50.003: Elements of Software Construction

Week 5

Basics of Concurrency: Requirements

Course Plan

Week	Cohort Class 1	Cohort Class 2	Cohort Class 3	Remarks
1 (Jan 28)	Software Development Process			
2 (Feb 4)	Software Design and UML		**Project Meeting I**	Problem Set 1
3 (Feb 11)	Software Design and UML			
4 (Feb 18)	Software Testing			Problem Set 2, Quiz 1
5 (Feb 25)	Basics of Concurrency		**Project Meeting II**	
6 (Mar 4)	Thread Safe Java Class			Problem Set 3
7 (Mar 11)	Recess			Problem Set 4
8 (Mar 18)	Testing Concurrent Programs		**Project Meeting III**	Quiz 2
9 (Mar 25)	Performance			Problem Set 5
10 (Apr 1)	Design Patterns			Problem Set 6; Quiz 3
11 (Apr 8)	Advanced Software Testing		**Project Meeting IV**	Problem Set 7
12 (Apr 15)	Software Debugging, Code smells and Maintenance			Quiz 4
13 (Apr 22)	**Final Project Presentation (15 minutes for each group)**			Project Report/Code Due
14 (May 2)	Final Exam			

Example

NSA intercepted a RSA-encrypted secrete message which tells the location of a terrorist act, we believe that the act is going to happen one week from now, we need your help in decrypting the message.

Requirement/Analysis

"Given a semi-prime, your program outputs its prime factors within 6 days."

green: pre-condition

red: post-condition

purple: non-functional requirement

Total Correctness: If the precondition is satisfied, it is guaranteed that the method terminates and satisfies the postcondition.

Design/Implementation

Design

- Use the trial division method
- http://en.wikipedia.org/wiki/Integer_factorization

Implementation

- FactorPrime.java

Testing

Correctness Testing

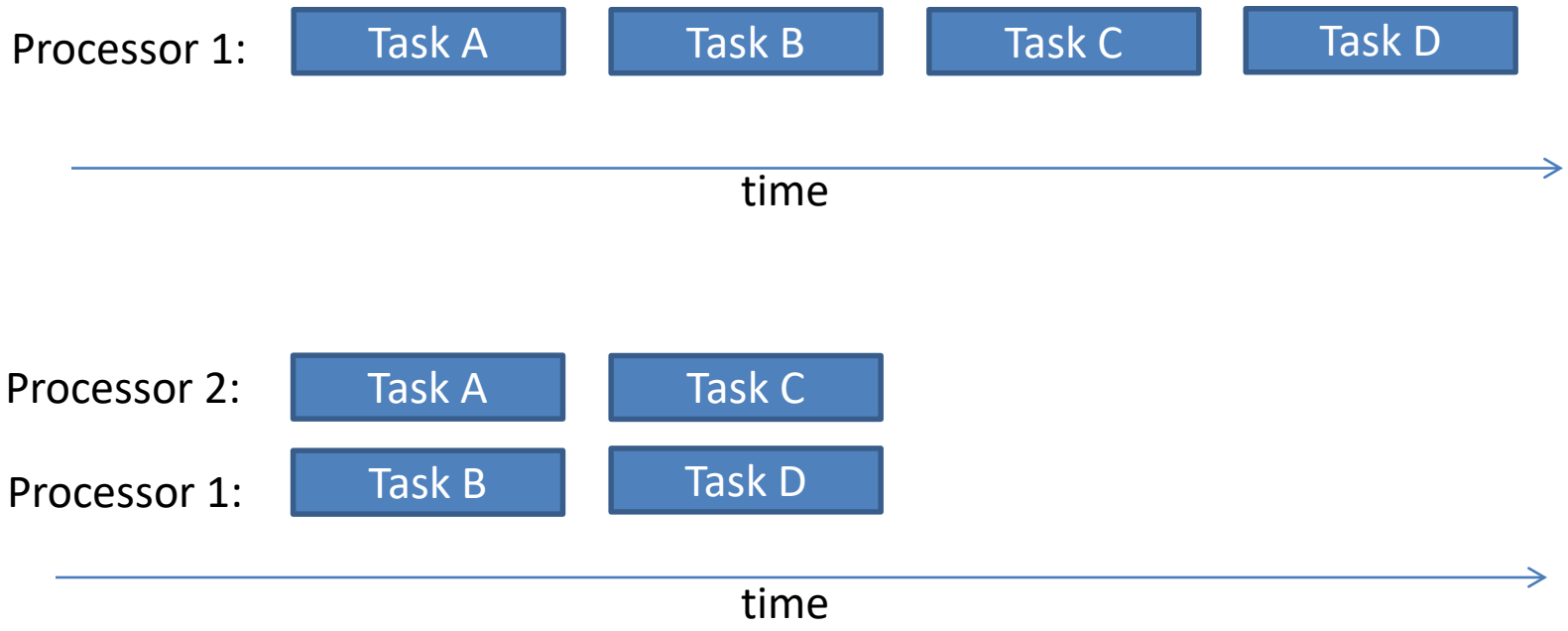
- 4294967297 (famous Fermat Number)
- 1127451830576035879
- 160731047637009729259688920385507056726966793490579598495689711866432421212774967029895340327197901
  756096014299132623454583177072050452755510701340673282385647899694083881316194642417451570483466327
  782135730575564856185546487053034404560063433614723836456790266457438831626375556854133866958349817
  172727462462516466898479574402841071703909138062456567624565784254101568378407242273207660892036869
  708190688033351601539401621576507964841597205952722487750670904522932328731530640706457382162644738
  538813247139315456213401586618820517823576427094125197001270350087878270889717445401145792231674098
  948416888868250143592026973853973785120217077951766546939577520897245392186547279572494177680291506
  578508962707934879124914880885500726439625033021936728949277390185399024276547035995915648938170415
  663757378637207011391538009596833354107737156273037494727858302028663366296943925008647348769272035
  532265048049709827275179381252898675965528510619258376779171030556482884535728812916216625430187039
  533668677528079544176897647303445153643525354817413650848544778690688201005274443717680593899

Performance Testing

FactorPrimeTest.java

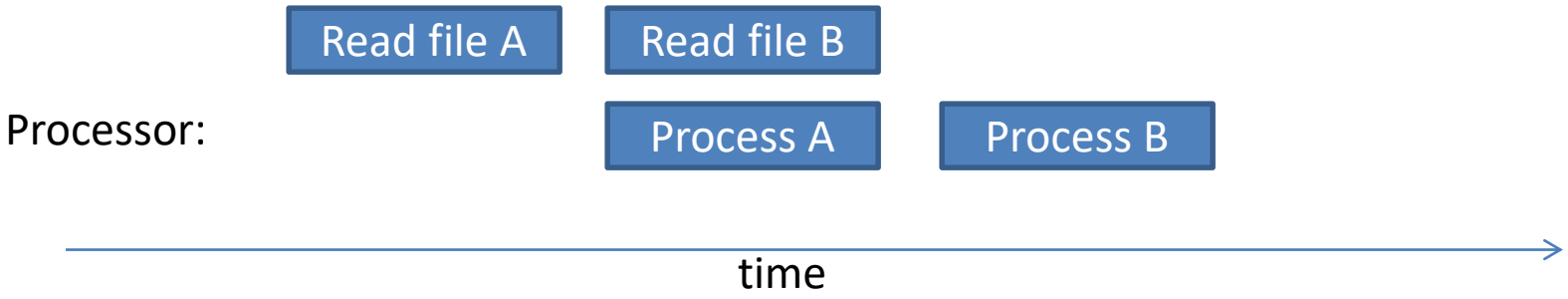
Concurrency: Benefit

Better resource utilization: with K processors, ideally we can be K times faster*



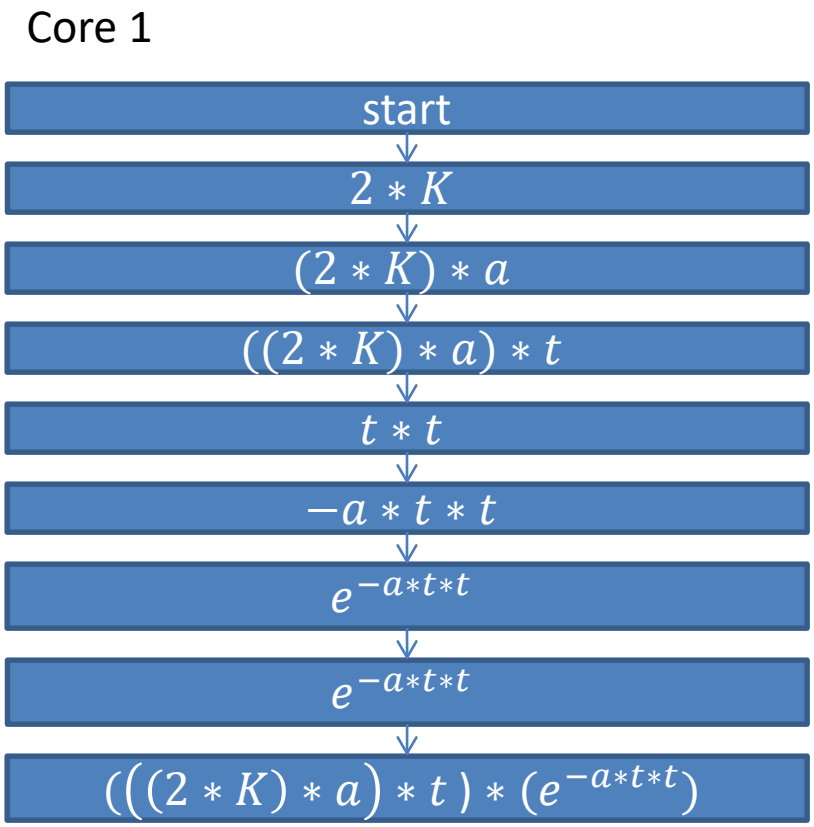
Concurrency: Benefit

Can we get better performance with 1 processor only?



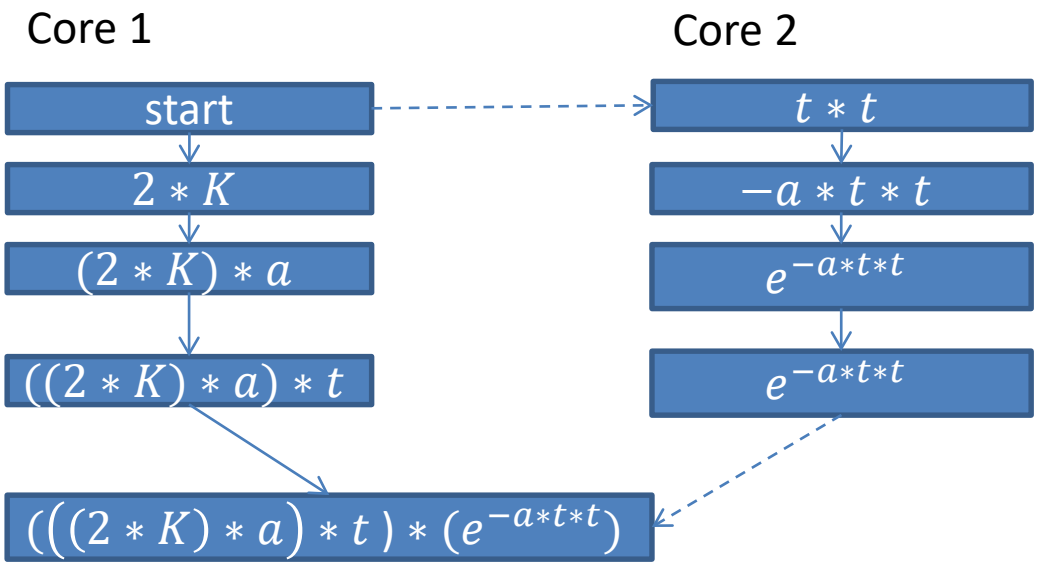
Threads

Example: evaluate $2 * K * a * t * e^{-a*t*t}$



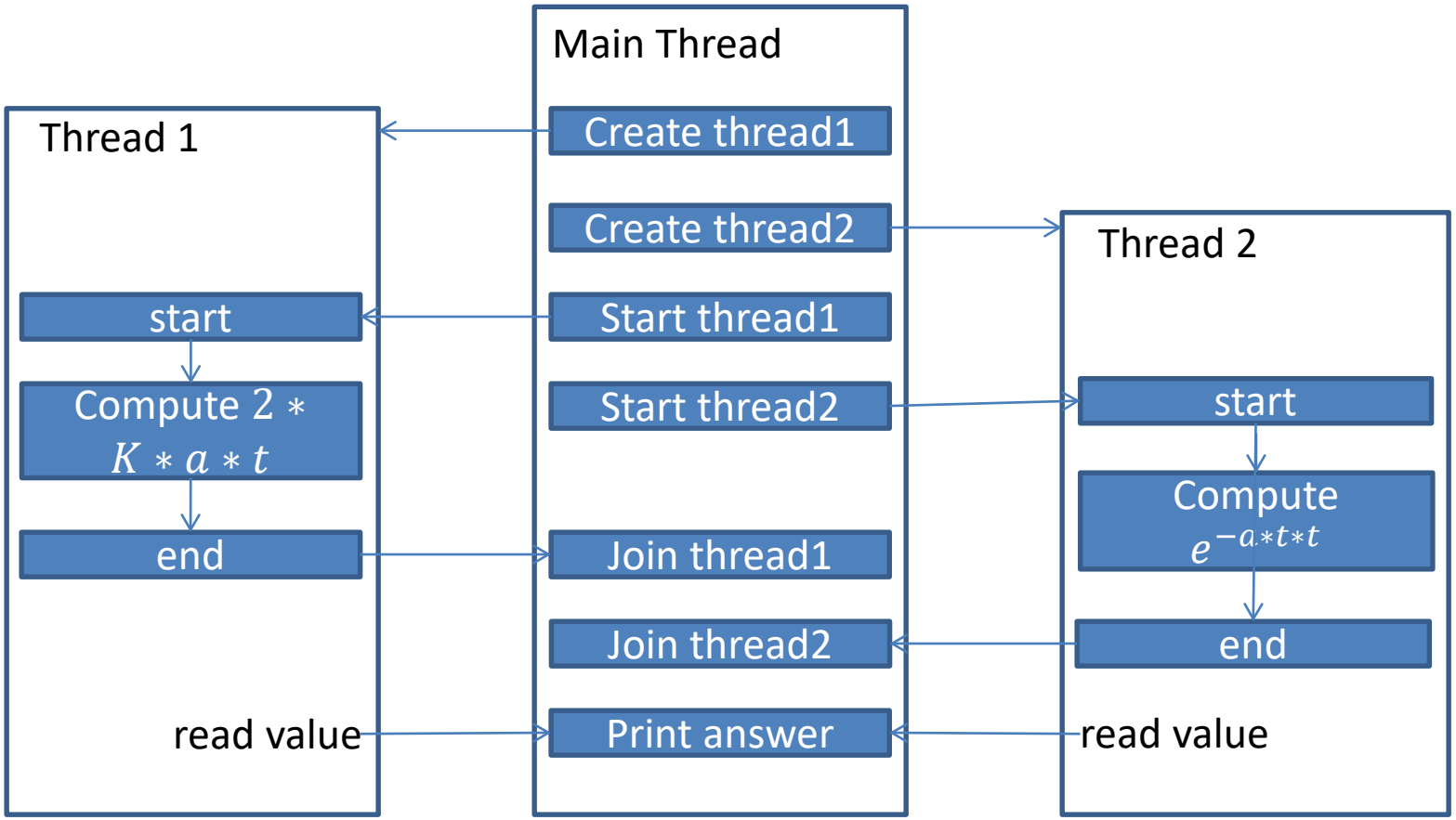
Threads

Example: evaluate $2 * K * a * t * e^{-a*t*t}$

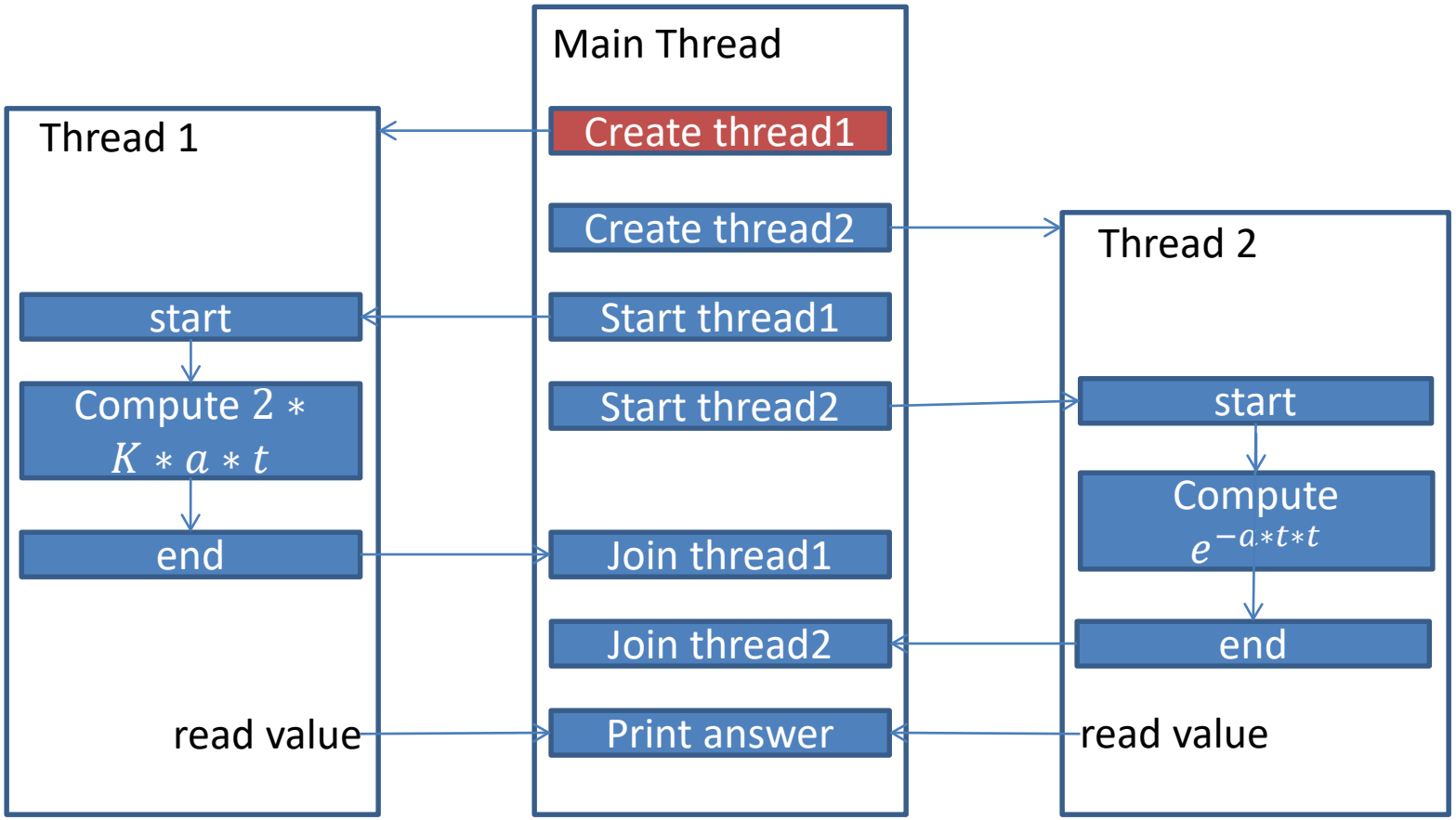


HelloWorldThread.java

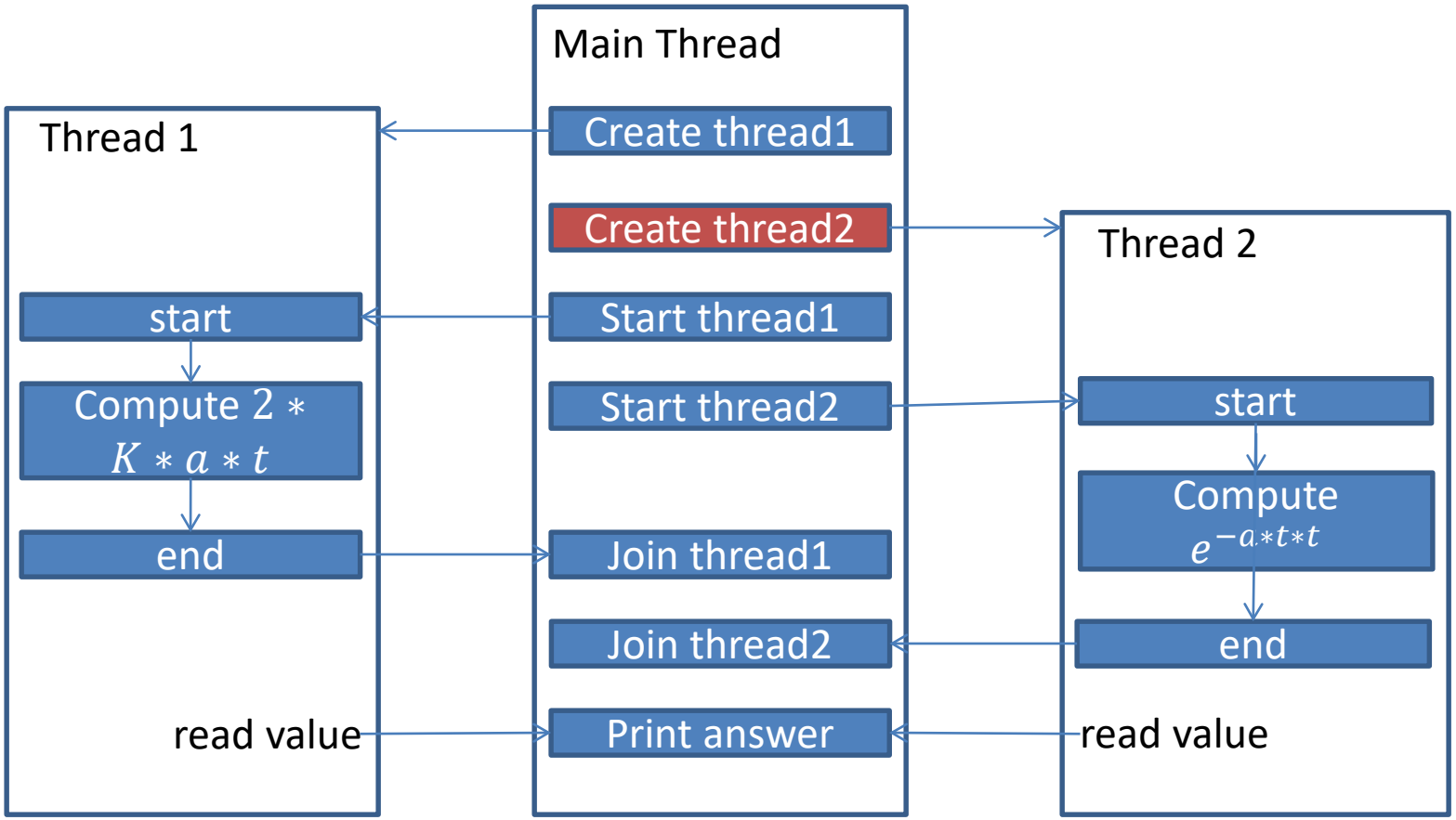
Hello World


Main Thread
Thread 1
Thread 2
Create thread1
Create thread2
start
Start thread1
Compute 2 ∗ K ∗ a ∗ t
Start thread2
start
Compute e^−a∗t∗t
end
Join thread1
Join thread2
end
read value
Print answer
read value

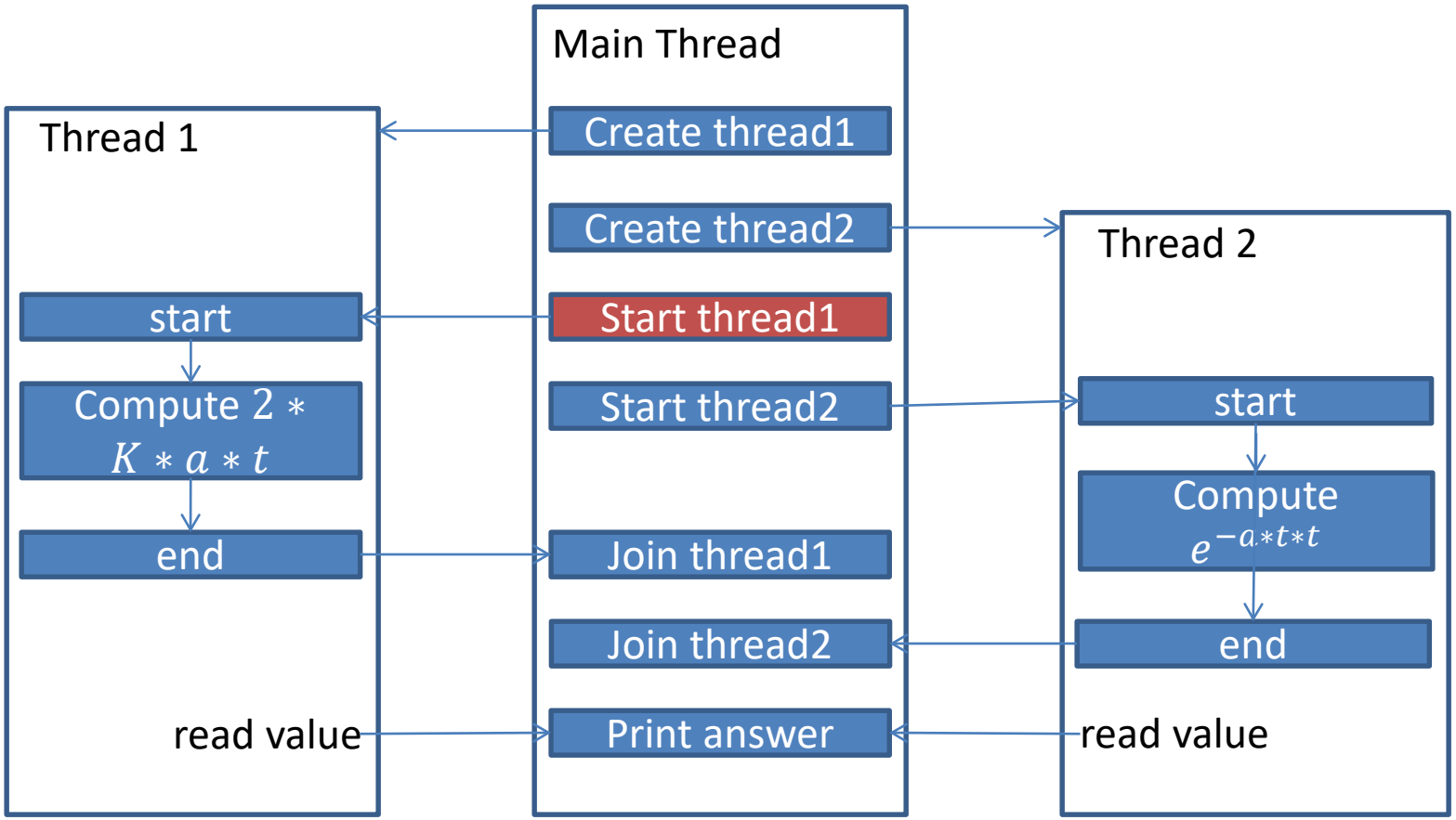
Hello World


Main Thread
Thread 1
Thread 2
Create thread1
Create thread2
start
Start thread1
Compute 2 ∗ K ∗ a ∗ t
Start thread2
start
Compute e^{−a∗t∗t}
end
Join thread1
Join thread2
end
read value
Print answer
read value

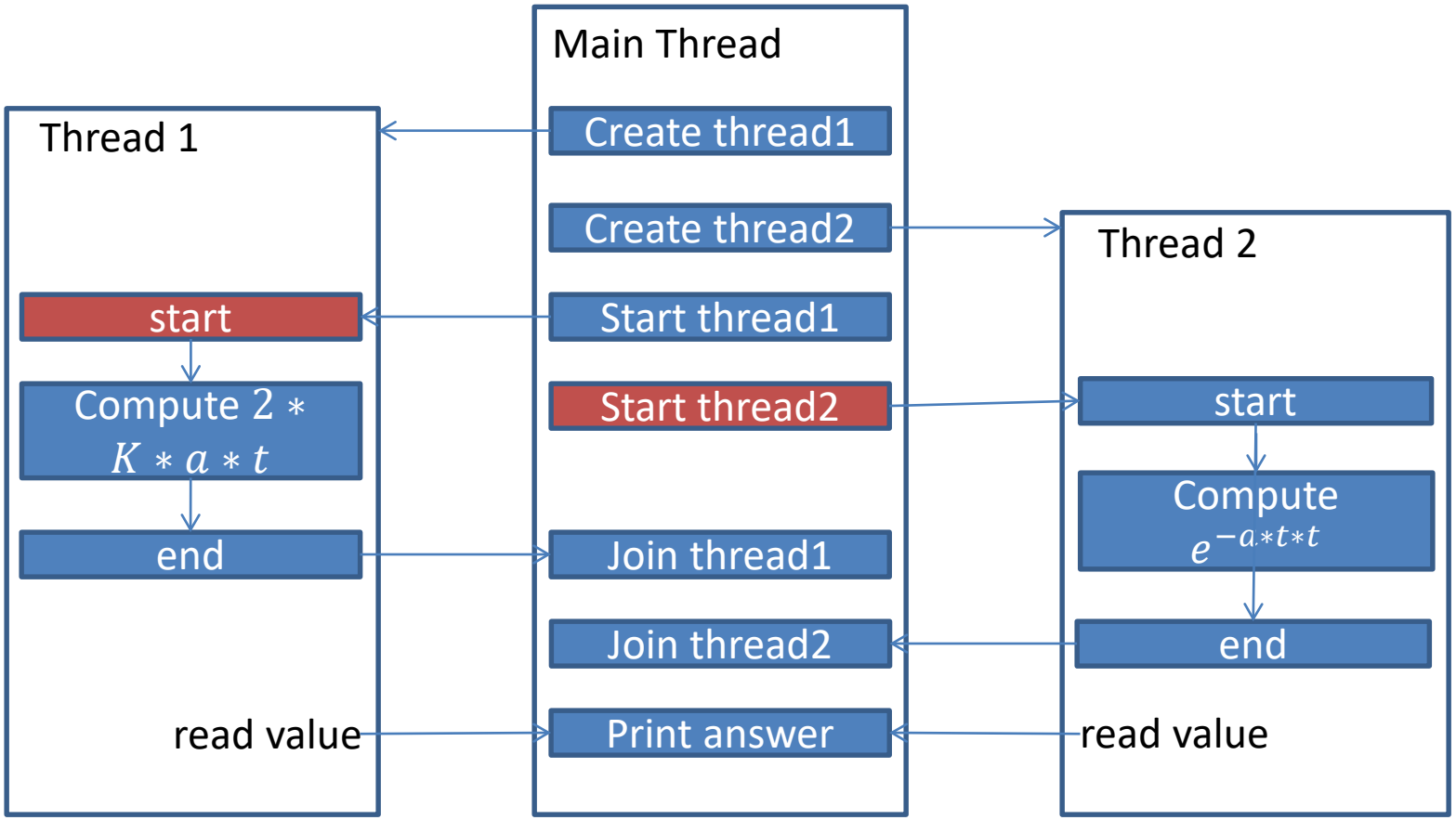
Hello World


Main Thread
Thread 1
Thread 2
Create thread1
Create thread2
start
Start thread1
Compute 2 ∗ K ∗ a ∗ t
Start thread2
start
Compute e^{−a∗t∗t}
end
Join thread1
Join thread2
end
read value
Print answer
read value

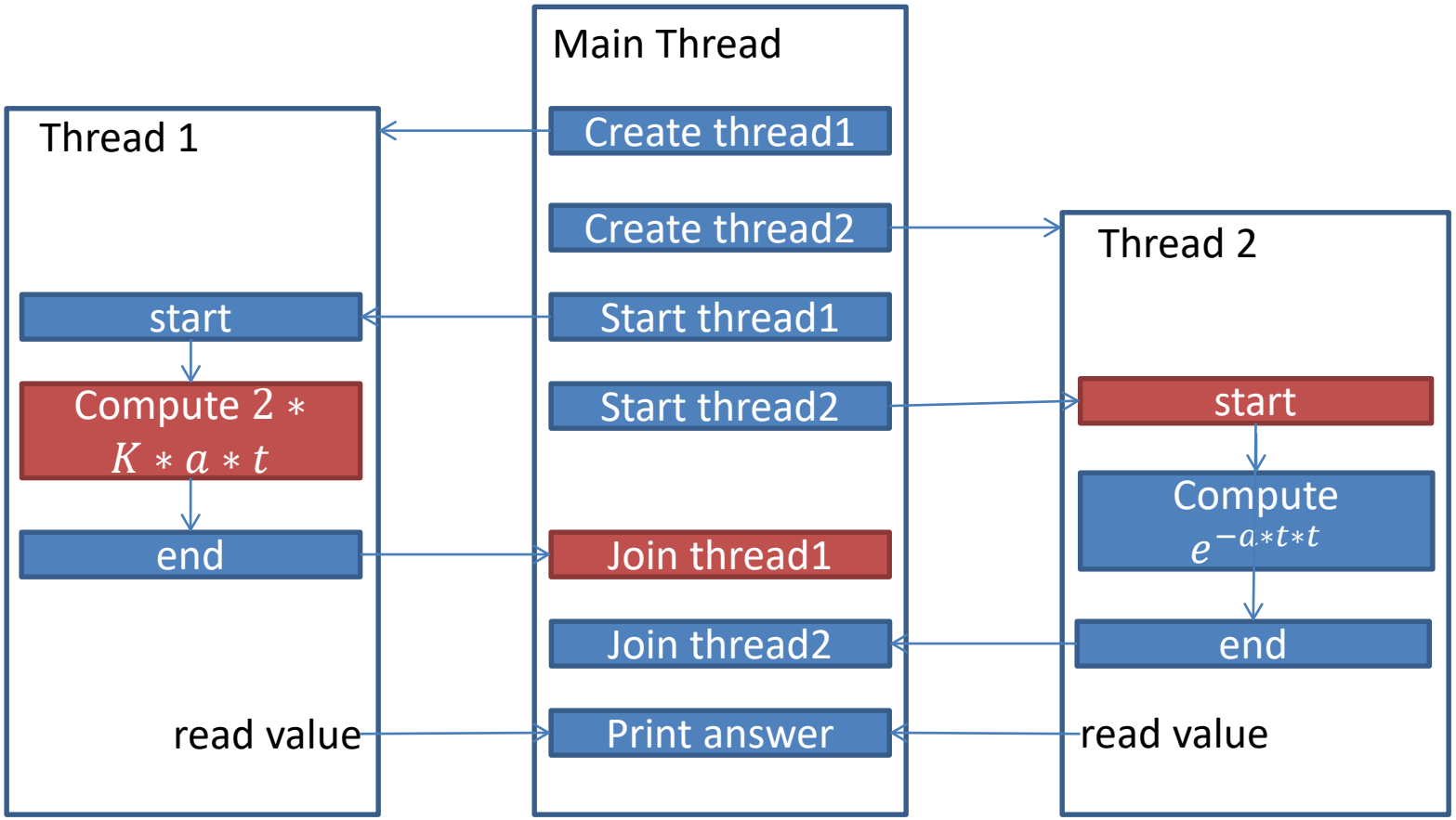
Hello World


Main Thread
Thread 1
Thread 2
Create thread1
Create thread2
start
Start thread1
Compute 2 ∗ K ∗ a ∗ t
Start thread2
start
Compute e^−a∗t∗t
end
Join thread1
Join thread2
end
read value
Print answer
read value

Hello World

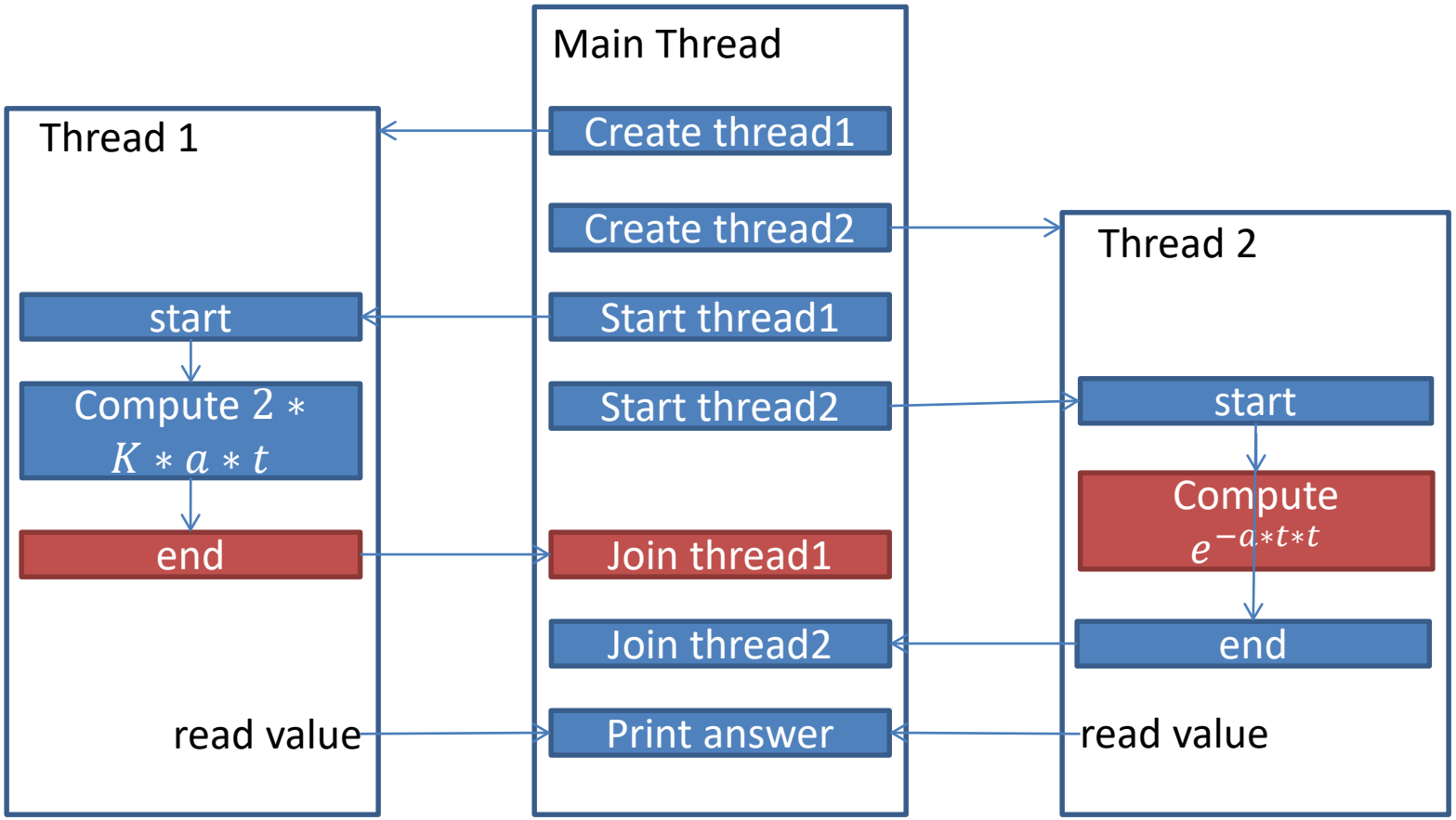

Main Thread
Thread 1
Thread 2
Create thread1
Create thread2
start
Start thread1
Compute 2 ∗ K ∗ a ∗ t
Start thread2
start
Compute e^{−a∗t∗t}
end
Join thread1
Join thread2
end
read value
Print answer
read value

Hello World

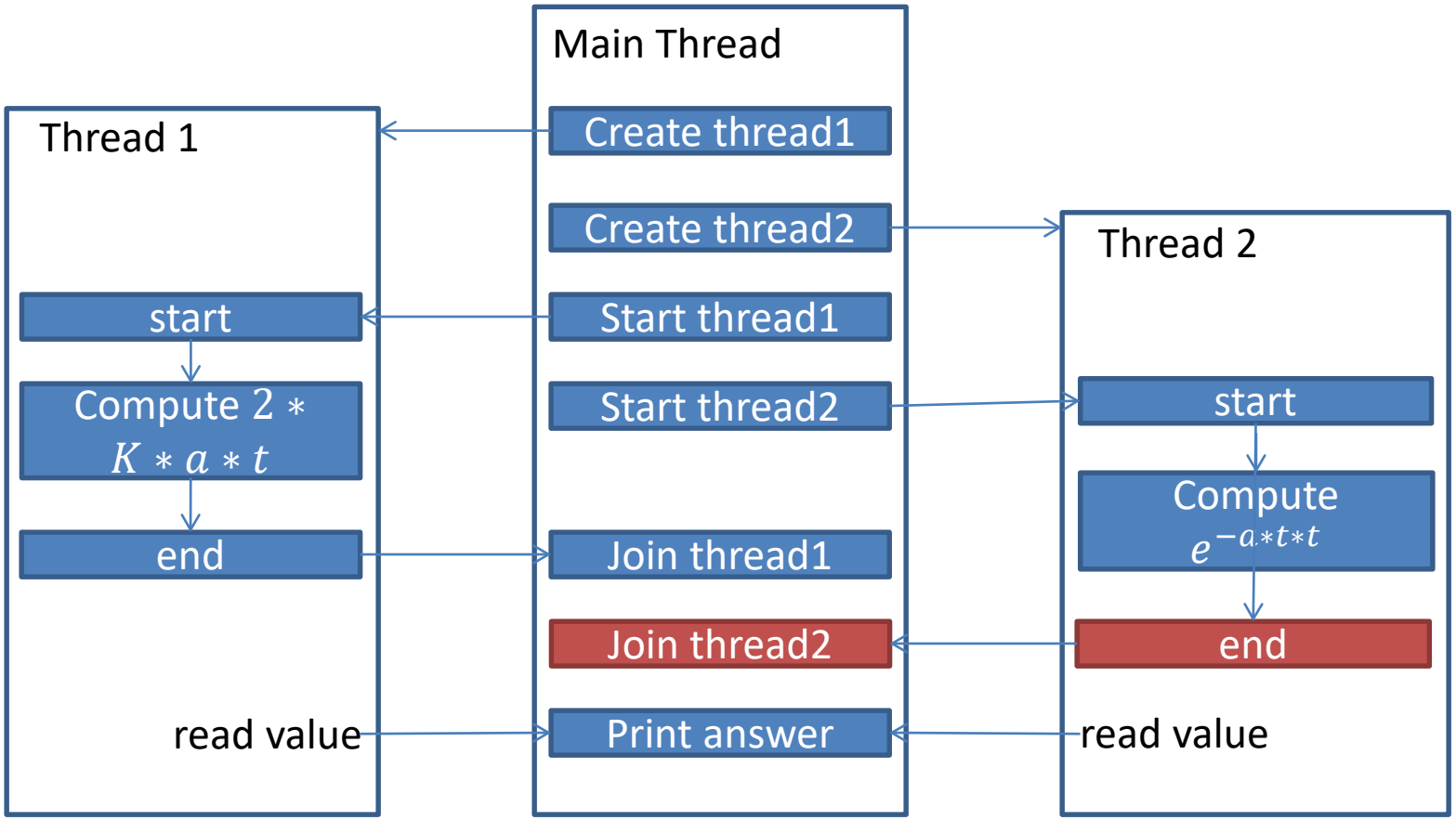


Which thread moves first is non-deterministic

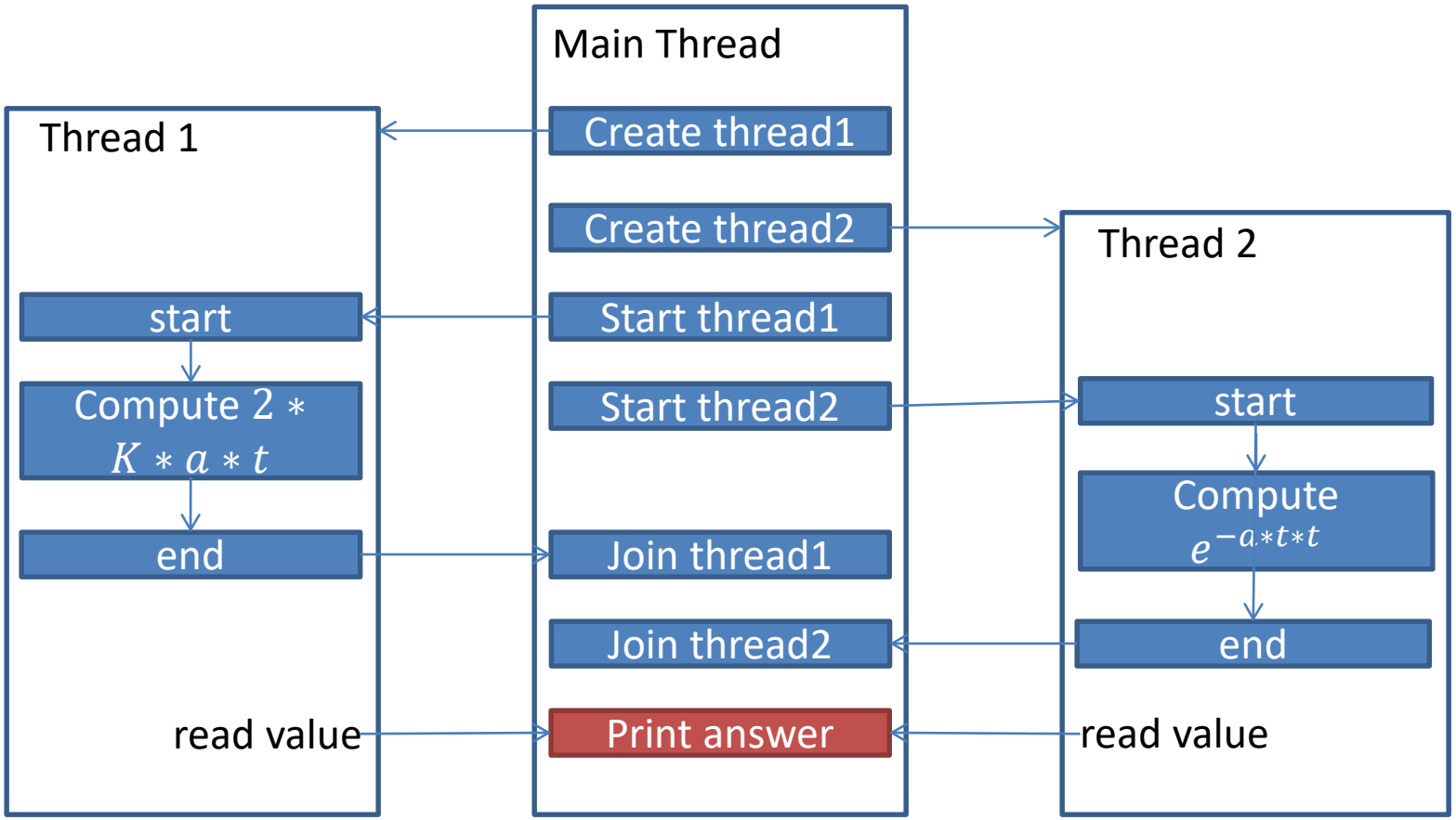
Hello World


Main Thread
Thread 1
Thread 2
Create thread1
Create thread2
start
Start thread1
Compute 2 ∗ K ∗ a ∗ t
Start thread2
start
Compute e^−a∗t∗t
end
Join thread1
Join thread2
end
read value
Print answer
read value

Hello World


Main Thread
Thread 1
Thread 2
Create thread1
Create thread2
start
Start thread1
Compute 2 ∗ K ∗ a ∗ t
Start thread2
start
Compute e^−a∗t∗t
end
Join thread1
Join thread2
end
read value
Print answer
read value

Hello World


Main Thread
Thread 1
Thread 2
Create thread1
Create thread2
start
Start thread1
Compute 2 ∗ K ∗ a ∗ t
Start thread2
start
Compute e^{−a∗t∗t}
end
Join thread1
Join thread2
end
read value
Print answer
read value

Runnable interface

- “implements Runnable” vs. “extends Thread”
  - Use Runnable that you can inherit other classes
  - Runnable allows better separation of the actual computation and thread control.

RunnableExample.java

Cohort Exercise 1

Given FactorPrime.java, write a program so that multi-threads factor the number at the same time. Print the factor as soon as it is found.

How to Stop a Thread

Given FactorPrime.java, write a program so that multi-threads factor the number at the same time. Print the factor as soon as it is found.

How do we stop a thread as soon as a factor has been found?

How to Stop a Thread

- No
  - destroy()
  - or stop()
  - or stop(Throwable obj)
  - or suspend()
- Yes
  - Interrupt()

interrupt()

- Example 1: interrupt() results in Exception
- Example 2: handle interrupt() explicitly in run() by checking Thread.interrupted()

InterruptExample1.java and InterruptExample2.java

Cohort Exercise 2

Continue with Exercise 1, modify your program such that once the factors are found, stop all threads as soon as possible by interrupting the threads. (Use a static Boolean variable found for communication between threads.)

Test your program with FactorPrimeTest.java.

It’s Not That Simple

- Multi-threaded programs could have race conditions, visibility issues, deadlocks, etc.

Problem 1: Race Condition

Example

Program FirstError.java has 10000 threads which concurrently increment a static variable (initially 0) by 1.

What is the Problem?

- A sequential program consisted of a sequence of instructions (and a memory), where each instruction executed one after the other (to modify the memory, etc.).
- The sequential paradigm has the following two characteristics: the textual order of statements specifies their order of execution; successive statements must be executed without any overlap (in time) with one another.

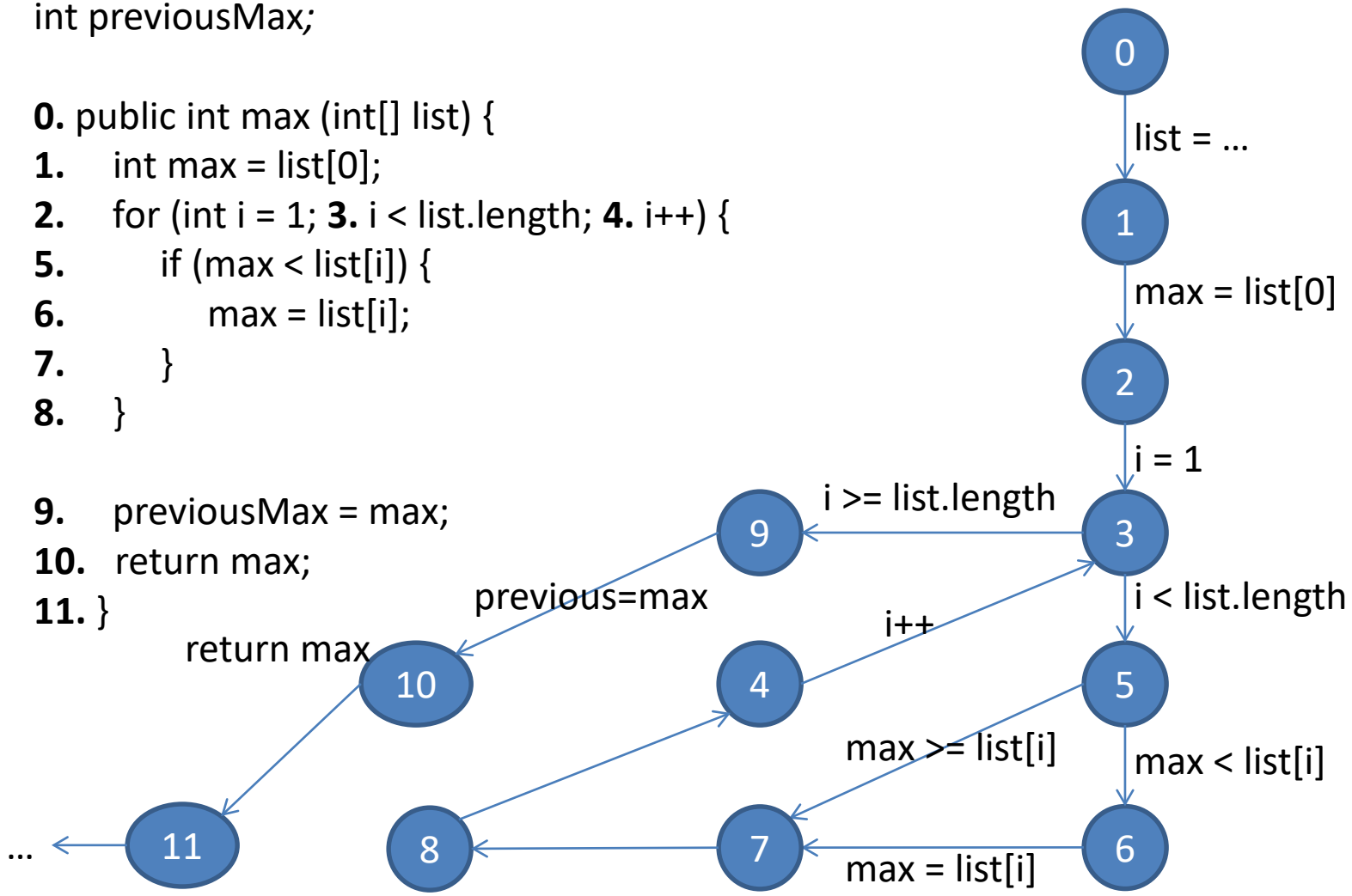
Both are not true in concurrent programs.

The Illusion

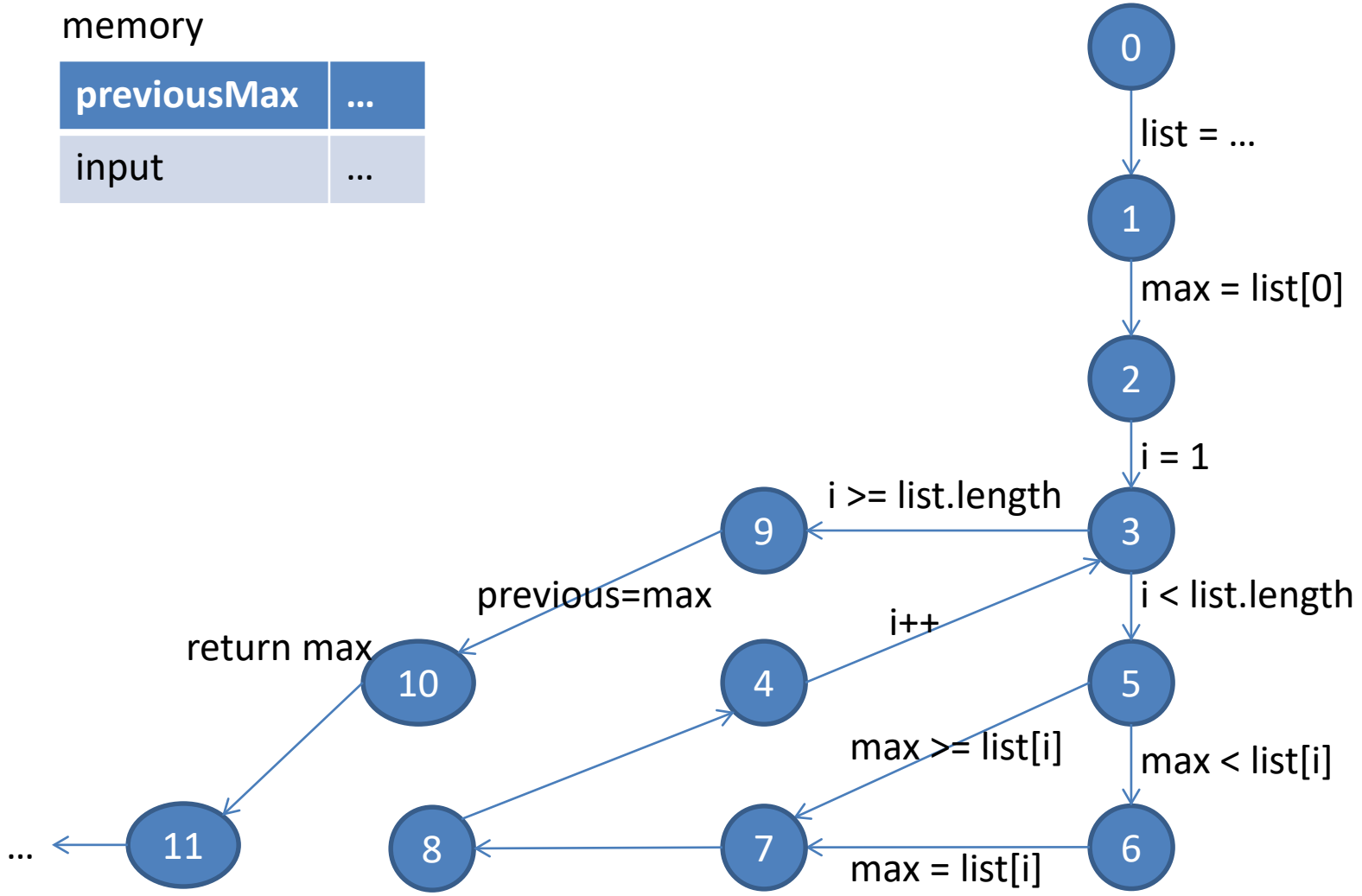
```
int previousMax;

0. public int max (int[] list) {
1.     int max = list[0];
2.     for (int i = 1; 3. i < list.length; 4. i++) {
5.         if (max < list[i]) {
6.             max = list[i];
7.         }
8.     }

9.     previousMax = max;
10.    return max;
11. }
```



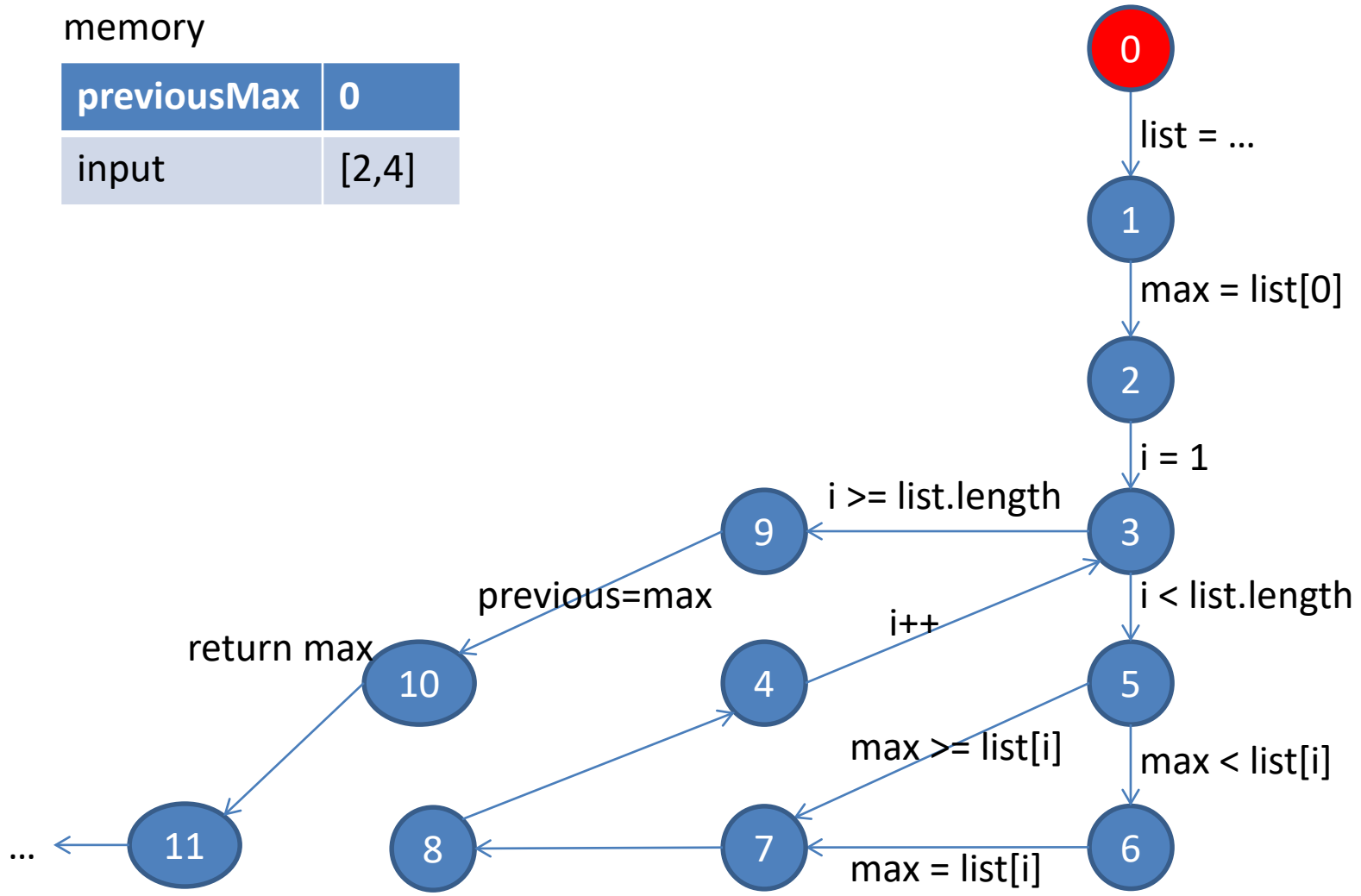
Control Flow Graph



System Execution

memory

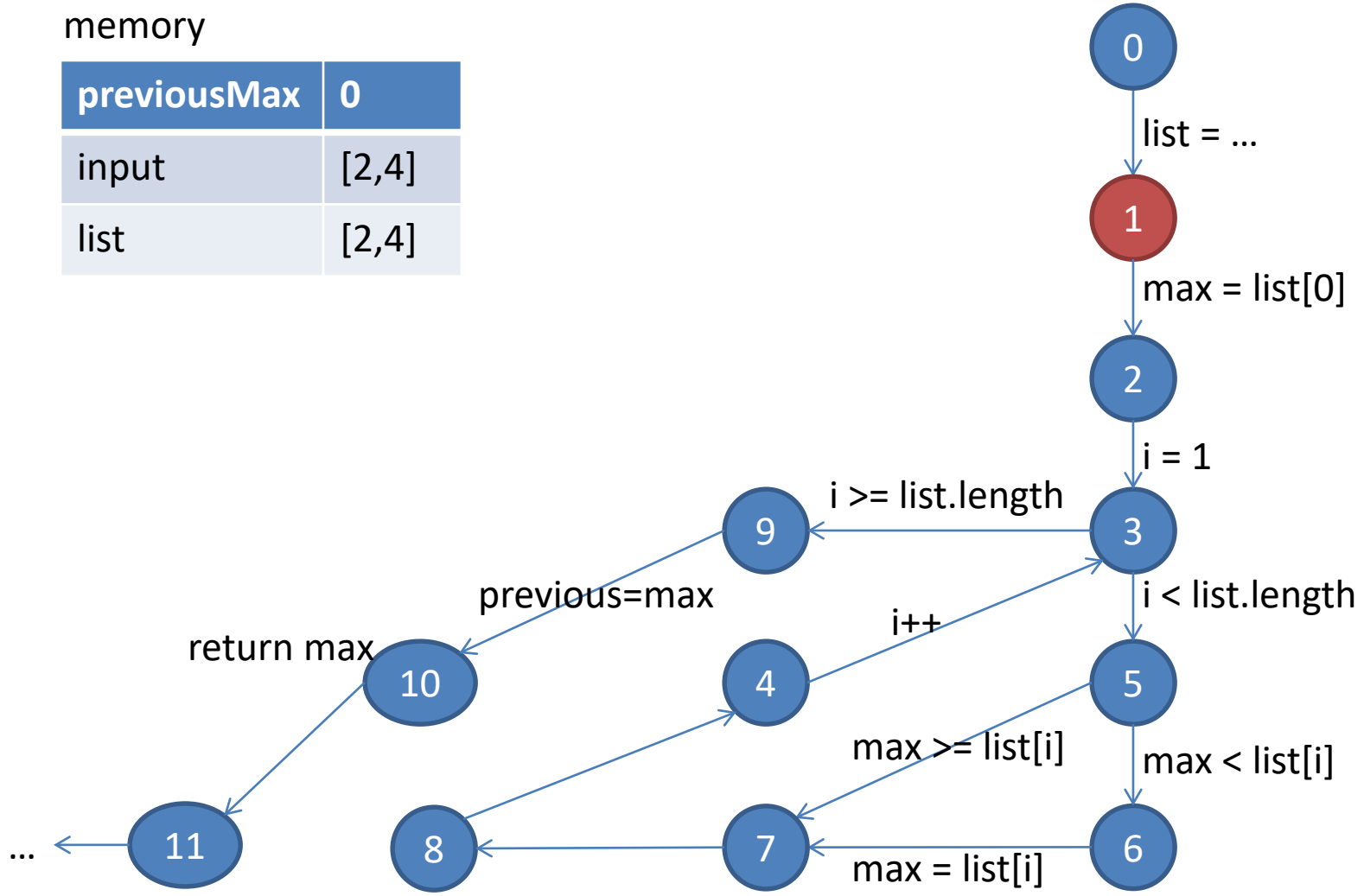
previousMax	0
input	[2,4]



System Execution

memory

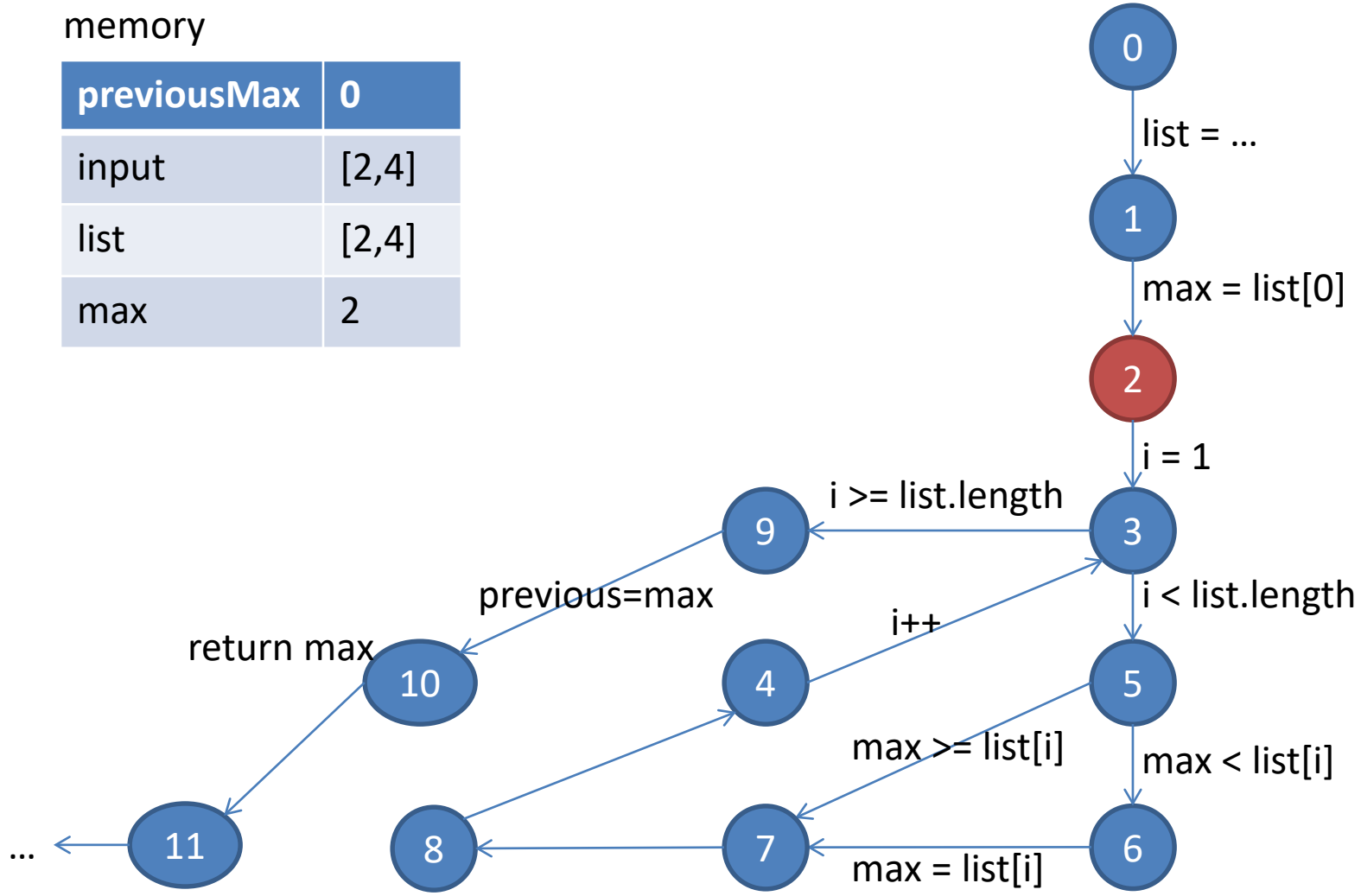
previousMax	0
input	[2,4]
list	[2,4]



System Execution

memory

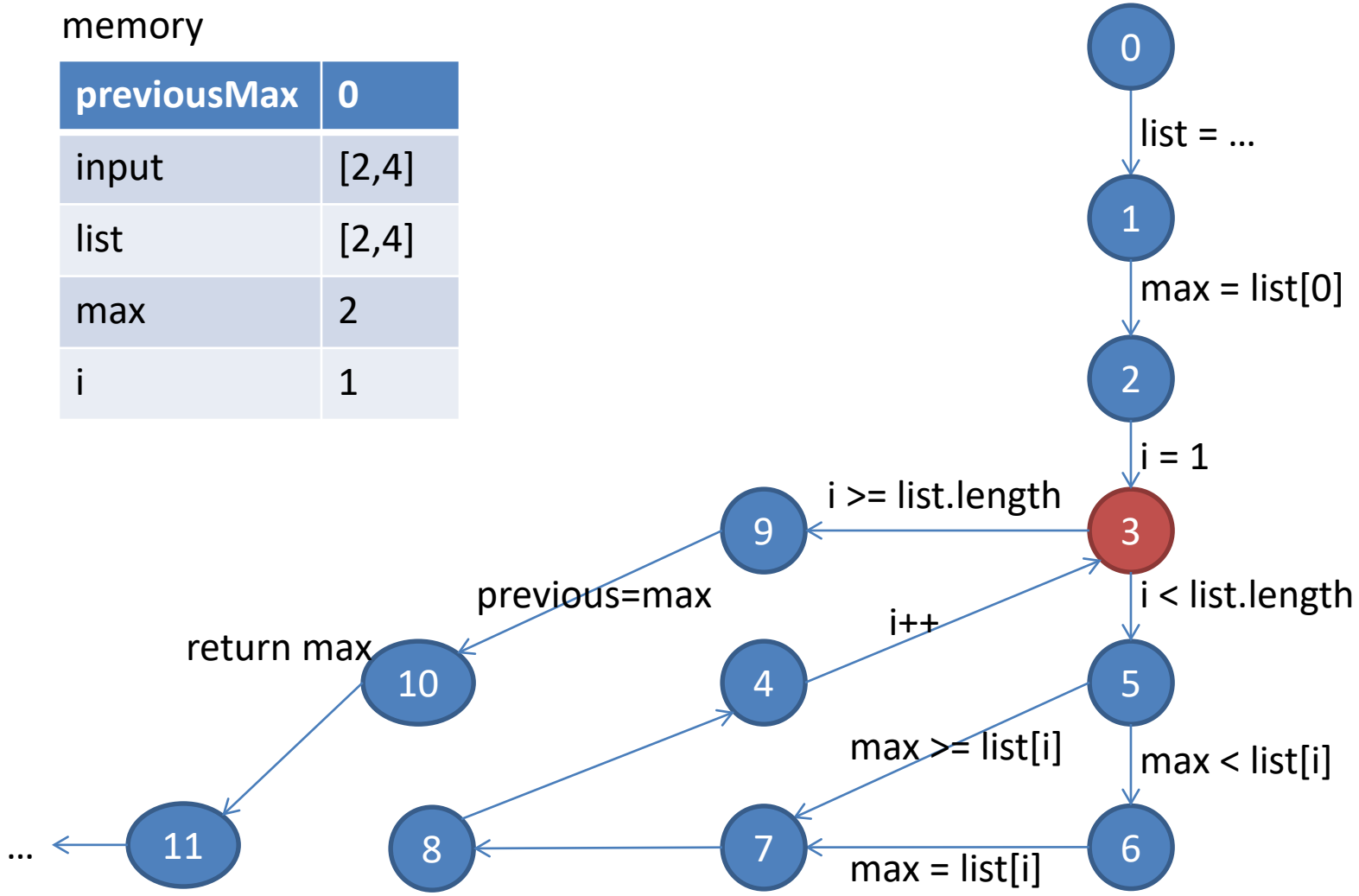
previousMax	0
input	[2,4]
list	[2,4]
max	2



System Execution

memory

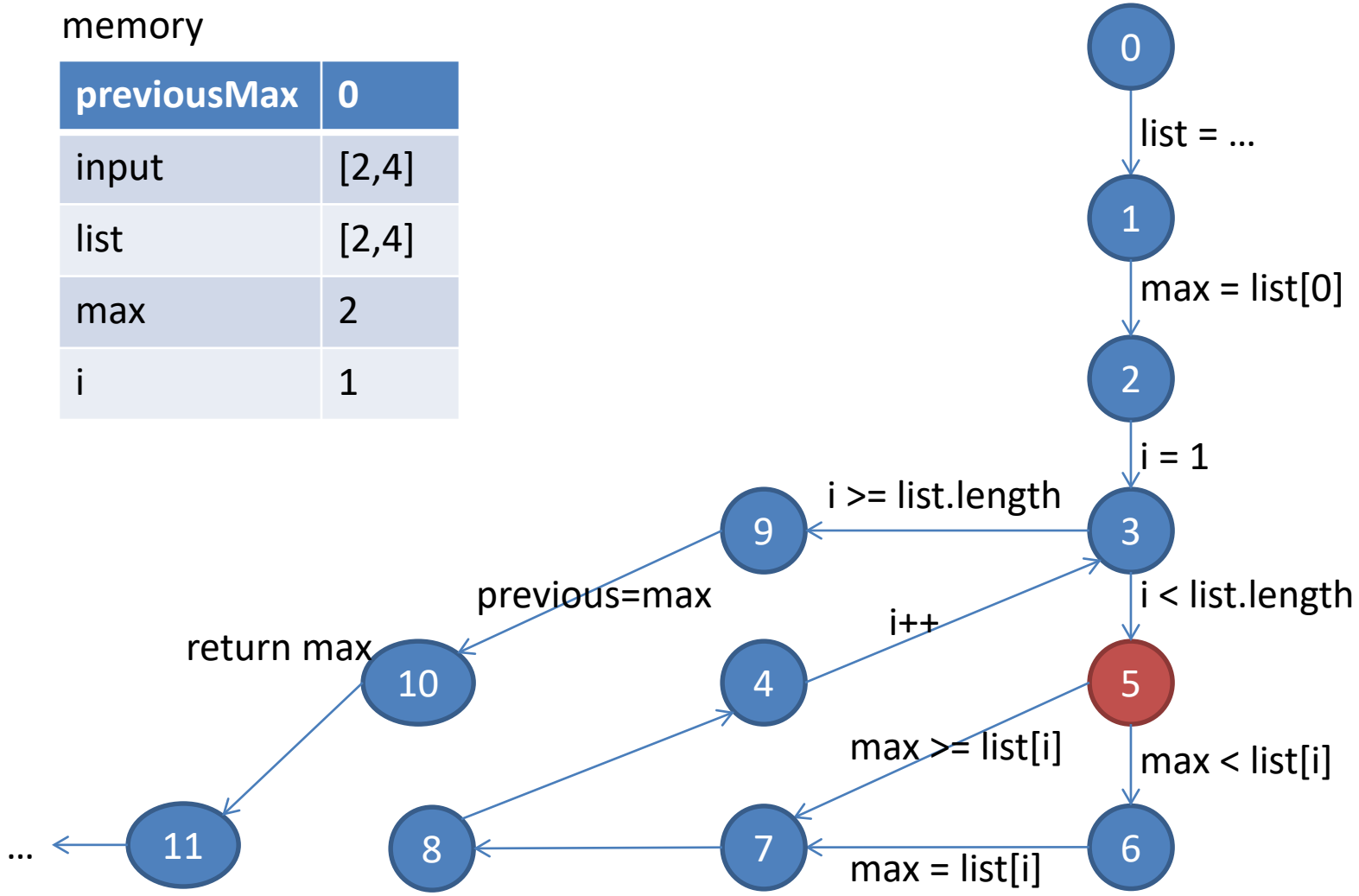
previousMax	0
input	[2,4]
list	[2,4]
max	2
i	1



System Execution

memory

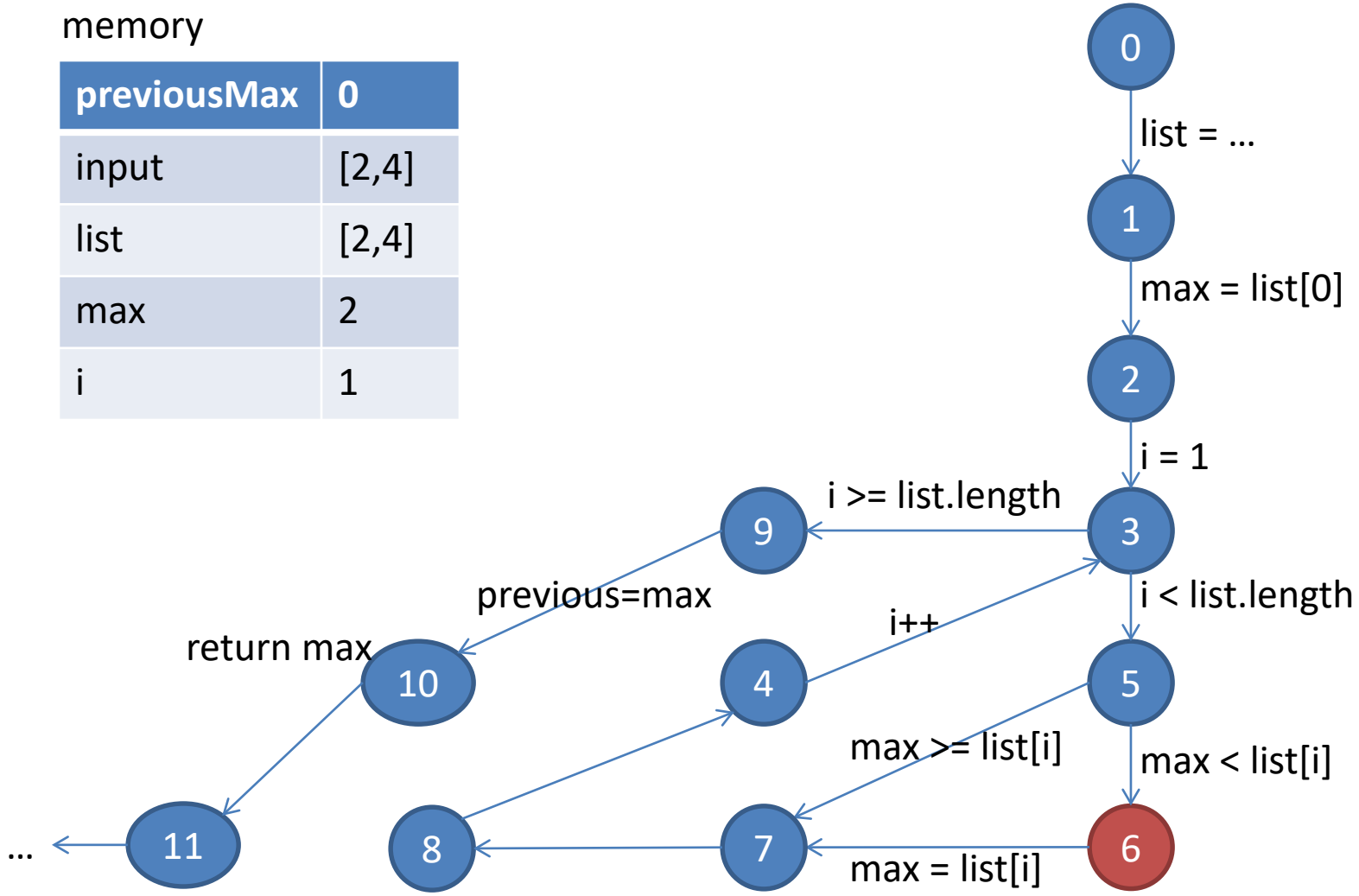
previousMax	0
input	[2,4]
list	[2,4]
max	2
i	1



System Execution

memory

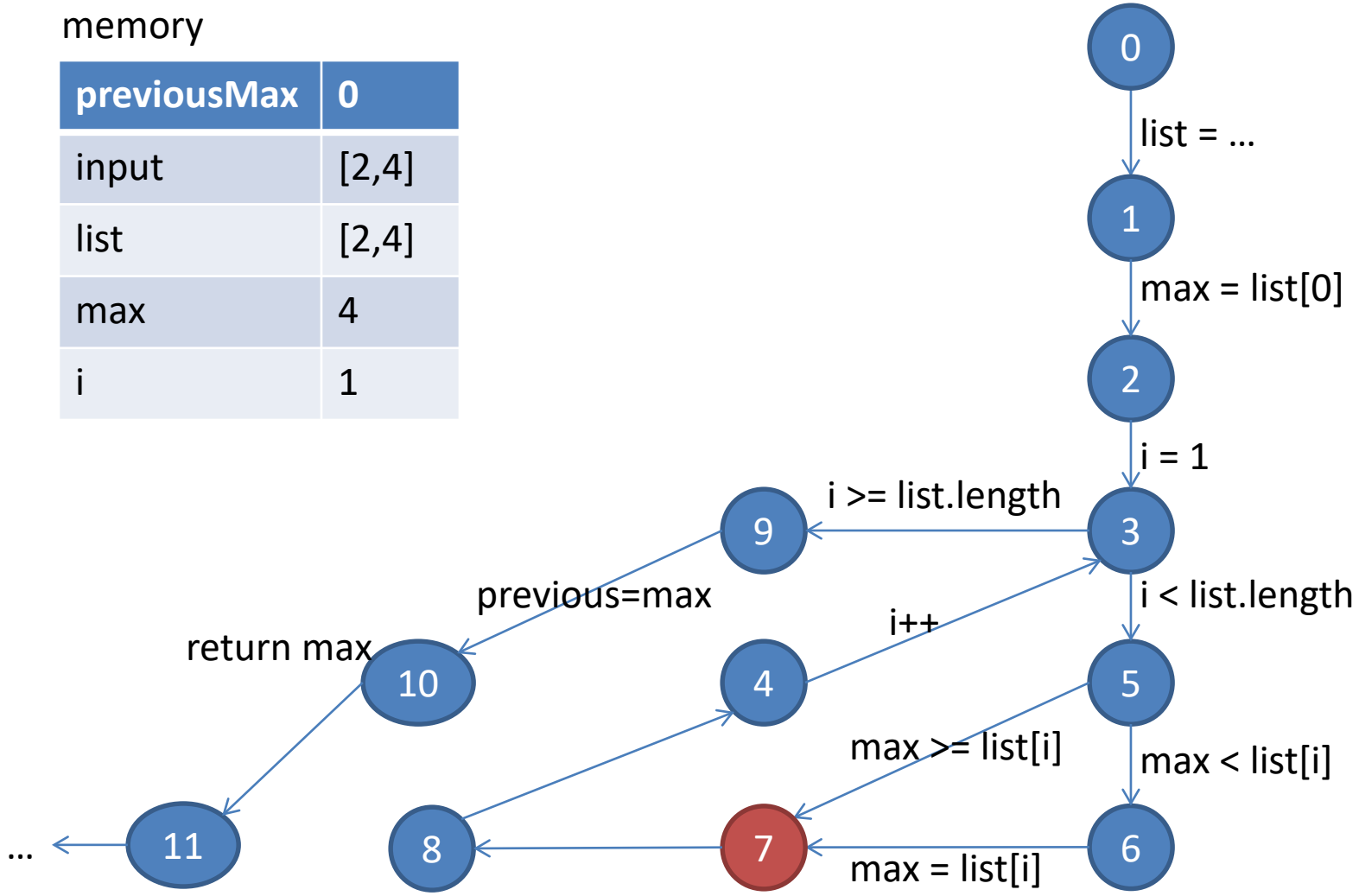
previousMax	0
input	[2,4]
list	[2,4]
max	2
i	1



System Execution

memory

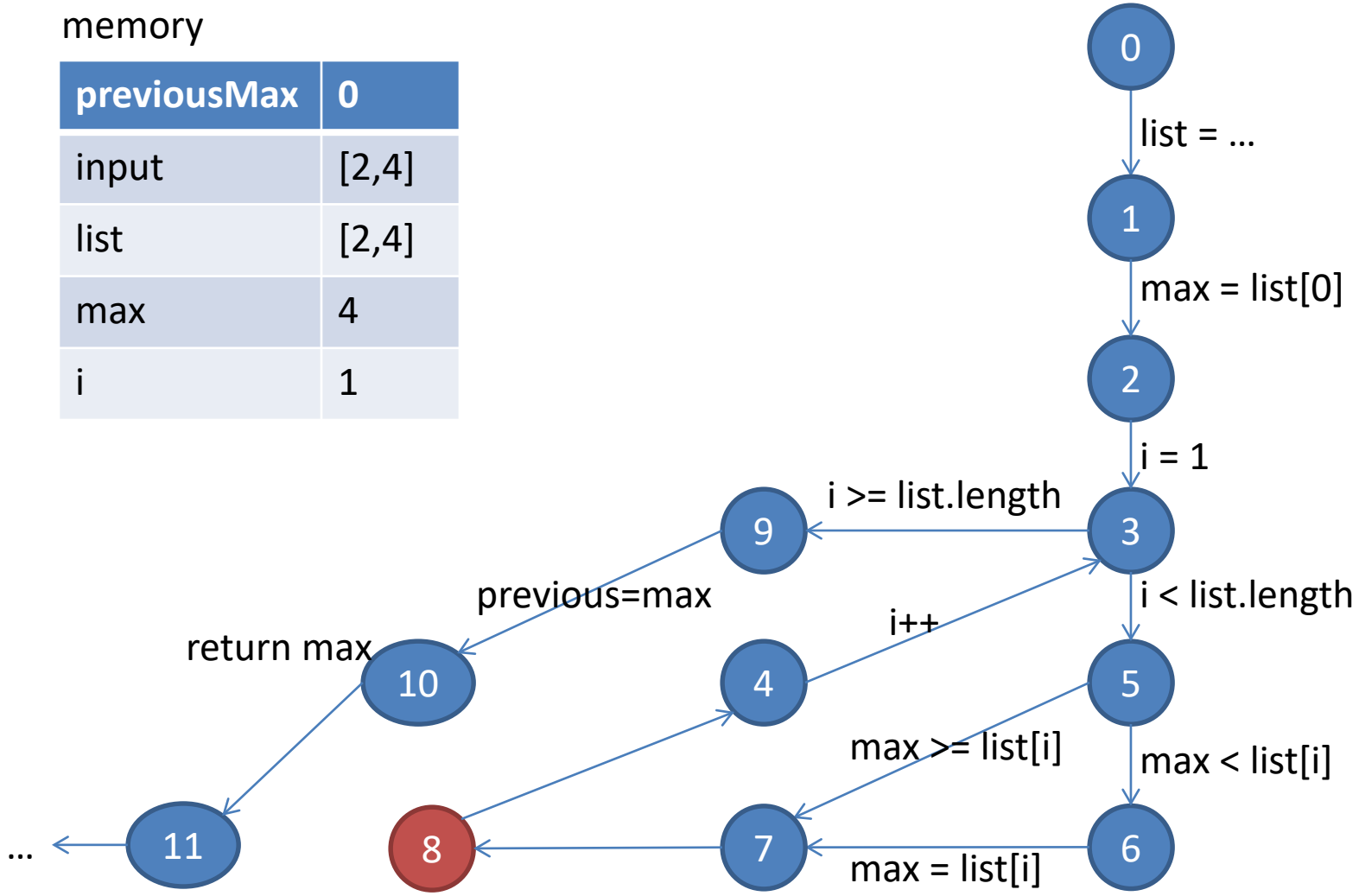
previousMax	0
input	[2,4]
list	[2,4]
max	4
i	1



System Execution

memory

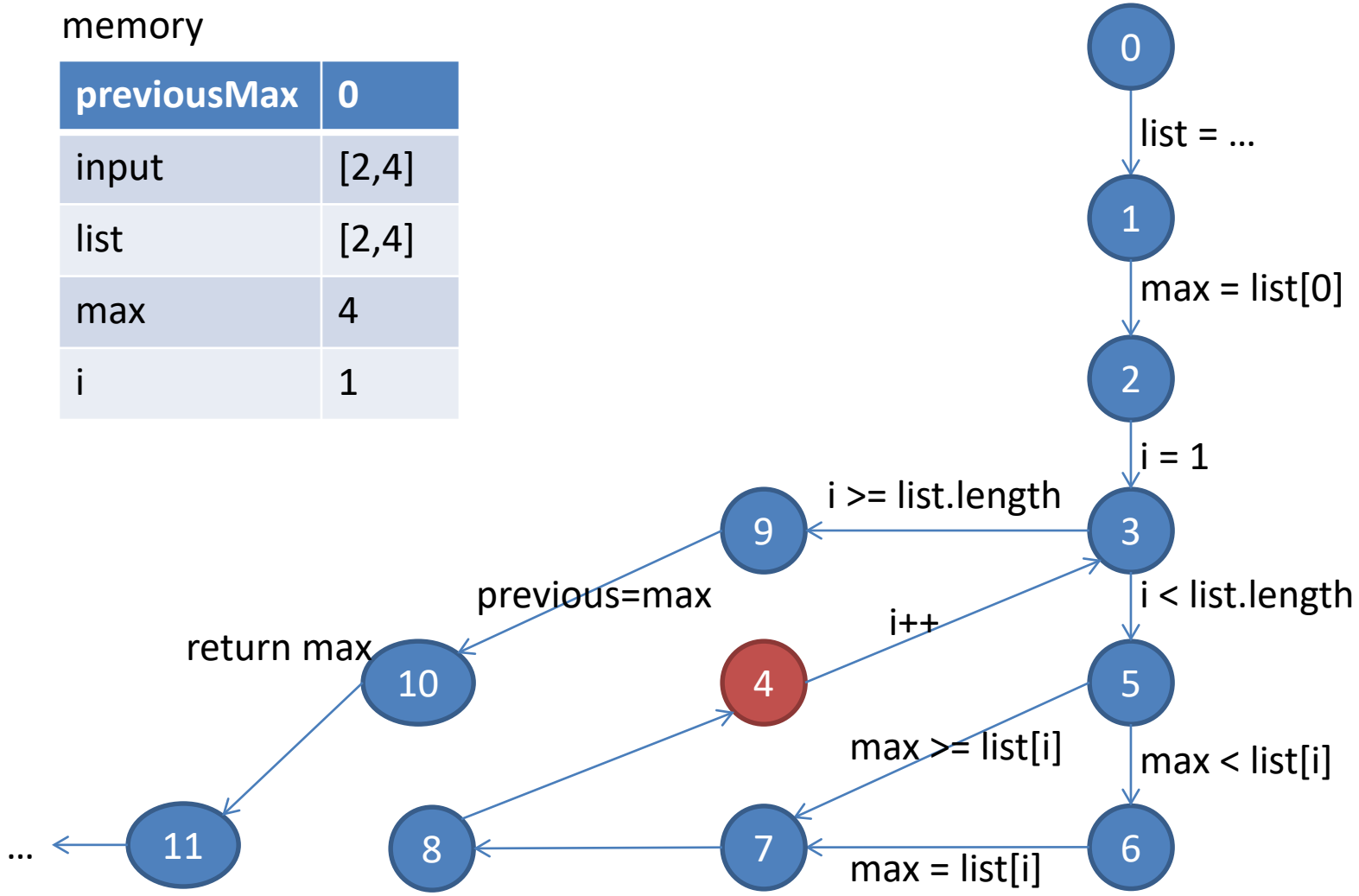
previousMax	0
input	[2,4]
list	[2,4]
max	4
i	1



System Execution

memory

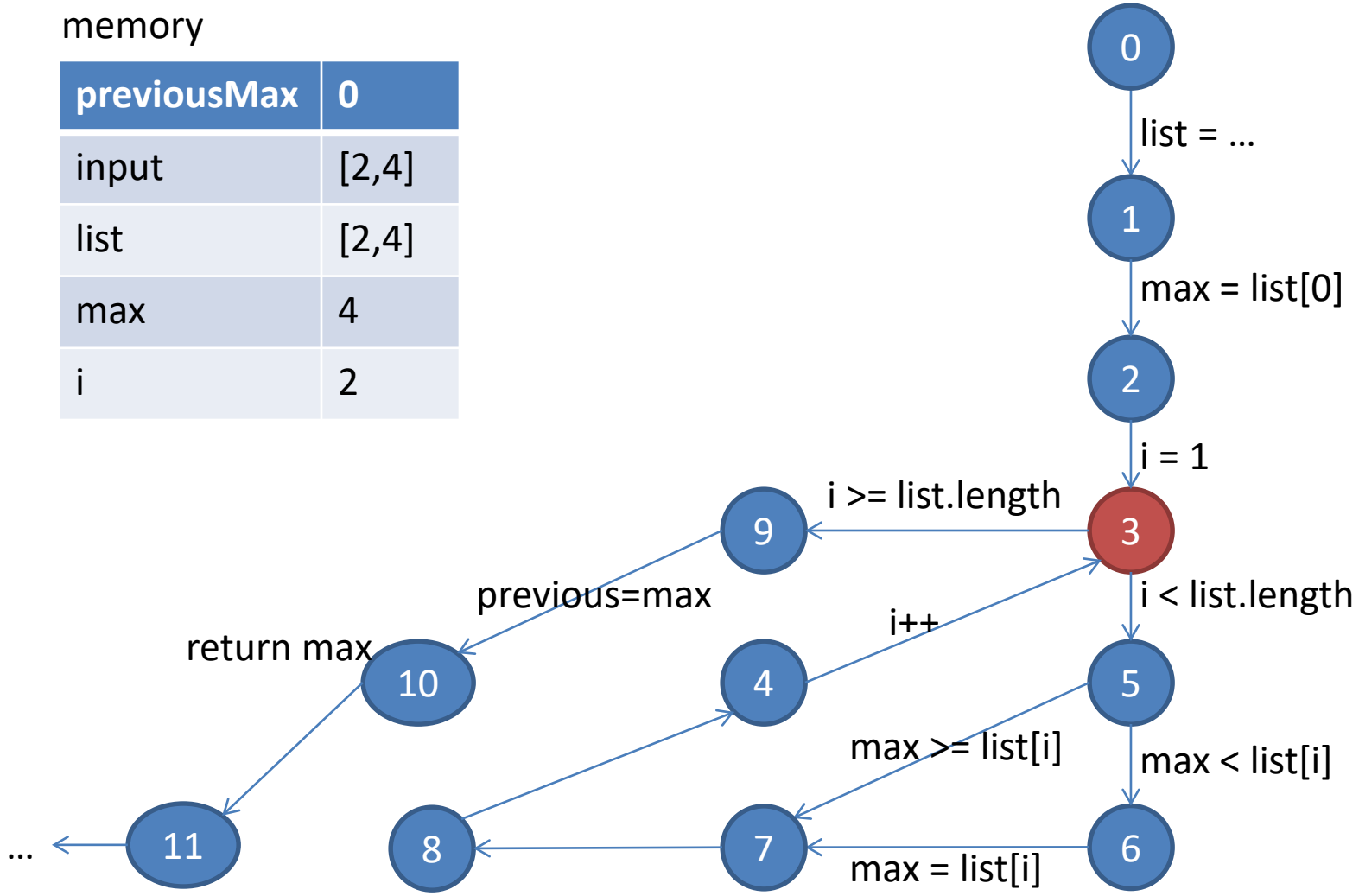
previousMax	0
input	[2,4]
list	[2,4]
max	4
i	1



System Execution

memory

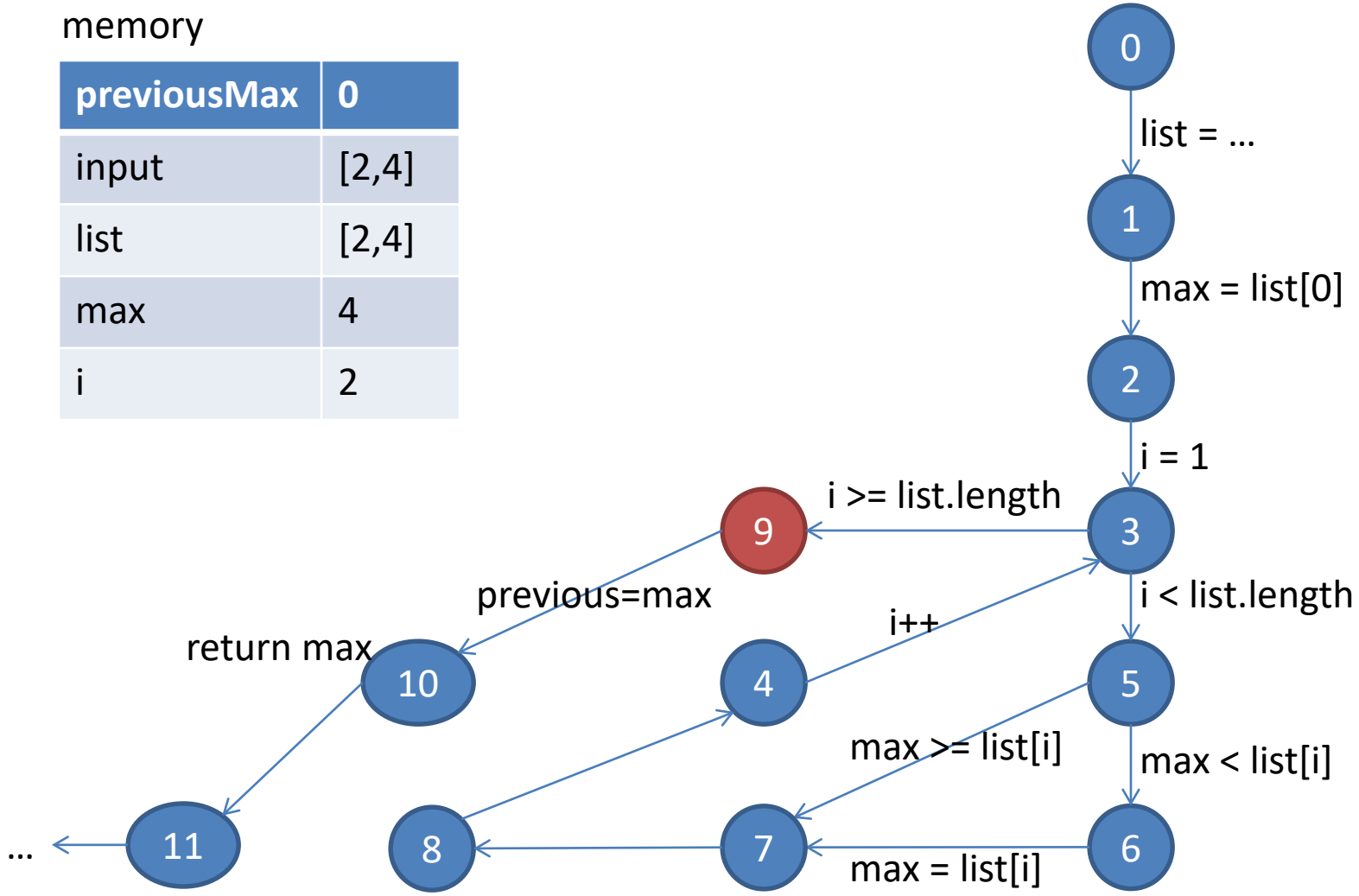
previousMax	0
input	[2,4]
list	[2,4]
max	4
i	2



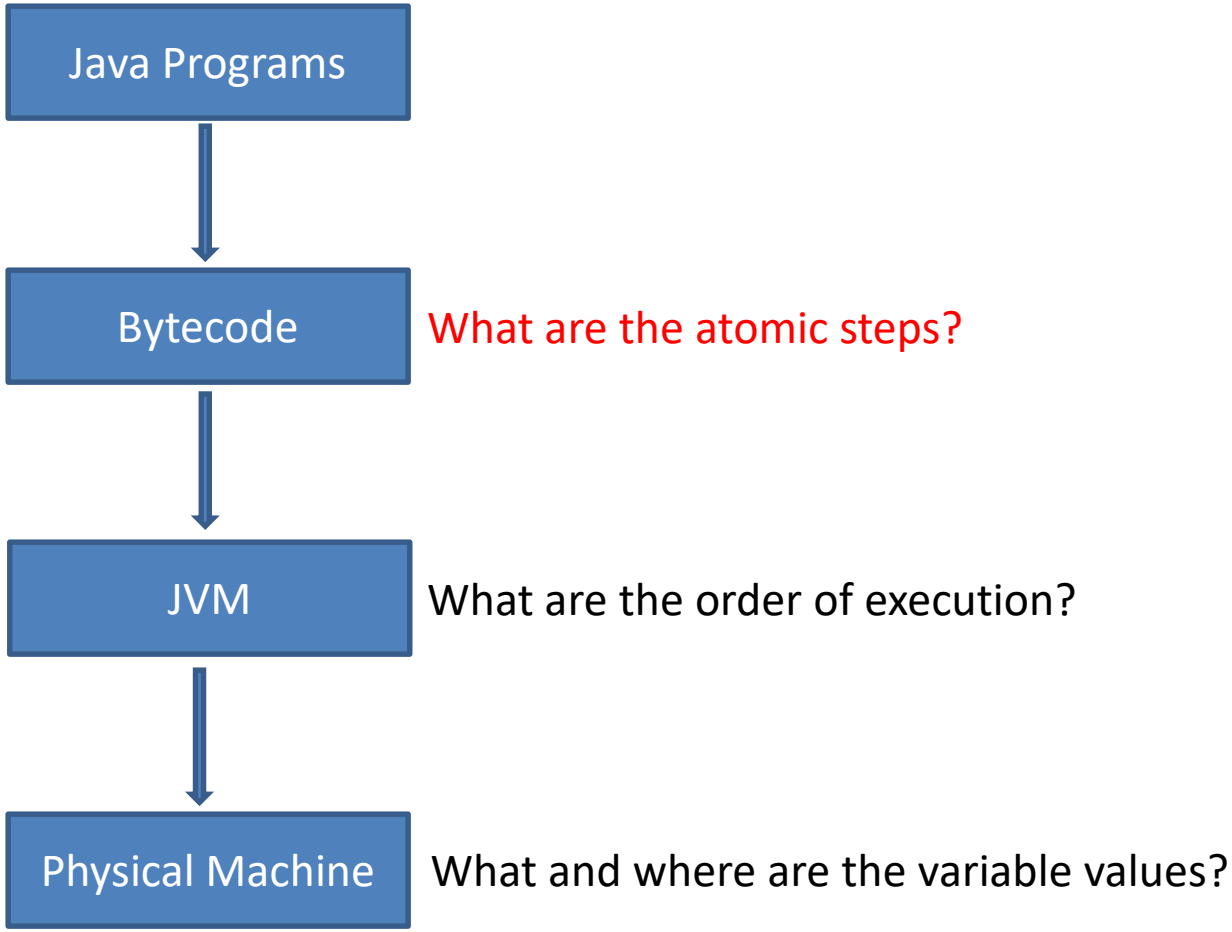
System Execution

memory

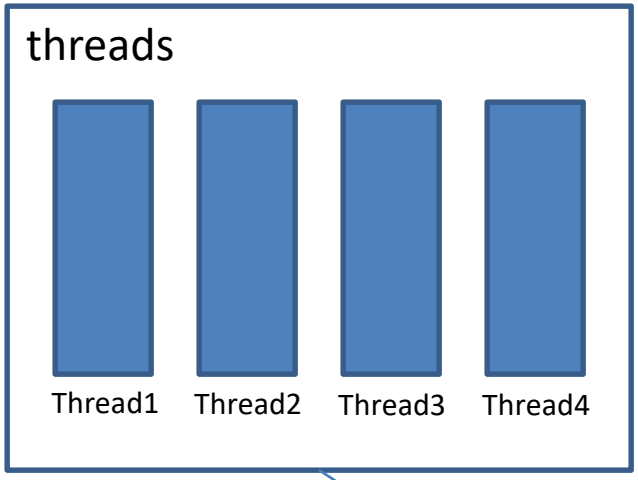
previousMax	0
input	[2,4]
list	[2,4]
max	4
i	2

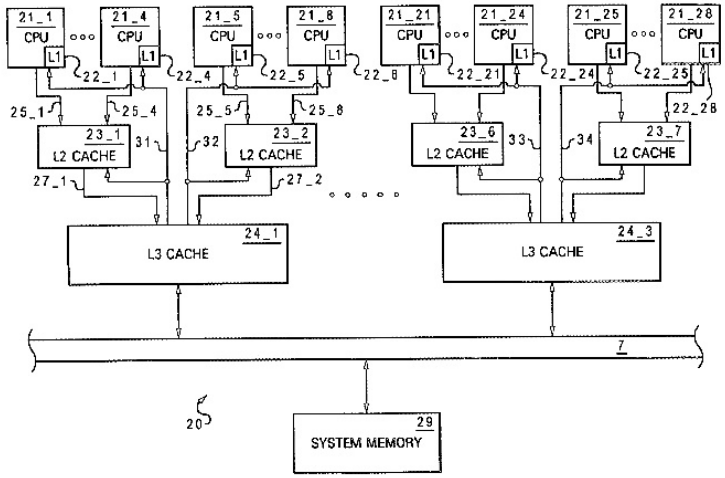


Reality is Messy



Scheduling

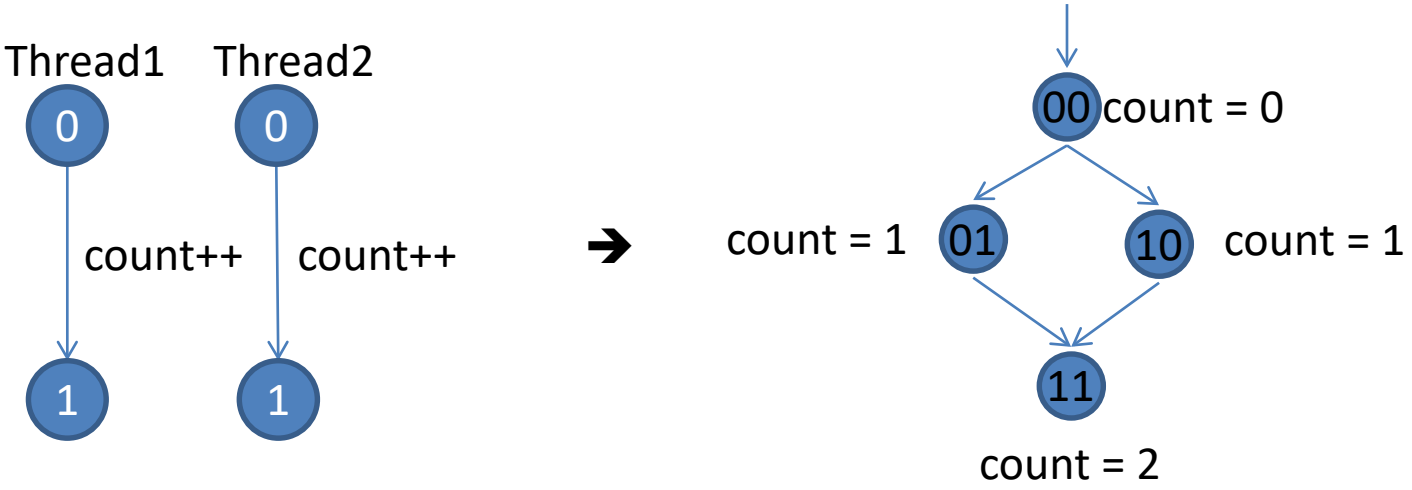




Scheduler

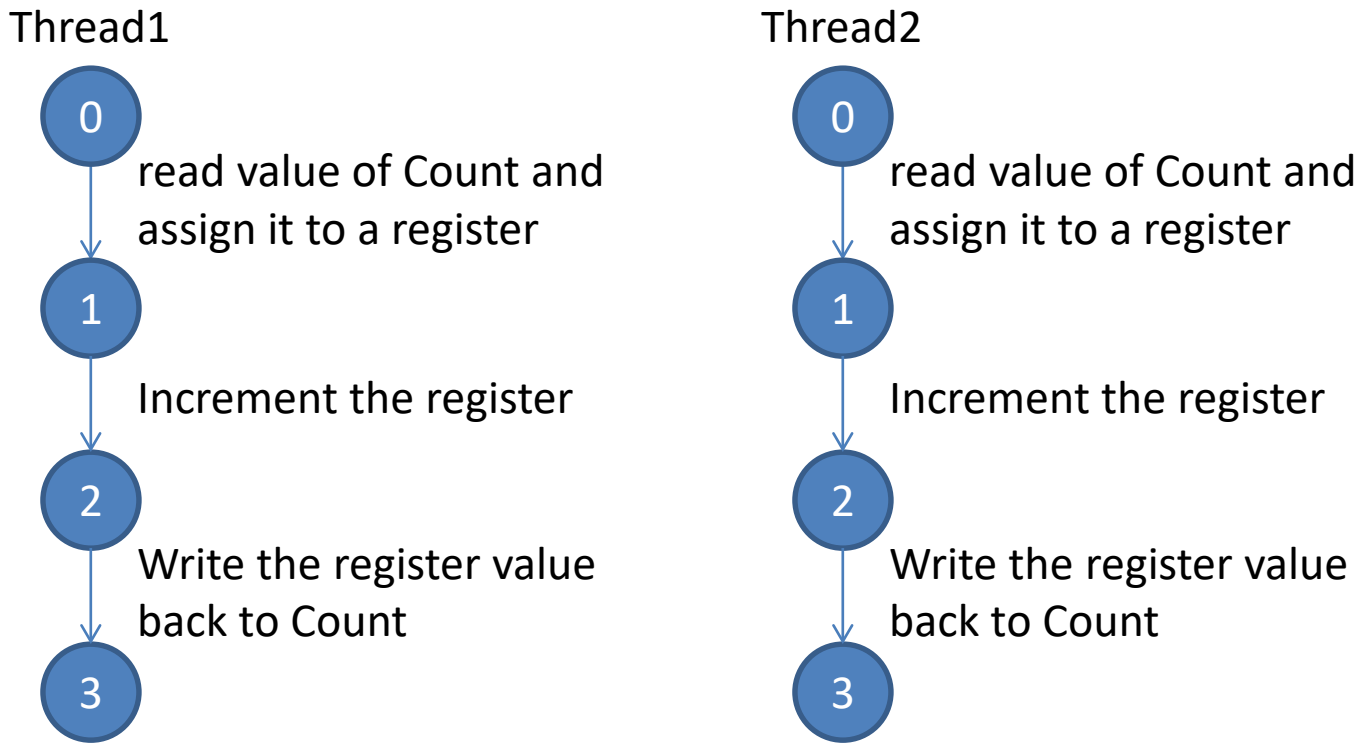
The scheduler is ‘un-predictable’

Is This Real?



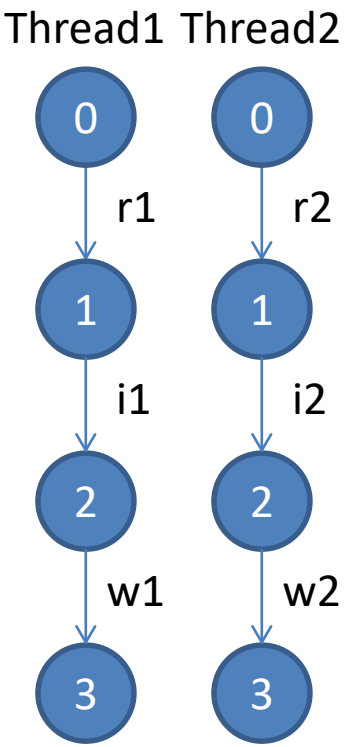
This is assuming that count++ is one step. Or is it?

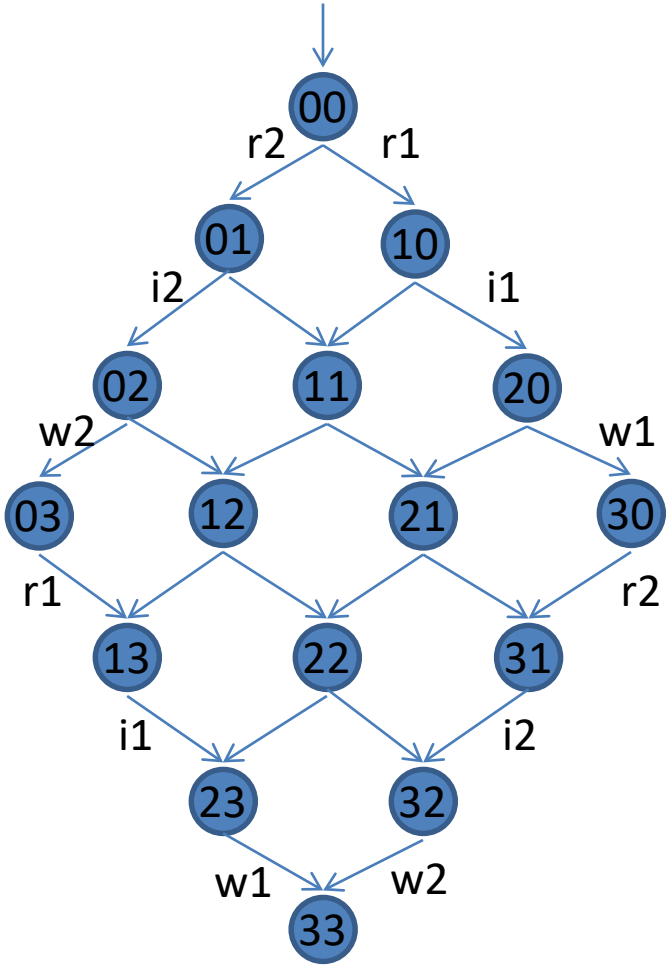
What Really Happened?



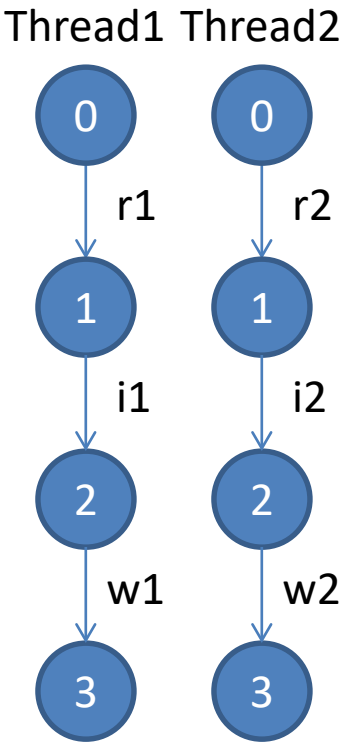
For double type, even read/write is not atomic!

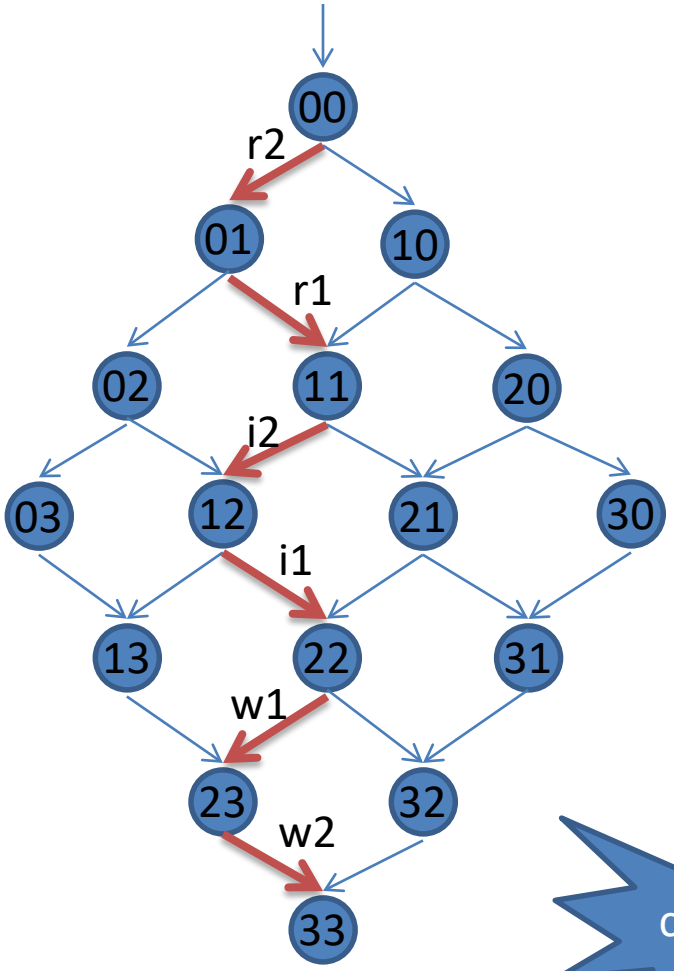
What Really Happened?

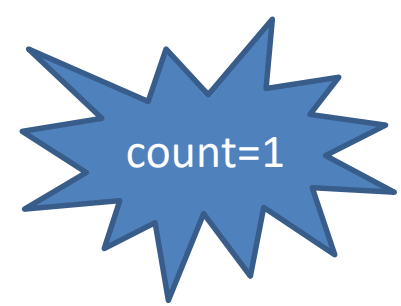




What Really Happened?







Is this correct?

A Joke

Some people, when confronted with a problem, think, 'I know, I'll use threads' - and then two they hav erpoblesms.

AtomicXXX

If the shared variable is of a type Boolean, int, int array, and etc., use classes in package java.util.concurrent.atomic

Example:

AtomicInteger x = new AtomicInteger(0)

x.incrementAndGet() //increments x by 1 atomically

Example: FirstErrorFixed.java

Compound Actions

Sometime it is not sufficient to simply use an AtomicXXX object.

```
//withdraw from a bank account
//check and update
if (amount >= 1000) {
        amount = amount - 1000;
}
```

Refer to example: SecondError.java

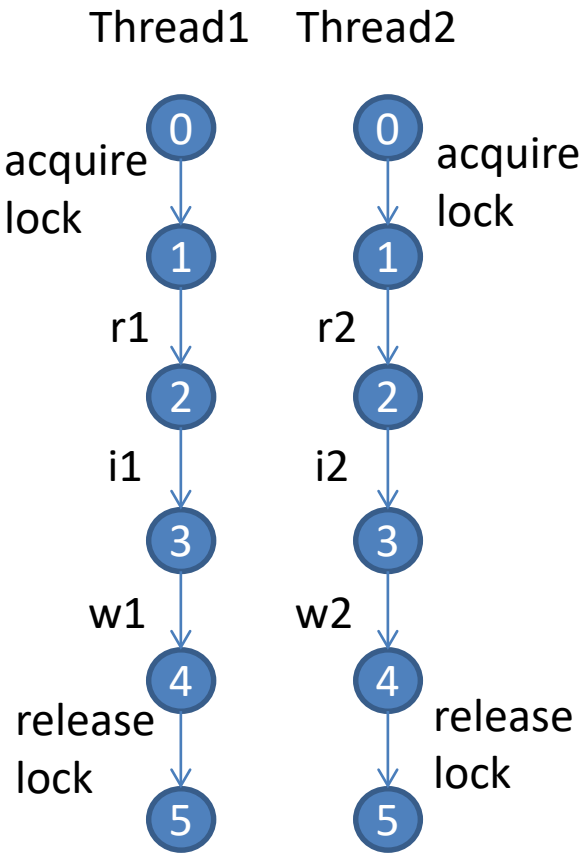
Intrinsic Locks

- Every Java object can implicitly act as a lock for purposes of synchronization.

```
synchronized (lock) {
        //Access shared state guarded by lock
}
```

- Intrinsic locks acts as mutexes (mutual exclusion locks), i.e., at most one thread may own the lock.
- Since only one thread at a time can execute a block of code guarded by a given lock, the synchronized blocks guarded by the same lock execute atomically with respect to one another.

How Lock Works



SecondErrorFixed.java

Cohort Exercise 3

Assuming that we would like to maintain that "saving + cash = 5000" always, fix the following class: LockStaticVariables.java.

Hint: Think about what is the lock?

if synchronise (this)
it will not work for the amount thing
cause there will be different locks, you are only
sync-ing your thread with ur thread.

wait() and notify()

- Busy waiting is not efficient
  - Consider a voting system with two threads. One collects votes and the other is waiting to count the votes when the voting is completed.

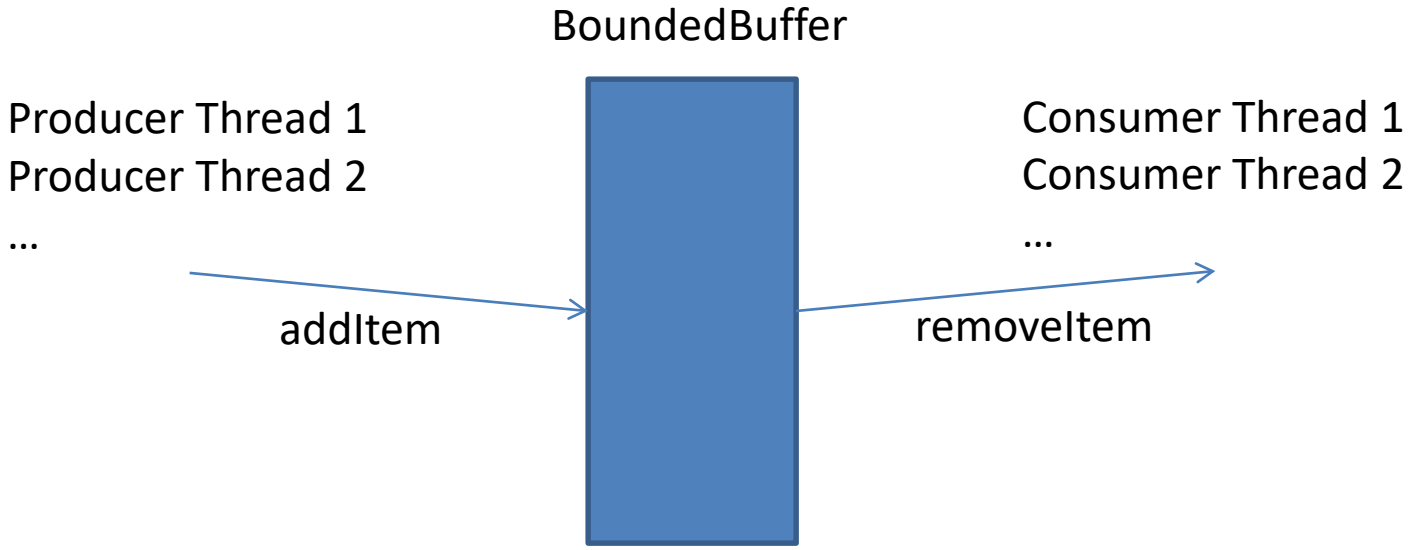
```
while (true) {
        synchronized(this) {
                if (votingComplete) {
                        break;
                }
        }
}
```

- Use wait()/nofityAll() to avoid busy waiting

Voting.java

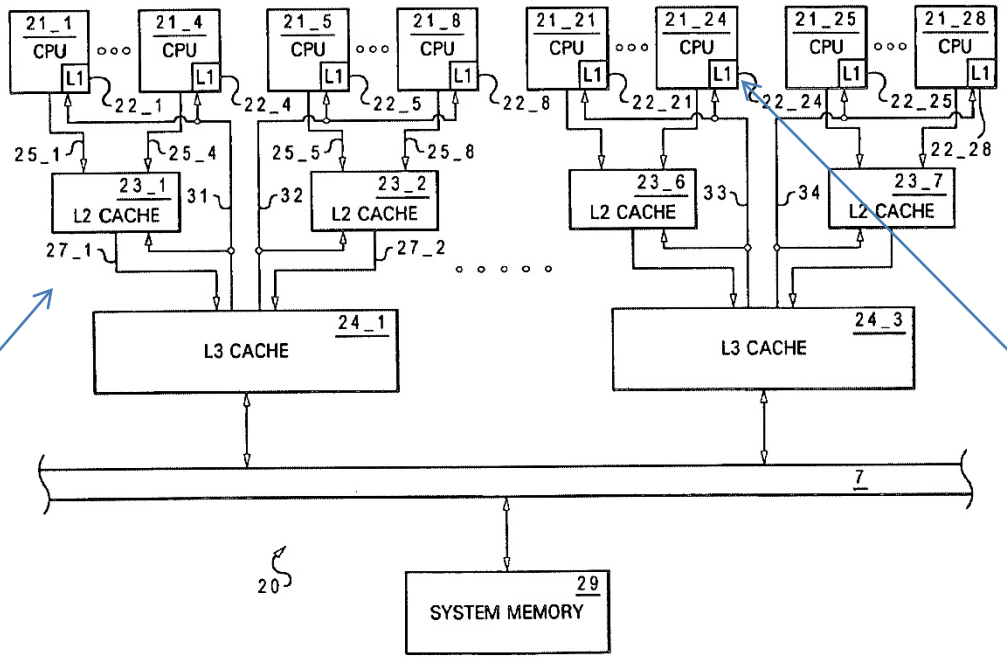
Cohort Exercise 4

- Producer/Consumer Pattern



- Exercise: fixed the Buffer class in BufferExample.java so that it is thread-safe and efficient

Problem 2: Visibility



new found

old found

How could we know where?

Example: FactorThread.java

Problem 3: Execution Ordering

Thread 1	Thread 2
1: r2 = A;	3: r1 = B;
2: B = 1;	4: A = 2;

- Initially A, B, r1 and r2 are all 0.
- What are the values of the variables after both threads complete?
- Is it possible to have B = 1 and r2 = 2 and A = 2 and r1 = 1?

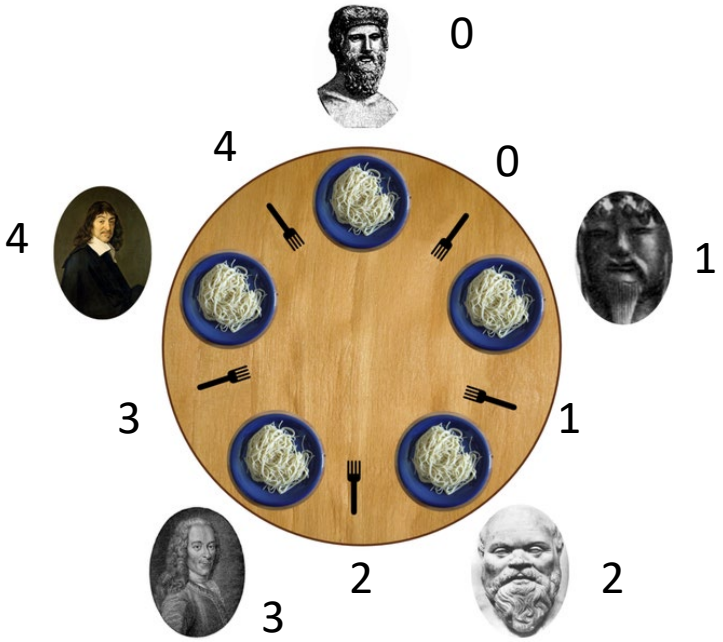
What are the order of execution?

- Java compiler might switch the order of sequential statements (e.g., for efficiency)
- Example: line 2 and line 3 might be switched

1. x++;
2. y++;
3. x++;

How could we know the order of execution?
Self-read: Java Memory Model

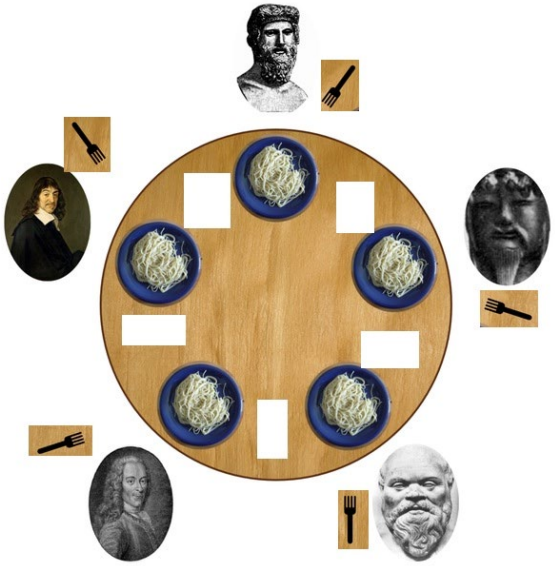
Problem 4: Deadlock



- Each philosopher needs two forks to eat.
- Each philosopher picks the one on the left first.

Deadlock

- Deadlock is the situation when two or more threads are both waiting for the others to complete, forever.



Cohort Exercise 5

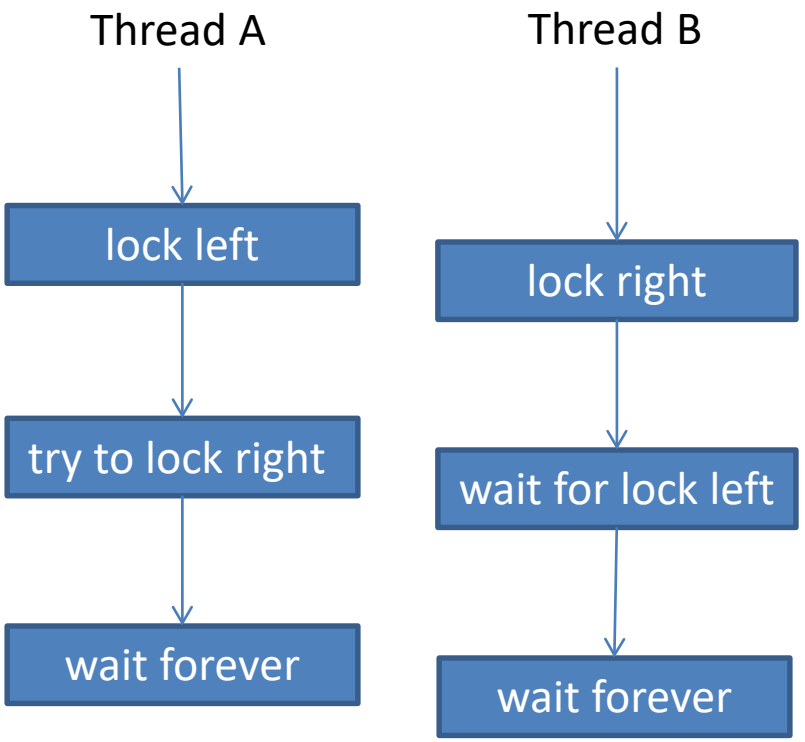
Given DiningPhil.java, modify it so as to demonstrate the deadlock.

Lock-Ordering Deadlock

```
public class LeftRightDeadlock {
        private final Object left = new Object ();
        private final Object right = new Object ();

        public void leftRight () {
                synchronized (left) {
                        synchronized (right) {
                                doSomething();
                        }
                }
        }

        public void rightLeft () {
                synchronized (right) {
                        synchronized (left) {
                                doSomethingElse();
                        }
                }
        }
}
```



Example

```
public void transferMoney (Account from, Account to, int amount) {
    synchronized (from) {
        synchronized (to) {
            if (from.getBalance() < amount) {
                //raiseException
            }
            else {
                from.debit(amount);
                to.credit(amount)
            }
        }
    }
}
```

deadlock when you try to transfer from A to B and B to A.

A->B
lock A
wait for B

B->A
lock B
wait for A

Is it deadlocking?

Example

```
public void transferMoney (Account from, Account to, int amount) {
    synchronized (from) {
        synchronized (to) {
            if (from.getBalance() < amount) {
                //raiseException
            }
            else {
                from.debit(amount);
                to.credit(amount)
            }
        }
    }
}
```

How can transferMoney deadlock?

Thread A: transferMoney(myAccount, yourAccount, 1)

Thread B: transferMoney(yourAccount, myAccount, 1)

Check out: DemonstrateDeadlock.java

Cohort Exercise 6

- Given DLExample.java, explain whether it is possibly deadlocking.
- Write a test case which potentially demos the deadlock.

to explain deadlock, need 2 threads and 2 locks
each thread, which is calling which methods, lock they hold

if only 1 lock, worst case is you having to wait till the other thread is done.

Other Liveness Hazards

- Deadlock is the most widely encountered liveness hazard.
- Starvation occurs when a thread is denied access to resources it needs in order to make progress.
  - Often caused by use of thread priority or executing infinite loops with a lock held.

Avoid using thread priority, since they increase platform dependence and can cause liveness problems.

Other Liveness Hazards (cont'd)

- Poor responsiveness
  - may be caused by poor lock management.
- Livelock: a thread, while not blocked, still cannot make progress because it keeps retrying an operation that will always fail.
  - e.g., when two overly polite people are walking in the opposite direction in a hallway.

Requirements

Multi-threaded programs have at least the following additional requirements.

- No race condition
- No visibility issue
- No execution ordering problem
- No deadlocks
- Efficiency!

How do we make sure our multi-threaded program satisfies these requirements?