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

\[ p_1 \rightarrow q_1 \rightarrow p_2 \rightarrow q_2, \]
\[ p_1 \rightarrow q_1 \rightarrow q_2 \rightarrow p_2, \]
\[ p_1 \rightarrow p_2 \rightarrow q_1 \rightarrow q_2, \]
\[ q_1 \rightarrow p_1 \rightarrow q_2 \rightarrow p_2, \]
\[ q_1 \rightarrow p_1 \rightarrow p_2 \rightarrow q_2, \]
\[ q_1 \rightarrow q_2 \rightarrow p_1 \rightarrow p_2. \]
**Algorithm 2.1: Trivial concurrent program**

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>integer n ← 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>integer k1 ← 1</td>
<td>integer k2 ← 2</td>
</tr>
<tr>
<td>p₁</td>
<td>n ← k₁</td>
<td>q₁: n ← k₂</td>
</tr>
<tr>
<td></td>
<td>p₁: n ← k₁</td>
<td></td>
</tr>
</tbody>
</table>
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

p1: \( n \leftarrow k1 \)
q1: \( n \leftarrow k2 \)
\( k1 = 1, k2 = 2 \)
\( n = 0 \)

\( k1 = 1, k2 = 2 \)
\( n = 1 \)

\( k1 = 1, k2 = 2 \)
\( n = 2 \)

\( k1 = 1, k2 = 2 \)
\( n = 2 \)

\( k1 = 1, k2 = 2 \)
\( n = 1 \)
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

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer temp</td>
<td>integer temp</td>
</tr>
<tr>
<td>p1:  do 10 times</td>
<td>q1:  do 10 times</td>
</tr>
<tr>
<td>p2:  temp ← n</td>
<td>q2:  temp ← n</td>
</tr>
<tr>
<td>p3:  n ← temp + 1</td>
<td>q3:  n ← temp + 1</td>
</tr>
</tbody>
</table>

integer n ← 0
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.

\[ n = k + 1; \]

\[ \text{LDA } k, s \]
\[ \text{ADDA } 1, i \]
\[ \text{STA } n, d \]
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).

\[
\begin{align*}
\text{integer } n &\leftarrow 0 \\
\text{p} &\begin{align*}
\text{integer } n &\leftarrow k_1 \\
p_1 &:\text{LDA } k_1, s \\
p_2 &:\text{STA } n, d
\end{align*}
\end{align*}
\begin{align*}
\text{q} &\begin{align*}
\text{integer } n &\leftarrow k_2 \\
q_1 &:\text{LDA } k_2, s \\
q_2 &:\text{STA } n, d
\end{align*}
\end{align*}
\]
Justification of The Concurrency Theorem

Suppose in a multiprocessing system, one CPU tries to execute \texttt{p2: STA n, d} at the same time another CPU tries to execute \texttt{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

Global memory

0000 0000 0000 0011

Local memory

0000 0000 0000 0001

Local memory

0000 0000 0000 0010
Justification of The Concurrency Theorem

Conclusion:

The effect is the same as if an arbitrary interleaving happens in a multiprogramming 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

<table>
<thead>
<tr>
<th></th>
<th>( p )</th>
<th>( q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p1 ):</td>
<td>( n \leftarrow n + 1 )</td>
<td>( q1 ):</td>
</tr>
</tbody>
</table>

integer \( n \leftarrow 0 \)
Scenario for Atomic Assignment Statements

<table>
<thead>
<tr>
<th>Process p</th>
<th>Process q</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: n←n+1</td>
<td>q1: n←n+1</td>
<td>0</td>
</tr>
<tr>
<td>(end)</td>
<td>q1: n←n+1</td>
<td>1</td>
</tr>
<tr>
<td>(end)</td>
<td>(end)</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process p</th>
<th>Process q</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: n←n+1</td>
<td>q1: n←n+1</td>
<td>0</td>
</tr>
<tr>
<td>p1: n←n+1</td>
<td>(end)</td>
<td>1</td>
</tr>
<tr>
<td>(end)</td>
<td>(end)</td>
<td>2</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1:</td>
<td>load R1,n</td>
<td>q1: load R1,n</td>
</tr>
<tr>
<td>p2:</td>
<td>add R1,#1</td>
<td>q2: add R1,#1</td>
</tr>
<tr>
<td>p3:</td>
<td>store R1,n</td>
<td>q3: store R1,n</td>
</tr>
</tbody>
</table>

integer n ← 0
Register Machine

Memory

Load

Registers

0

0

... 0 ...

Memory

... 0 ...

Registers

1

1

... 1 ...

Memory

... 1 ...

Registers

1

1

... 1 ...

Load Store
### Scenario for a Register Machine

<table>
<thead>
<tr>
<th>Process p</th>
<th>Process q</th>
<th>n</th>
<th>p.R1</th>
<th>q.R1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>p1: load R1,n</strong></td>
<td>q1: load R1,n</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td><strong>p2: add R1,#1</strong></td>
<td>q1: load R1,n</td>
<td>0</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td><strong>p2: add R1,#1</strong></td>
<td>q2: add R1,#1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>p3: store R1,n</strong></td>
<td>q2: add R1,#1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>p3: store R1,n</strong></td>
<td>q3: store R1,n</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(end)</td>
<td>q3: store R1,n</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(end)</td>
<td>(end)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Justification of the atomic assumption

Even though the results are different depending on whether we make the atomic assumption, we can still model 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

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>integer temp</td>
<td>integer temp</td>
</tr>
<tr>
<td>p1:</td>
<td>temp ← n</td>
<td>q1:</td>
</tr>
<tr>
<td>p2:</td>
<td>n ← temp + 1</td>
<td>q2:</td>
</tr>
</tbody>
</table>

integer n ← 0
Correct Scenario for Assignment Statements

<table>
<thead>
<tr>
<th>Process p</th>
<th>Process q</th>
<th>n</th>
<th>p.temp</th>
<th>q.temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: temp ← n</td>
<td>q1: temp ← n</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>p2: n ← temp + 1</td>
<td>q1: temp ← n</td>
<td>0</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>(end)</td>
<td>q1: temp ← n</td>
<td>1</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>(end)</td>
<td>q2: n ← temp + 1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(end)</td>
<td>(end)</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
**Incorrect Scenario for Assignment Statements**

<table>
<thead>
<tr>
<th>Process p</th>
<th>Process q</th>
<th>n</th>
<th>p.temp</th>
<th>q.temp</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>p1: temp ← n</strong></td>
<td>q1: temp ← n</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td><strong>p2: n ← temp + 1</strong></td>
<td><strong>q1: temp ← n</strong></td>
<td>0</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td><strong>p2: n ← temp + 1</strong></td>
<td><strong>q2: n ← temp + 1</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(end)</td>
<td><strong>q2: n ← temp + 1</strong></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(end)</td>
<td>(end)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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 every state in every computation.

Liveness property, \( P \)

“Eventually”
In every computation, there is some state in which \( P \) is true.
Duality

If $P$ is a safety property, then $\neg P$ is a liveness property.
Duality

If $P$ is a safety property, then $\neg P$ is a liveness property.

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

If P is a safety property, then $\neg P$ is a liveness property.

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

If P is a liveness property, then $\neg P$ 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 always true that if no money is inserted, no drink is dispensed.
Safety examples

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

Star wars defense: It is always true that a missile is never launched unless the launch button is pressed.
Safety examples

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

Star wars defense: It is always 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 *eventually* be dispensed.
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 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

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: while flag = false</td>
<td>q1: flag ← true</td>
</tr>
<tr>
<td>p2: n ← 1 − n</td>
<td></td>
</tr>
</tbody>
</table>
Answer: No

There is a scenario for which it never stops:
Answer: No

There is a scenario for which it never stops:

\[ p_1, p_2, p_1, p_2, p_1, p_2, p_1, p_2, p_1, p_2, p_1, p_2, \ldots \]
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:

\[ p_1, p_2, p_1, p_2, p_1, p_2, p_1, p_2, p_1, p_2, p_1, p_2, ... \]

\[ q_1 \] 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

<table>
<thead>
<tr>
<th>Algorithm 2.3: Atomic assignment statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer ( n \leftarrow 0 )</td>
</tr>
<tr>
<td>( p )</td>
</tr>
<tr>
<td>( q )</td>
</tr>
<tr>
<td>( p_1 : n \leftarrow n + 1 )</td>
</tr>
<tr>
<td>( q_1 : n \leftarrow n + 1 )</td>
</tr>
</tbody>
</table>
### Example: Algorithm 2.3

<table>
<thead>
<tr>
<th>Algorithm 2.3: Atomic assignment statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer n ← 0</td>
</tr>
<tr>
<td>p</td>
</tr>
<tr>
<td>p1: n ← n + 1</td>
</tr>
<tr>
<td>q</td>
</tr>
<tr>
<td>q1: n ← n + 1</td>
</tr>
</tbody>
</table>

**Critical reference**

**Example: Algorithm 2.3**

**Algorithm 2.3: Atomic assignment statements**

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>p1:</strong></td>
<td>n ← n + 1</td>
<td>q1: n ← n + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Critical reference**
### Example: Algorithm 2.3

**Algorithm 2.3: Atomic assignment statements**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>q</td>
</tr>
<tr>
<td>p1:</td>
<td>q1:</td>
</tr>
<tr>
<td>n ← n + 1</td>
<td>n ← n + 1</td>
</tr>
</tbody>
</table>

integer n ← 0

Critical reference
Example: Algorithm 2.3

**Algorithm 2.3: Atomic assignment statements**

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>n ← n + 1</td>
<td>q1: n ← n + 1</td>
</tr>
</tbody>
</table>

**Conclusion:** Algorithm 2.3 does not satisfy LCR.
### Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer temp</td>
<td>integer temp</td>
</tr>
<tr>
<td>p1: temp ← n</td>
<td>q1: temp ← n</td>
</tr>
<tr>
<td>p2: n ← temp + 1</td>
<td>q2: n ← temp + 1</td>
</tr>
</tbody>
</table>
Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer temp</td>
<td>integer temp</td>
</tr>
<tr>
<td>p1: temp ← n</td>
<td>q1: temp ← n</td>
</tr>
<tr>
<td>p2: n ← temp + 1</td>
<td>q2: n ← temp + 1</td>
</tr>
</tbody>
</table>

Not critical (temp in q is a different temp)
Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer temp</td>
<td>p1: temp ← n</td>
<td>q1: temp ← n</td>
</tr>
<tr>
<td></td>
<td>p2: n ← temp + 1</td>
<td>q2: n ← temp + 1</td>
</tr>
</tbody>
</table>

Not critical (temp in q is a different temp)
Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer n ← 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p1:</td>
<td>temp ← n</td>
<td>q1:</td>
</tr>
<tr>
<td>p2:</td>
<td>n ← temp + 1</td>
<td>q2:</td>
</tr>
</tbody>
</table>

Integer temp

Not critical (temp in q is a different temp)

Critical reference
**Example: Algorithm 2.4**

<table>
<thead>
<tr>
<th>Algorithm 2.4: Assignment statements with one global reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer n ← 0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>integer temp</td>
</tr>
<tr>
<td>p1: temp ← n</td>
</tr>
<tr>
<td>p2: n ← temp + 1</td>
</tr>
</tbody>
</table>

Example: Algorithm 2.4

<table>
<thead>
<tr>
<th>Algorithm 2.4: Assignment statements with one global reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer n ← 0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>p</strong></td>
</tr>
<tr>
<td>integer temp</td>
</tr>
<tr>
<td>p1: temp ← n</td>
</tr>
<tr>
<td>p2: n ← temp + 1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>q</strong></td>
</tr>
<tr>
<td>integer temp</td>
</tr>
<tr>
<td>q1: temp ← n</td>
</tr>
<tr>
<td>q2: n ← temp + 1</td>
</tr>
</tbody>
</table>

Critical reference
Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer n ← 0</td>
<td>integer temp</td>
</tr>
<tr>
<td>integer temp</td>
<td>integer temp</td>
</tr>
<tr>
<td>p1: temp ← n</td>
<td>q1: temp ← n</td>
</tr>
<tr>
<td>p2: n ← temp + 1</td>
<td>q2: n ← temp + 1</td>
</tr>
</tbody>
</table>

Critical reference
### Algorithm 2.4: Assignment statements with one global reference

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>n ← 0</td>
<td></td>
</tr>
<tr>
<td>p1:</td>
<td>temp ← n</td>
<td>q1: temp ← n</td>
</tr>
<tr>
<td>p2:</td>
<td>n ← temp + 1</td>
<td>q2: n ← temp + 1</td>
</tr>
</tbody>
</table>

Critical reference

Not critical
Example: Algorithm 2.4

Algorithm 2.4: Assignment statements with one global reference

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>integer n ← 0</td>
<td></td>
</tr>
<tr>
<td>p1</td>
<td>integer temp</td>
<td>integer temp</td>
</tr>
</tbody>
</table>
| p2     | temp ← n              | q1: temp ← n
|        | n ← temp + 1          | q2: n ← temp + 1      |

Critical reference

Not critical

Conclusion: Algorithm 2.4 does 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

```plaintext

type integer

integer n ← 0

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer local1, local2</td>
<td>integer local</td>
</tr>
<tr>
<td>p1: n ← some expression</td>
<td>q1: local ← n + 6</td>
</tr>
<tr>
<td>p2: computation not using n</td>
<td>q2:</td>
</tr>
<tr>
<td>p3: local1 ← (n + 5) * 7</td>
<td>q3:</td>
</tr>
<tr>
<td>p4: local2 ← n + 5</td>
<td>q4:</td>
</tr>
<tr>
<td>p5: n ← local1 * local2</td>
<td>q5:</td>
</tr>
</tbody>
</table>
```

Volatile variables

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

\begin{align*}
    p1 &: \text{tempReg1 } \leftarrow \text{some expression} \\
    p2 &: \text{computation not using } n \\
    p3 &: \text{tempReg2 } \leftarrow \text{tempReg1 } + 5 \\
    p4 &: \text{local2 } \leftarrow \text{tempReg2} \\
    p5 &: \text{local1 } \leftarrow \text{tempReg2 } * 7 \\
    p6 &: \text{n } \leftarrow \text{local1 } * \text{local2}
\end{align*}
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.
#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;
    }
}
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;
    }
}
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;
}
#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!
Similar to that of C, and it has the advantage of being object-oriented. Java provides an extensive library of graphical user interface (GUI) elements for input and output. The programs in this chapter get their input as a string of terminal characters from a single input window and send the results of the translation to the standard output window.

Part (a) shows the translation process for a compiled language like C. Every run in the computation process executes a machine language program with input and output. In the first run, a C compiler converts the source code in a high-level language to the object code in machine language. In the second run, the machine language object code executes, processing the application input and producing the application output.

Part (b) shows the translation process for an interpreted language like Java and Perl, both of which are based on virtual machines. In the first interpretation, the virtual machine executes the byte code, which is a high-level representation of the source code. The virtual machine then produces the application output.
Java provides an extensive library of graphical user interface (GUI) elements for input and output. The programs in this chapter get their input as a string of terminal characters from a single input window and send the results of the translation to the standard output window. Java itself is an interpreted language based on the Java Virtual Machine (JVM). Figure 7.26 shows the difference between a compiled language and an interpreted language. Part (a) shows the translation process for a compiled language like C. Every run in the computation process executes a machine language program with input and output. In the first run, a C compiler converts the source code in a high-level language to the object code in machine language. In the second run, the machine language object code executes, processing the application input and producing the application output. Part (b) shows the translation process for an interpreted language like Java and Pep/9, both of which are based on virtual machines. In the first interpretation, Java.
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.
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

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 not predictable.

The random sleep delays are long enough to trigger a timeout, which forces interleaving to occur.
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) {
            }
        }
    }
}
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);
}
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 in an operating system.
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.
The PCB contains additional information to help the operating system schedule the CPU. An example is a unique process identification number assigned by the system, labeled Process ID in Figure 8.18, that serves to reference the process. Suppose a user wants to terminate a process before it completes execution normally, and he knows the ID number is 782. He could issue a `KILL(782)` command that would cause the operating system to search through the queue of PCBs, find the PCB with ID 782, remove it from the queue, and deallocate it.

Another example of information stored in the PCB is a record of the total amount of CPU time used so far by the suspended process. If the CPU becomes available and the operating system must decide which of several suspended processes gets the CPU, it can use the recorded time to make a fair decision.

As a job progresses through the system toward completion, it passes through several states, as Figure 8.19 shows. The figure is in the form of a state transition diagram and is another example of a finite state machine. Each transition is labeled with the event that causes the change of state.

![State Transition Diagram](image)

When a user submits a job for processing, the operating system creates a process for it by allocating a new PCB and attaching it to a queue of processes that are waiting for CPU time. It loads the program into main memory and sets the copy of `PC` in the PCB to the address of the first instruction of the process. That puts the job in the ready state.

Eventually, the operating system should select the job to receive some processing time. It sets the alarm clock to generate an interrupt after a quantum of time and puts the copies of the registers from the PCB into the CPU. That puts the job in the running state.

While in the running state, three things can happen: (1) The running process may time out if it is still executing when the alarm clock interrupts. If so, the operating system attaches the process's PCB to the ready queue, which puts it back in the ready state. (2) The process may complete its execution normally, in which case the last instruction it executes is an `SVC` to request that the operating system terminate it. (3) The process may need some input, in which case it executes an `SVC` for the request. The operating system would transfer the request to the appropriate I/O device and put the PCB in another queue of processes that are waiting for their I/O operations to complete. That puts the process in the waiting-for-I/O state.
Example

MultipleThreads

The main program creates a new thread, binds it to u, and starts it. Now two threads are executing concurrently: one executes main, and another executes run.

While the main method is blocked waiting for keyboard input, the new thread keeps incrementing i. The new thread executes yield to make sure that the other thread is allowed to run (when not blocked).

```java
class Incrementer extends Thread {
    public int i;
    public void run() {
        for (;;) {
            i++;
            yield();
        }
    }
}
```

```java
class Thendoemo {
    public static void main(String[] args) throws InterruptedException {
        Incrementer u = new Incrementer();
        u.start();
        System.out.println("Repeatedly press Enter to get the current value of i:");
        for (;;) {
            System.in.read();
            System.out.println(u.i);
        }
    }
}
```
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

```java
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`
      `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

```java
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
}
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.
package example89;

class WeekdayException extends Exception {
    public WeekdayException(String wday) {
        super("Illegal weekday: " + wday);
    }
}
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, 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" };

Demo with
$ java Example89 Wednesday

Demo with
$ java Example89 Wedxxx

Demo with
$ java Example89

Sestoft, Example89.java
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
public class MyException extends RuntimeException {

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

MyException.java
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;
    }
}
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

```java
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.
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;
}