

Vending machine: If enough money is in the machine, a drink will eventually be dispensed.

Star wars defense: If the launch button is pressed, the missile will eventually be launched.

Liveness ensures that the system does what it is supposed to do.

Definition

Weakly fair: A scenario is weakly fair if, at any state in the scenario, a statement that is continually enabled, eventually appears in the scenario.

Question

Does Algorithm 2.5 (next slide) necessarily stop?

Algorithm 2.5: Stop the loop A	
integer n ← 0	
boolean flag ← false	
p	**q**
p1: while flag = false	q1: flag ← true
p2: n ← 1 − n	

Answer: No

There is a scenario for which it never stops:

Answer: No

There is a scenario for which it never stops:

p1, p2, p1, p2, p1, p2, p1, p2, p1, p2, p1, p2, p1, p2, ...

Answer: No

There is a scenario for which it never stops:

p1, p2, p1, p2, p1, p2, p1, p2, p1, p2, p1, p2, p1, p2, ...

q1 is continually enabled, but does not appear in the scenario.

Answer: No

There is a scenario for which it never stops:

p1, p2, p1, p2, p1, p2, p1, p2, p1, p2, p1, p2, p1, p2, ...

q1 is continually enabled, but does not appear in the scenario.

Therefore, the scenario is not weakly fair.

Weak fairness

If the operating system can assure weak fairness, then Algorithm 2.5 is guaranteed to terminate.

So, fairness depends on the scheduling policy of the operating system.

Critical reference

Variable v is a critical reference if
(a) it is assigned in one process and has an occurrence in another process,
or
(b) it occurs in an expression in one process and is assigned in another.

Limited critical reference (LCR)

A program satisfies LCR if each statement contains at most one critical reference.

Example: Algorithm 2.3

Algorithm 2.3: Atomic assignment statements	
integer n $\leftarrow$ 0	
p	**q**
p1: n $\leftarrow$ n + 1	q1: n $\leftarrow$ n + 1

Example: Algorithm 2.3

Algorithm 2.3: Atomic assignment statements	
integer n ← 0	
p	**q**
p1: n ← n + 1	q1: n ← n + 1

Critical reference

Example: Algorithm 2.3

Algorithm 2.3: Atomic assignment statements	
integer n ← 0	
p	**q**
p1: n ← n + 1	q1: n ← n + 1

Critical reference

Example: Algorithm 2.3

Algorithm 2.3: Atomic assignment statements	
integer n $\leftarrow$ 0	
p	**q**
p1: n $\leftarrow$ n + 1	q1: n $\leftarrow$ n + 1

Critical reference

Critical reference

Example: Algorithm 2.3

Algorithm 2.3: Atomic assignment statements	
integer n ← 0	
p	**q**
p1: n ← n + 1	q1: n ← n + 1

Critical reference

Critical reference

Conclusion: Algorithm 2.3 does <u>not</u> satisfy LCR.

Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Not critical (temp in q is a different temp)

Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Not critical (temp in q is a different temp)

Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Not critical (temp in q is a different temp)

Critical reference

Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Critical reference

Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Critical reference

Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Critical reference

Not critical

Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Critical reference

Not critical

Conclusion: Algorithm 2.4 <u>does</u> satisfy LCR.

Limited critical reference

If an algorithm satisfies LCR, then it behaves the same regardless of whether its statements are atomic!

Then, you do not need to modify your algorithm with temp to mimic the lower level.

Algorithm 2.8: Volatile variables			
integer n ← 0			
p		**q**	
	integer local1, local2		integer local
p1:	n ← *some expression*	q1:	local ← n + 6
p2:	*computation not using n*	q2:	
p3:	local1 ← (n + 5) $*$ 7	q3:	
p4:	local2 ← n + 5	q4:	
p5:	n ← local1 * local2	q5:	

Volatile variables

An optimizing compiler could translate the statements in process p, Algorithm 2.8, as follows:

p1: tempReg1 ← some expression
p2: computation not using n
p3: tempReg2 ← tempReg1 + 5
p4: local2 ← tempReg2
p5: local1 ← tempReg2 * 7
p6: n ← local1 * local2

Volatile variables

The optimizing compiler does not assign to n in the first statement. Original statements p3 and p4 are executed out of order.

If there were no concurrency, the translated code would be correct.

With concurrency and interleaving, any translated code might not be correct.

Volatile variables

Specifying a variable as *volatile* instructs the compiler to load and store the value of the variable at each use, rather than to optimize away these loads and stores.

Concurrency in C++

`CountA.cpp`

Uses class `thread`.

Passes a function as a parameter to the constructor that the thread executes.

Functional programming!

`p.join()` forces `main()` to suspend execution until `p` terminates.

CountA.cpp

```
#include <cstdlib>
#include <iostream>
#include <thread>
using namespace std;

volatile int n = 0;

void pRun(int m) {
    int temp;
    for (int i = 0; i < m; i++) {
        temp = n;
        n = temp + 1;
    }
}

void qRun(int m) {
    int temp;
    for (int i = 0; i < m; i++) {
        temp = n;
        n = temp + 1;
    }
}
```

CountA.cpp

```
int main(int argc, char **argv) {
    int myMax = stoi(argv[1]);
    cout << "The value of myMax is " << myMax << endl;
    thread p(pRun, myMax);
    thread q(qRun, myMax);
    p.join();
    q.join();
    cout << "The value of n should be " << 2*myMax << endl;
    cout << "The value of n is " << n << endl;
    return EXIT_SUCCESS;
}
```

Function syntax

Think of the statement

```
thread p(pRun, myMax);
```

as if it were

```
thread p(pRun(myMax));
```

where `myMax` is the actual parameter that corresponds to the formal parameter `m`.

Demo `CountA.cpp`

Take `main()` input from command line or from CLion Program arguments in Run/Debug Configuration.

Conclusion: The program works for small values of `m`, but not for large values of `m`.

Why?

Demo `CountA.cpp`

Because for small values of `m` each thread will complete its entire computation within a single time slice.

Therefore, no interleaving!

Concurrency in C++

`CountB.cpp`

Uses a random delay to force interleaving due to time slice timeouts even for small values of `m`.

CountB.cpp

```
#include <cstdlib>
#include <iostream>
#include <thread>
#include "Util450.cpp"
using namespace std;

volatile int n = 0;

void pRun() {
    int temp;
    for (int i = 0; i < 10; i++) {
        randomDelay(10);
        temp = n;
        randomDelay(10);
        n = temp + 1;
    }
}
```

CountB.cpp

```
void qRun() {
    int temp;
    for (int i = 0; i < 10; i++) {
        randomDelay(10);
        temp = n;
        randomDelay(10);
        n = temp + 1;
    }
}

int main(int argc, char **argv) {
    thread p(pRun);
    thread q(qRun);
    p.join();
    q.join();
    cout << "The value of n is " << n << endl;
    return EXIT_SUCCESS;
}
```

Util450.cpp

```
#include <thread>
#include <chrono>
#include <random>
#include <iostream>

using namespace std;

random_device rdev{}; // For random seed
default_random_engine engine{rdev()}; // Seed the engine

void randomDelay(int delay) {
    uniform_int_distribution<int> distr(0, delay);
    int d = distr(engine);
    // cout << "delay == " << d << endl;
    this_thread::sleep_for(chrono::milliseconds(d));
}
```

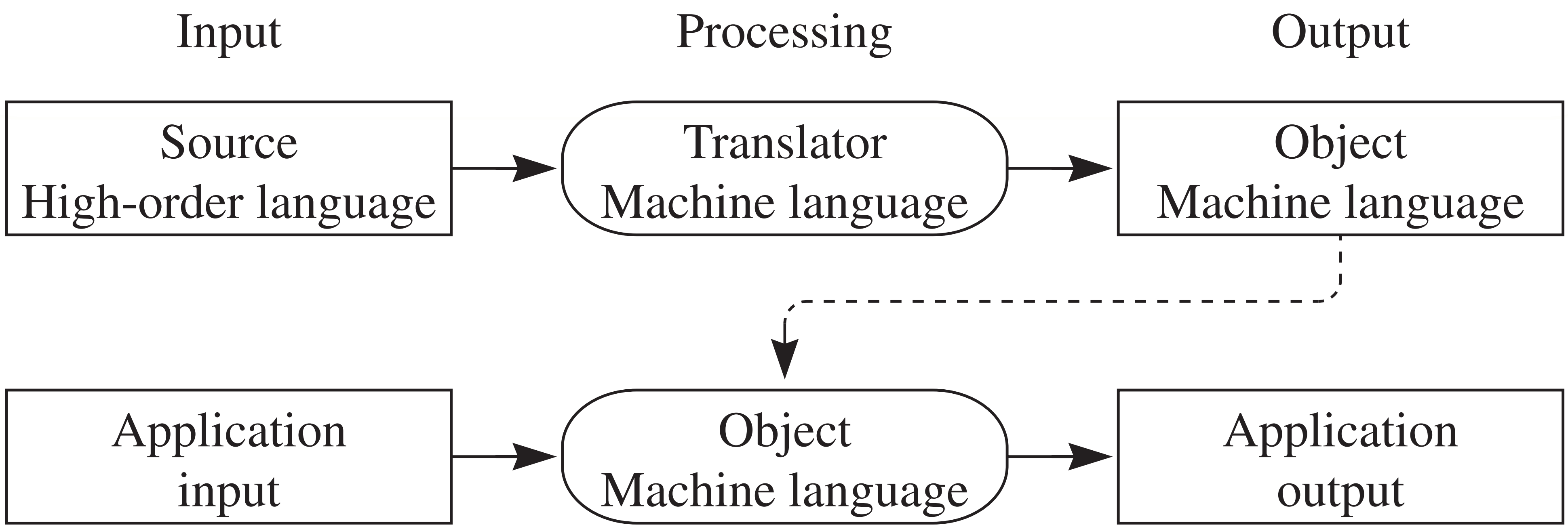
Demo `CountB.cpp`

Results are unpredictable because of the random delays.

To see the delays, repeat demo with `cout` uncommented in `randomDelay()`.

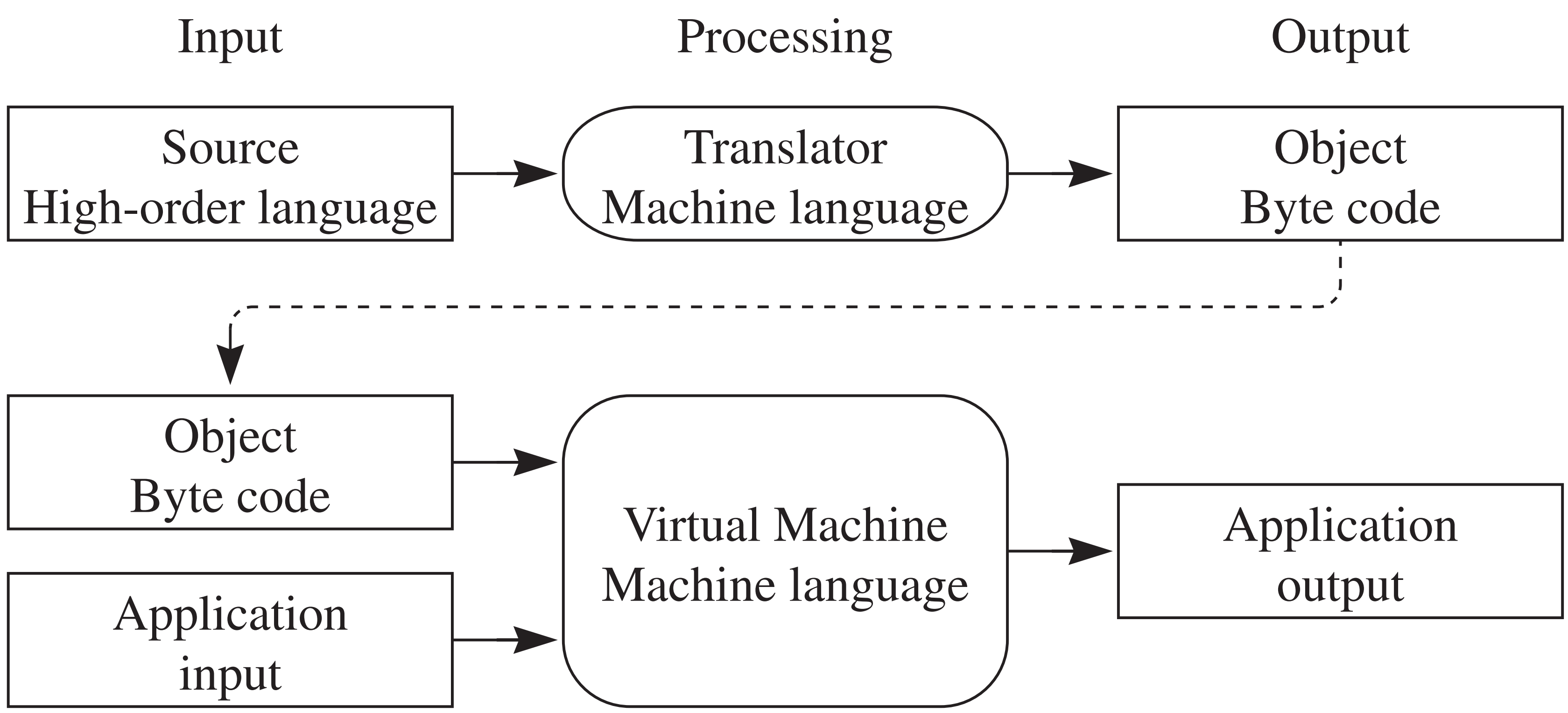
Interleaving may occur within the `cout` streams!

Compilation, C++



(**a**) Compilation.

Interpretation, Java



(b) Interpretation.

Java

Use IntelliJ IDE.

Modify Ben-Ari and Sestoft Java programs to make project class public for two reasons:

- Allows JavaDoc html documentation.
- Allows to execute the class file from the command line.

Must rename class to match file name.

Java

`p.start()` puts thread p in the ready (Enabled) queue.

`p.join()`, executed by `main()`, suspends `main()` until thread `p` terminates.

CountA.java

```java
package counta;

public class CountA extends Thread {

    static volatile int n = 0;
    int m;

    CountA(int myM) {
        m = myM;
    }

    public void run() {
        int temp;
        for (int i = 0; i < m; i++) {
            temp = n;
            n = temp + 1;
        }
    }
```

CountA.java

```java
    public static void main(String[] args) {
        int myMax = Integer.parseInt(args[0]);
        System.out.println("The value of myMax is " + myMax);
        CountA p = new CountA(myMax);
        CountA q = new CountA(myMax);
        p.start();
        q.start();
        try {
            p.join();
            q.join();
        } catch (InterruptedException e) {
        }
        System.out.println(
            "The value of n should be " + 2 * myMax);
        System.out.println("The value of n is " + n);
    }
}
```

Demo `CountA.java`

Take `main()` input from command line or from IntelliJ Program arguments in Run/Debug Configuration.

Conclusion: As with C++, the program works for small values of `m`, but not for large values of `m`.

Command line: Compilation vs Interpretation

```
warford$ ./CountA 10
```

Execute the machine language app

Command line: Compilation vs Interpretation

```
warford$ ./CountA 10
```

Execute the machine language app

```
warford$ java counta/CountA 10
```

The app is input to the Java virtual machine

Execute the Java virtual machine

`CountB.java`

Similar to `CountB.cpp`

Must put `randomDelay()` in a `try` statement because an exception is possible.

Insert random delays that make multiple runs <u>not</u> predictable.

The random sleep delays are long enough to trigger a timeout, which forces interleaving to occur.

<u>CountB.java</u>

```
package countb;

import static util450.Util450.*;

public class CountB extends Thread {

    static volatile int n = 0;

    public void run() {
        int temp;
        for (int i = 0; i < 10; i++) {
            try {
                randomDelay(10);
                temp = n;
                randomDelay(10);
                n = temp + 1;
            } catch (InterruptedException e) {
            }
        }
    }
```

<u>CountB.java</u>

```
   public static void main(String[] args) {
      CountB p = new CountB();
      CountB q = new CountB();
      p.start();
      q.start();
      try {
         p.join();
         q.join();
      } catch (InterruptedException e) {
      }
      System.out.println("The value of n is " + n);
   }
}
```

<u>Util450.java</u>

```
package util450;

public final class Util450 {

   public static void randomDelay(int delay)
         throws InterruptedException {
      int d = (int) (delay * Math.random());
      // System.out.println("delay == " + d);
      Thread.sleep(d); // milliseconds
   }
}
```

Threads

Like processes, threads are also programs during execution.

However, a thread is under control of a process.

A process is under control of the operating system.

Demo Activity Monitor application.

Threads vs Processes

A process is a program during execution <u>in an operating system</u>.

Threads vs Processes

A process is a program during execution in an operating system.

- Processes communicate via message passing.

Threads vs Processes

A process is a program during execution in an operating system.

- Processes communicate via message passing.

A thread is a program during execution in a process.

Threads vs Processes

A process is a program during execution in an operating system.

- Processes communicate via message passing.

A thread is a program during execution in a process.

- Threads communicate via shared memory.

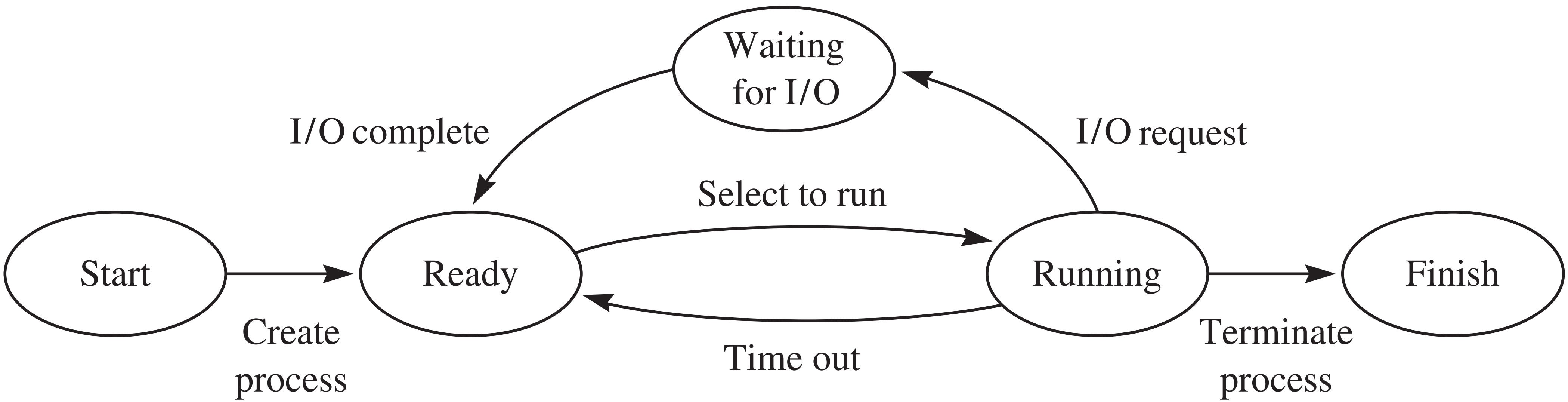
The action of `p.start()` and `p.join()`

(Sestoft 20.1, page 80)

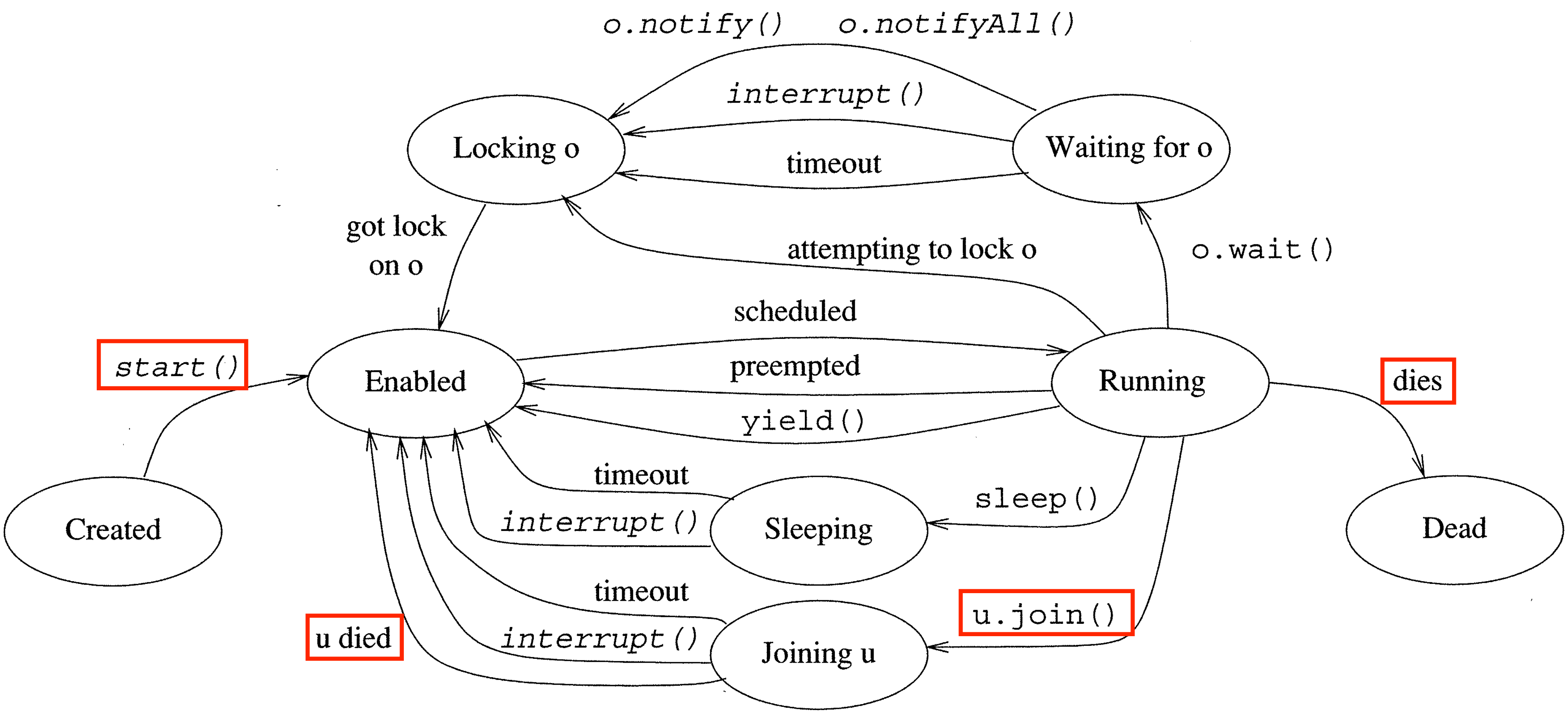
Let `u` be a thread (an object of a subclass of `Thread`).

`u.start()` changes the state of `u` to `Enabled` so that its `run()` method will be called when a processor becomes available.

`u.join()` waits for thread `u` to die; may throw `InterruptedException` if the current thread is interrupted while waiting.


Waiting
for I/O
I/O complete
I/O request
Select to run
Start
Ready
Running
Finish
Create
process
Time out
Terminate
process

Sestoft, page 81



The action of `p.start()` and `p.join()`

After `p` and `q` start, there could be three concurrent executions:

1. `p` executing its `run()` method
2. `q` executing its `run()` method
3. `main()` executing its statements after starting `q`

`p.join()`is not executed by thread `p`. It is executed by `main()`.

The `throw` statement

`throw` *expression* `;`

The type of *expression* must be a subtype of class `Throwable`.

The enclosing block statement terminates abruptly. The thrown exception may be caught by a `try-catch` statement.

Class hierarchy (partial)

```
Throwable
   Error
      OutOfMemoryError
   Exception
      IOException
      RuntimeException
         ArithmeticException
         IndexOutOfBoundsException
            ArrayIndexOutOfBoundsException
            StringIndexOutOfBoundsException
         NegativeArraySizeException
```

The `try-catch-finally` statement

```
try
    body
    catch(E1 x1)
        catchBody1
    catch(E2 x2)
        catchBody2
    ...
    finally
        finallyBody
```

`try-catch` with no `finally`

```
try {
    A
    B
    C
    D
} catch(E1 x1) {
        F
        G
}
```

`try-catch` with no `finally`

```
try {
    A
    B
    C
    D
} catch(E1 x1) {
        F
        G
}
```

Sequence with no exception

A B C D

`try-catch` with no `finally`

```
try {
    A
    B
    C
    D
} catch(E1 x1) {
       F
       G
}
```

Sequence with no exception

`A B C D`

Sequence with exception at `B`

`A B F G`

Sestoft, `Example89.java`

Uses Sestoft, `Example98.java`

`super()` is a call to the superclass constructor.

Passing a string to the constructor of a superclass causes `toString()` to append the string to the name of the exception.

Sestoft, Example98.java

```
package example89;

class WeekdayException extends Exception {

   public WeekdayException(String wday) {
      super("Illegal weekday: " + wday);
   }
}
```

Sestoft, <u>Example89.java</u>

```
package example89;

public class Example89 {
   public static void main(String[] args) {
      try {
         System.out.println(args[0]
            + " is weekday number " + wdayno4(args[0]));
      } catch (WeekdayException x) {
         System.out.println("Weekday problem: " + x);
      } catch (Exception x) {
         System.out.println("Other problem: " + x);
      }
   }
```

Sestoft, Example89.java

```
// Sestoft, Example 88
static int wdayno4(String wday) throws WeekdayException {
   for (int i = 0; i < wdays.length; i++)
      if (wday.equals(wdays[i]))
         return i + 1;
   throw new WeekdayException(wday);
}

// Sestoft, Example 80
static final String[] wdays =
   { "Monday", “Tuesday”, “Wednesday", “Thursday",
     “Friday", "Saturday", "Sunday" };
}
```

Sestoft, `Example89.java`

Demo with

```
$ java Example89 Wednesday
```

Demo with

```
$ java Example89 Wedxxx
```

Demo with

```
$ java Example89
```

`throws`

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.

You do this by including a `throws` clause in the method's declaration.

`ExceptionReview`

Uncaught exceptions propagate up the call chain.

Demo with

```
$ java ExceptionReview 10
```

Demo with

```
$ java ExceptionReview -10
```

MyException.java

```java
public class MyException extends RuntimeException {

    public MyException(String message) {
        super(message);
    }
}
```

ExceptionReview.java

```
public class ExceptionReview {
   public static void main(String[] args) {
      int x = Integer.parseInt(args[0]);
      System.out.println("Main started with x == " + x);
      int x2;
      try {
         x2 = top(x);
      } catch (MyException e) {
         System.out.println("Caught an exception: " + e);
         x2 = 99;
      }
      System.out.println("Main ending with x2 == " + x2);
   }
   static int top(int y) {
      System.out.println("Top started with y == " + y);
      int y2 = bottom(y);
      System.out.println("Top returning y2 == " + y2);
      return y2;
   }
   static int bottom(int z) throws MyException {
      System.out.println("Bottom started with z == " + z);
      if (z < 0) {
         throw new MyException("Throwing MyException");
      }
      System.out.println("Bottom returning 20");
      return 20;
   }
}
```

Sestoft, `Example99`

A clever example that can exercise all the possible flows of control through the `try-catch-finally` statement.

See Section 12.6.6, page 62, for a detailed explanation. The key sentence is the one about the `finally` clause:

The `try-catch-finally` statement

try
    body
    catch(E1 x1)
        catchBody1
    catch(E2 x2)
        catchBody2
    ...
    finally
        finallyBody

If there is a `finally` clause, the *finallyBody* will be executed regardless of whether the execution of *body* terminated normally, regardless of whether *body* exited by executing `return` or `break` or `continue`, regardless of whether any exception thrown by *body* was caught by a `catch` clause, and regardless of whether the `catch` clause exited by executing `return` or `break` or `continue` or by throwing an exception.

Example99.java

```
// Example 99 from page 73 of Java Precisely third edition (The MIT Press 2016)
// Author: Peter Sestoft (sestoft@itu.dk)

// To exercise all paths through the try-catch-finally statement in
// method m, run this program with each of these arguments:
// 101 102 103 201 202 203 301 302 303 411 412 413 421 422 423 431 432 433
// like this:
//    java Example99 101
//    java Example99 102
//    etc

class Example99 {
   public static void main(String[] args) throws Exception {
      System.out.println(m(Integer.parseInt(args[0])));
   }
```

Example99.java

```
    static String m(int a) throws Exception {
        try {
            System.out.print("try ... ");
            if (a / 100 == 2)
                return "returned from try";
            if (a / 100 == 3)
                throw new Exception("thrown by try");
            if (a / 100 == 4)
                throw new RuntimeException("thrown by try");
        } catch (RuntimeException x) {
            System.out.print("catch ... ");
            if (a / 10 % 10 == 2)
                return "returned from catch";
            if (a / 10 % 10 == 3)
                throw new Exception("thrown by catch");
        } finally {
            System.out.println("finally");
            if (a % 10 == 2)
                return "returned from finally";
            if (a % 10 == 3)
                throw new Exception("thrown by finally");
        }
        return "terminated normally with " + a;
    }
}
```