Concurrency

COMS W4115

Prof. Stephen A. Edwards
Spring 2002
Columbia University
Department of Computer Science

Concurrency

Multiple, simultaneous execution contexts.

Want to “walk and chew gum at the same time” to

- Capture simultaneity of system structure

  E.g., Web Servers must deal with multiple, simultaneous, independent requests.

- Deal with independent physical devices

  The disk drive is delivering data while the network is delivering packets while the user is typing while. . .

- Increase performance

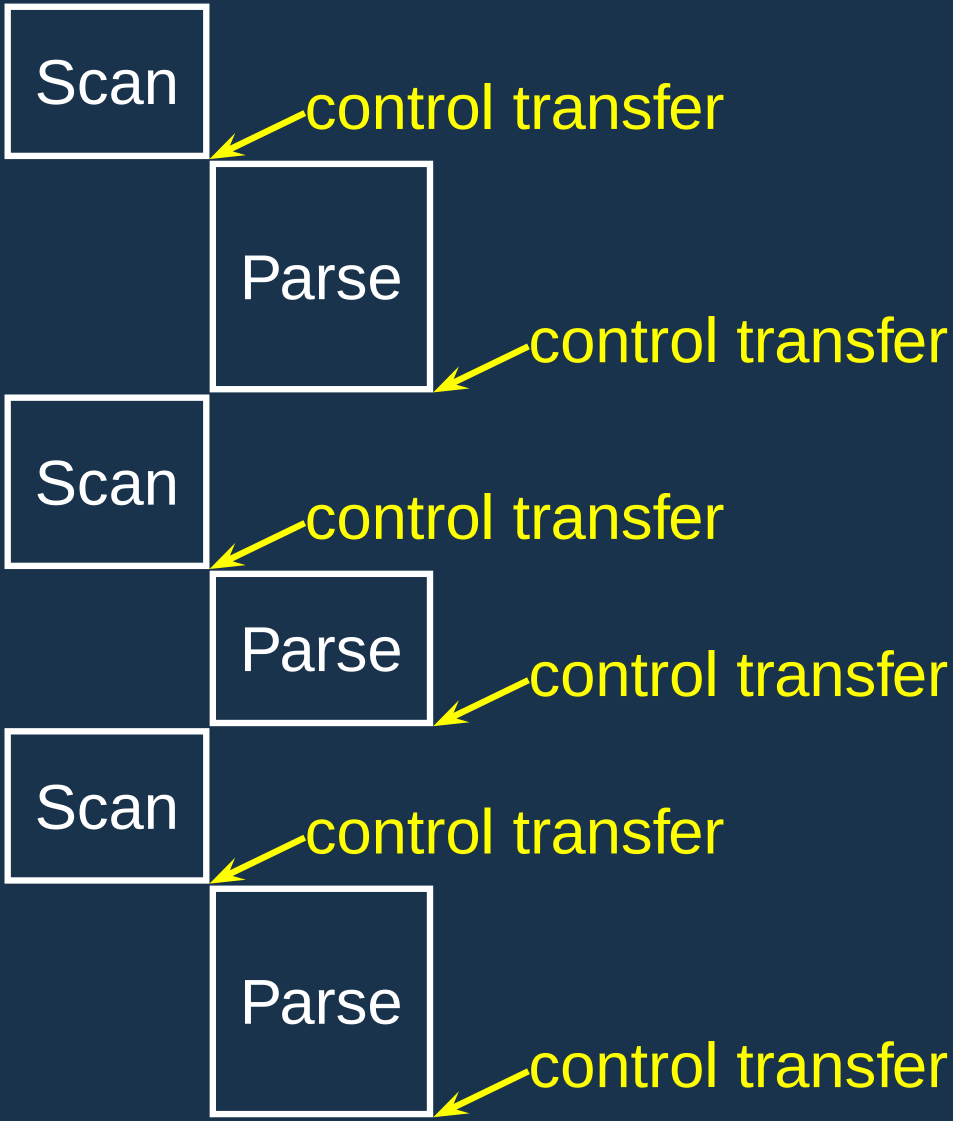
  Split the problem into parts and solve each on a separate processor

Coroutines

Basic idea: run two routines concurrently and let them trade control.

“Pick up where you left off”

Example: Lexer/parser



Coroutines

```
char c;
                      parse() {
void scan() {           char buf[10];
  c = ’s’;  <-----------transfer scan;
  transfer parse; ----->buf[0] = c;
  c = ’a’;  <-----------transfer scan;
  transfer parse; ----->buf[1] = c;
  c = ’e’;  <-----------transfer scan;
  transfer parse; ----->buf[2] = c;
}                     }
```

Implementing Coroutines

Languages such as C, C++, Java don’t have direct support.

Some libraries provide such a mechanism.

Challenge: Each couroutine needs a separate stack

Can be faked; often done.

Faking Coroutines in C

```
/* returns 0 1 .. 9 10 9 .. 1 0 0 .. */
int count() {

         int i;


    for ( i = 0 ; i < 10 ; i++ ) {
                 return i;

    }
    for ( i = 10 ; i > 0 ; i-- ) {
                 return i;

    }
    for (;;) {
                 return 0;

    }

}
```

Faking Coroutines in C

```
/* returns 0 1 .. 9 10 9 .. 1 0 0 .. */
int count() {
  static int state = 0;     /* program counter state */
  static int i;             /* use static, not automatic vars */
  switch (state) {
  case 0:
    for ( i = 0 ; i < 10 ; i++ ) {
      state = 1; return i;
  case 1: ;
    }
    for ( i = 10 ; i > 0 ; i-- ) {
      state = 2; return i;
  case 2: ;
    }
    for (;;) {
      state = 3; return 0;
  case 3: ;
    }
  }
}
```

Faking Coroutines in Java

Harder because it insists on more structure.

```
class Corout {
  int state = 0;
  int i;
  public int count() {
    switch (state) {
    case 0:
      i = 0;
    case 1:
      while (i < 10) { state = 1; return i++; }
      i = 10;
    case 2:
      while ( i > 0 ) { state = 2; return i--; }
    case 3:
      state = 3; return 0;
    }
    return 0;
  }
}
```

Cooperative Multitasking

Coroutines explicitly say when to context switch and who to run next.

Programmer completely responsible for scheduling.

Alternative: cooperative multitasking

Programs explicitly release control to operating system.

Operating system responsible for deciding which program runs next.

Cooperative Multitasking

Typical MacOS $<$ 10 or Windows $<$ 95 program:

Magical

```
void main() {
  Event e;
  while ( (e = get_next_event()) != QUIT ) {
    switch (e) {
      case CLICK: /* ... */ break;
      case DRAG: /* ... */ break;
      case DOUBLECLICK: /* ... */ break;
      case KEYDOWN: /* ... */ break;
      /* ... */
    }
  }
}
```

Cooperative Multitasking

Advantages:

- Frees the programmer from worrying about which other processes are running
- Cheap to implement.

Disadvantages:

- Malicious process may never call **`get_next_event`**.
- Programmer needs to add calls to long-executing event responses.
- Programmer still partially responsible for scheduling.

Multiprogramming History

First processors ran batch jobs: resident monitor loads one program, runs it, then loads the next.

Problem: I/O was slow, even by the standards of the time.

You’re wasting expensive cycles waiting for the punch card reader!

Solution: Multiprogramming with interrupt-driven I/O

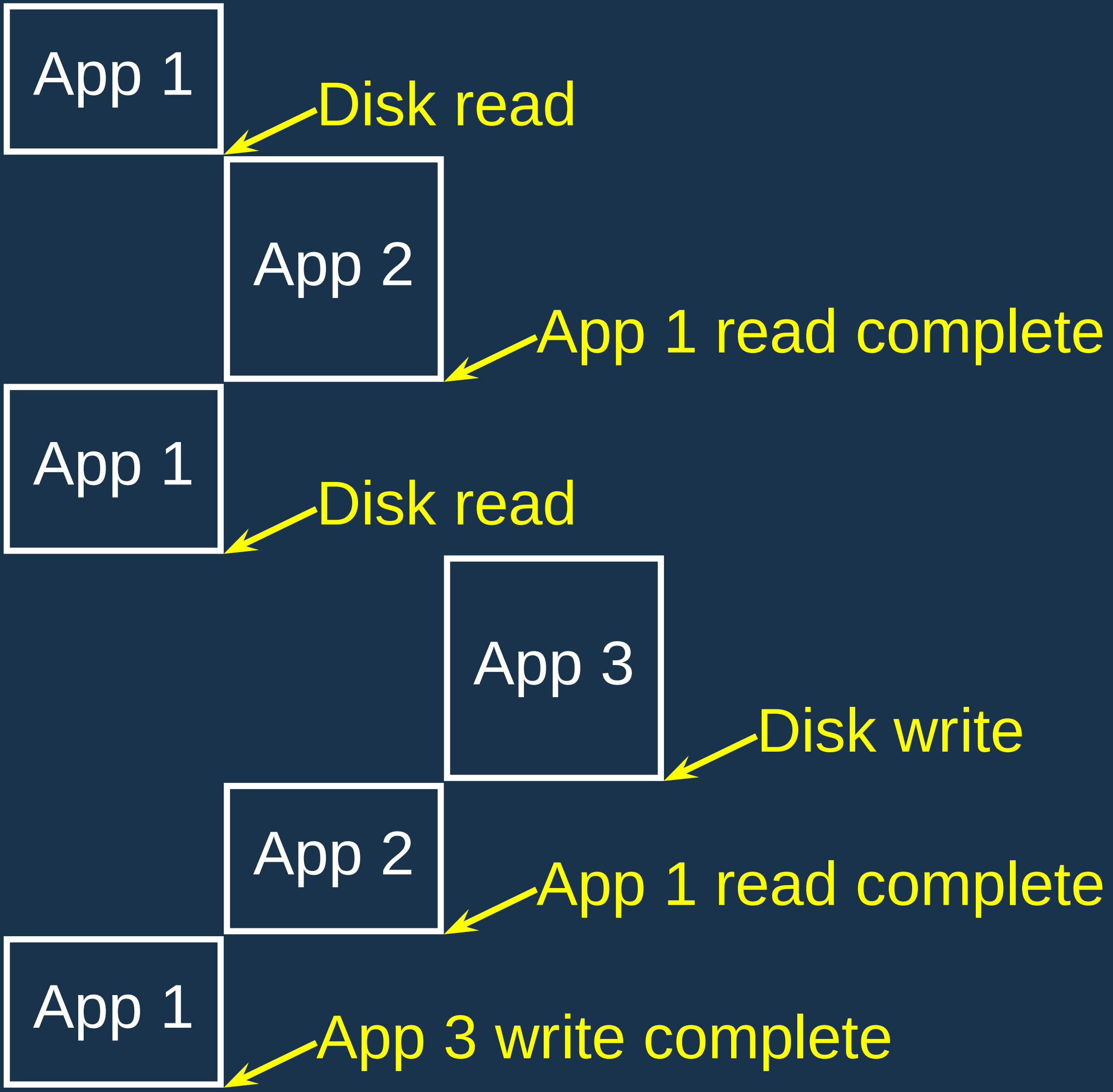
Multiprogramming

Avoids I/O busy waiting.

Context switch on I/O request.

I/O completion triggers interrupt.

Interrupt causes context switch.



Preemptive Multitasking

Idea: give the OS the power to interrupt any process.

Advantages:

- Programmer completely freed from thinking about scheduling: never needs to say “context switch.”
- Scheduler can enforce fairness: no process may monopolize processor

Disadvantages:

- Heavyweight: each process typically has own memory map (switching costly)
- Inter-program interaction now asynchronous: program may be interrupted anywhere

Timesharing

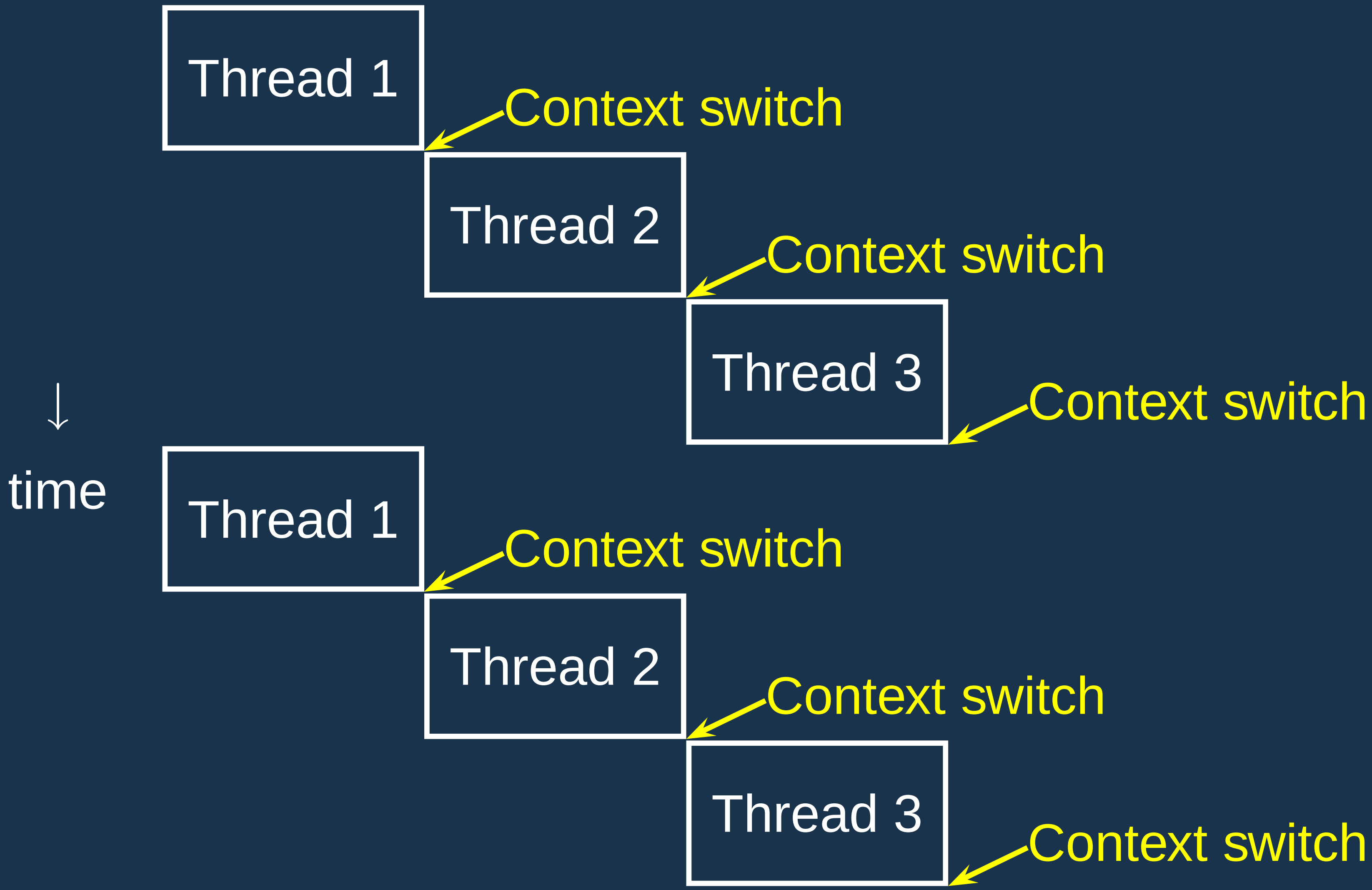
Model used on most modern operating systems (e.g., Unix 1970s, Windows/Mac 2000s)

System runs multiple threads. Each a separate execution context (registers, stack, memory).

Single-processor system has OS switch among threads. Each imagines it is running on its own computer.

Concurrent, but not simultaneous execution. Only one thread running at a time. Gives the impression of simultaneity.

Three Threads on a Uniprocessor



Concurrency Schemes Compared

	Scheduler	**Fair**	**Cost**
Coroutines	Program	No	Low
Cooperative Multitasking	Program/OS	No	Medium
Multiprogramming	OS	No	Medium
Preemptive Multitasking	OS	Yes	High

Java’s Support for Concurrency

Concurrency Support in Java

Based on preemptive multitasking.

Threads and synchronization part of language.

Model: multiple program counters sharing a memory space. Separate stacks.

All objects can be shared among threads.

Fundamentally nondeterministic, but language provides some facilities for avoiding it.

Thread Basics

Creating a thread:

```
class MyThread extends Thread {
  public void run() {
    /* thread body */
  }
}

MyThread mt = new MyThread(); // Create the thread
mt.start();   // Invoke run, return immediately
```

Thread Basics

A thread is a separate program counter with its own stack and local variables.

It is *not* an object: the **`Thread`** class is just a way to start a thread.

A thread has no sense of ownership: classes, objects, methods, etc. do not belong to any particular thread.

Any method may be executed by one or more threads, even simultaneously.

Suspension: The Sleep Method

```
public void run() {
  for(;;) {
    try {
      sleep(1000);   // Pause for 1 second
    } catch (InterruptedException e) {
       return;         // Caused by thread.interrupt()
    }
    System.out.println("Tick");
  }
}
```

Sleep

Does this print Tick once a second? No.

`sleep()` delay is a lower bound

Rest of the loop takes an indeterminate amount of time.

```
public void run() {
  for(;;) {
    try {
      sleep(1000);
    } catch (InterruptedException e) {
       return;
    }
    System.out.println("Tick");
  }
}
```

Races

In a concurrent world, always assume something else is accessing your objects.

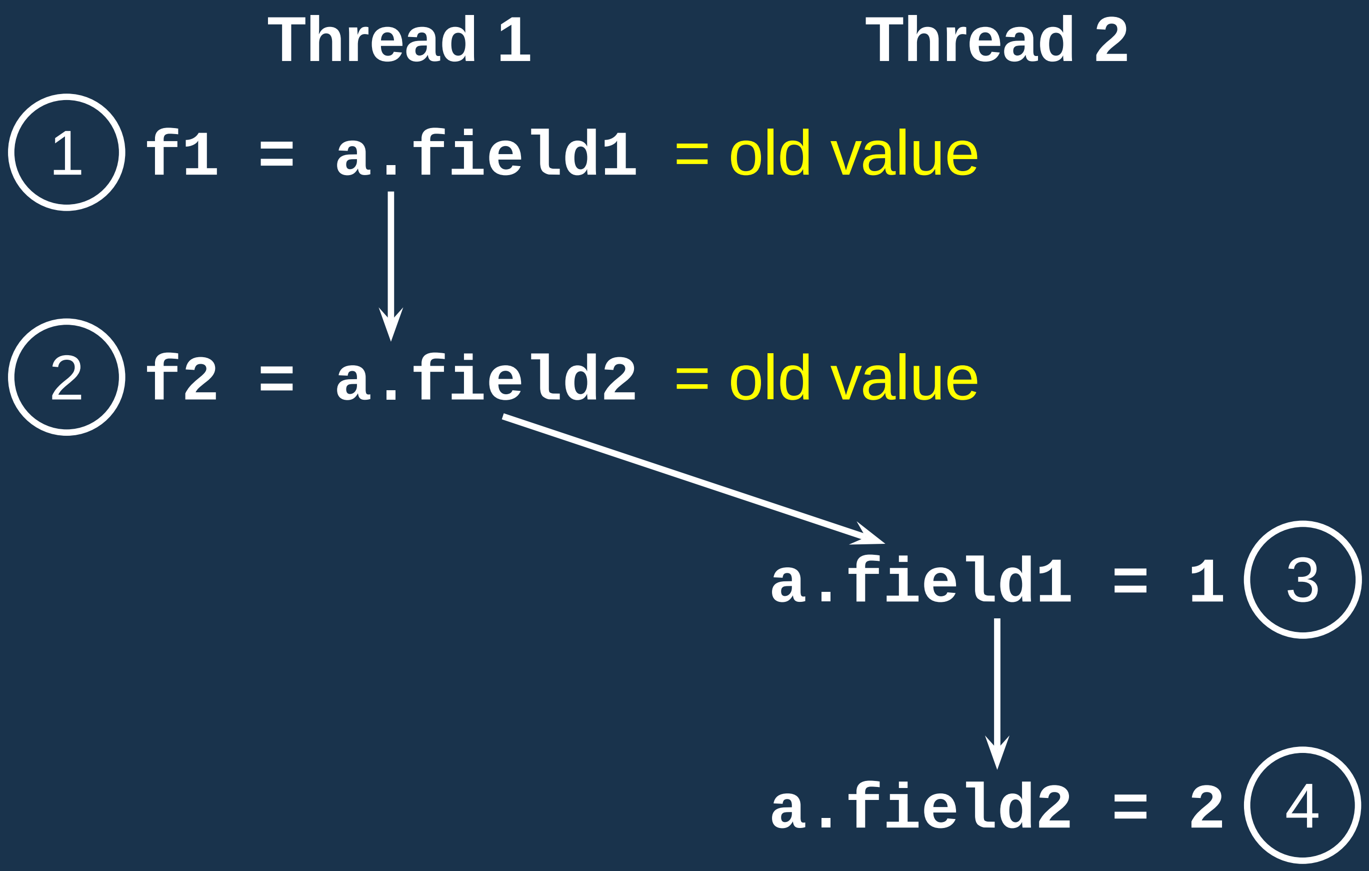
Other threads are your adversary

Consider what can happen when two threads are simultaneously reading and writing.

Thread 1	**Thread 2**
`f1 = a.field1`	`a.field1 = 1`
`f2 = a.field2`	`a.field2 = 2`

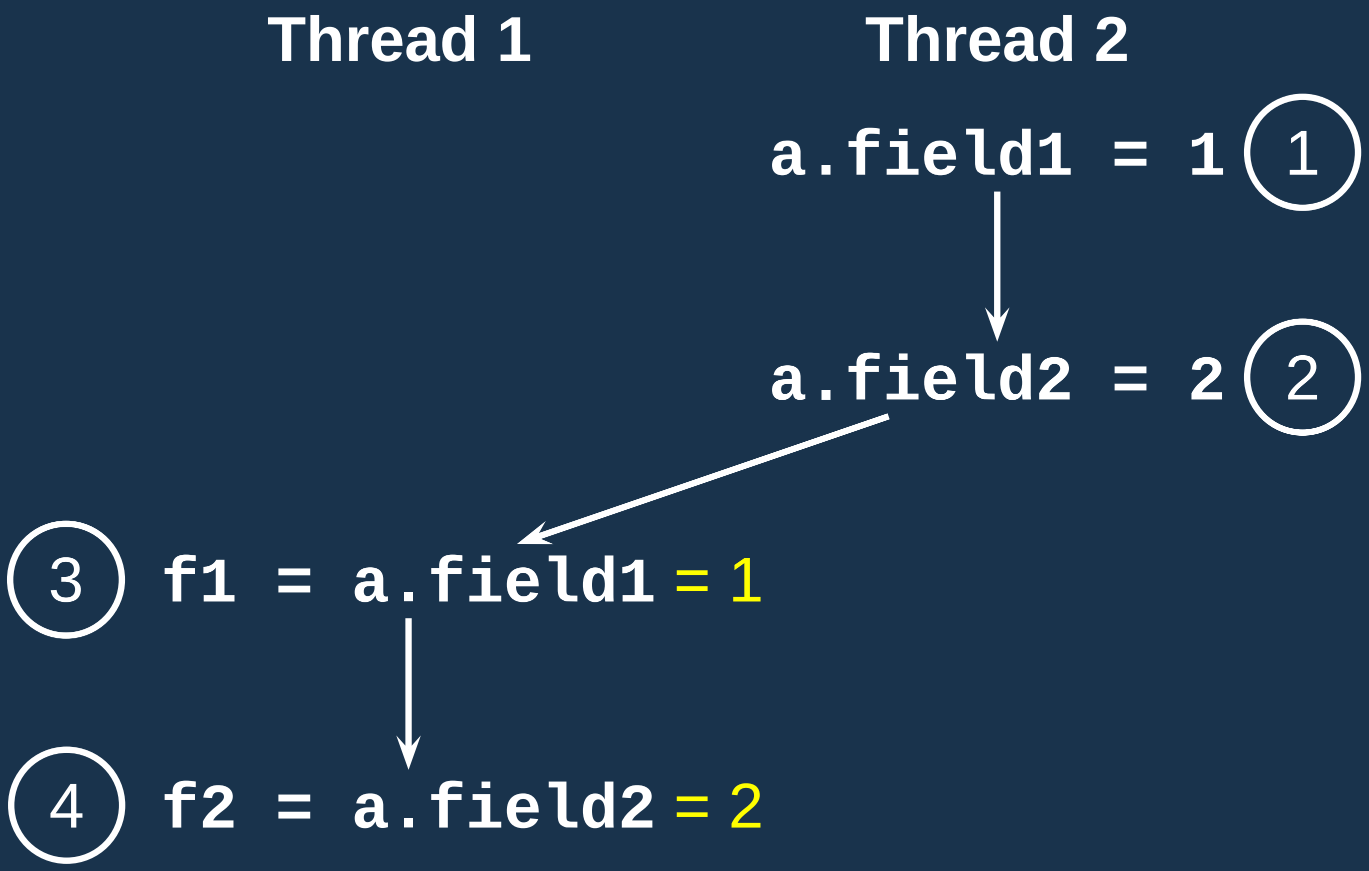
Thread 1 sees old values

Thread 1 runs before Thread 2



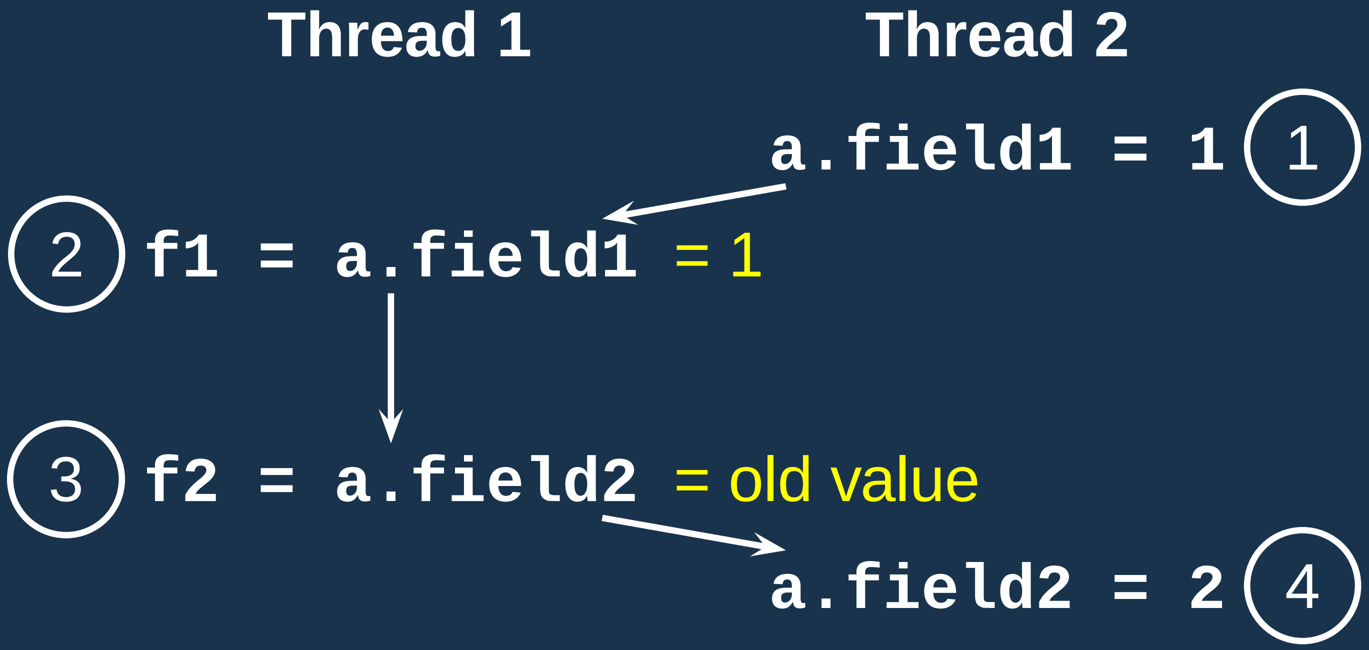
Thread 1 sees new values

Thread 1 runs after Thread 2



Thread 1 sees inconsistent values

Execution of Thread 1 interrupts execution of Thread 2



```
                                  a.field1 = 1
f1 = a.field1
f2 = a.field2
                                  a.field2 = 2
```

Non-atomic Operations

Biggest problem is the third case: reader thread sees partially-updated values

Might violate an invariant

Problem is non-atomic updates. Want

- no write to interrupt a read;
- no read to interrupt a write; and
- no write to interrupt a write.

Subtle Non-atomic Operations

Java assumes a 32-bit architecure

32-bit reads and writes are guaranteed atomic

64-bit operations may not be

```
int i; double d;
```

```
 Thread 1      Thread 2
 i = 10;       i = 20;
 d = 10.0;     d = 20.0;
```

`i` guaranteed to contain 10 or 20

`d` may contain garbage

(one word from 10.0, the other 20.0)

Locks: Making Things Atomic

Each object has a lock that may be owned by a thread

A thread waits if it attempts to acquire an lock already owned by another thread

The lock is a counter: a thread may lock an object twice

Non-atomic operations

```
class NonAtomCount {
  int c1 = 0, c2 = 2;

  public void count() { c1++; c2++; }

  public int readcount() { return c1 + c2; }
}
```

Invariant: **`readcount`** should return an even number.

Need both **`count`** and **`readcount`** to be atomic.

Synchronized Methods

```
class AtomCount {
  int c1 = 0, c2 = 2;

  public synchronized void count() {
     c1++; c2++;
  }

  public synchronized int readcount() {
     return c1 + c2;
  }
}
```

Grab lock while method running

Object's lock acquired when a **`synchronized`** method is invoked.

Lock released when method terminates.

Synchronized Methods

Marking a method synchronized is rather coarse

- Grabs the lock throughout the (potentially long) execution of the method. May block other threads.
- Only grabs the lock for its object. Can’t share a lock outside the object.

Alternative: The synchronized statement

The Synchronized Statement

```
public void myMethod() {
                                              Grab someobj's lock
  synchronized (someobj) {  ↙
    // quick operation that must be atomic
  } ← Release lock

  // take a long time
                                              Grab someobj's lock
  synchronized (someobj) {  ↙
    // quick operation that must be atomic
  } ← Release lock
}
```

The Synchronized Statement

Choice of object to lock is by convention; language/compiler is mute.

Responsibility of programmer to ensure proper synchronization.

Potentially every variable can be shared; compiler does not check for “missing” synchronized statements.

Difficult to get right: Java libraries from Sun still have thread-safety bugs.

Deadlock

```
1 synchronized (foo) {     synchronized (bar) { 2
    synchronized (bar) {     synchronized (foo) { 3
      // ...                   // ...
    }                        }
  }                        }
```

Moral: Always acquire locks in the same order.

Priorities

Each thread has a priority from 1 to 10 (5 is typical)

Scheduler’s job is to keep the highest-priority thread running

```
thread.setPriority(6)
```

What the Language Spec. Says

From *The Java Language Specification*,

> Every thread has a priority. When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion.

Vague enough?

Multiple Threads at the Same Priority?

Language gives implementer freedom

Calling **`yield()`** suspends the current thread to allow another at the same priority to run . . . maybe.

Solaris implementation runs threads until they stop themselves (**`wait()`**, **`yield()`**, etc.)

Windows implementation timeslices.

Starvation

Java does not provide a fair scheduler.

Higher-priority threads can consume all the resources and prevent threads from running.

This is *starvation*.

A timing dependent function of program, hardware, and implementation.

Waiting for a Condition

Say you want a thread to wait for a condition before proceeding.

An infinite loop might deadlock the system

```
while (!condition()) {}
```

Yielding avoids deadlock (probably), but is very inefficient.

```
while (!condition()) yield();
```

Thread reawakened frequently to check the condition: polling.

Java’s Solution: wait() and notify()

`wait()` is like **`yield()`**, but a waiting thread can only be reawakened by another thread.

Always in a loop; could be awakened be-fore condition is true

```
  while (!condition()) wait();
```

Thread that might affect the condition calls **`notify()`** to resume the thread.

Programmer’s responsible for ensuring each **`wait()`** has a matching **`notify()`**.

wait() and notify()

Each object maintains a set of threads that are waiting for its lock (its wait set).

```
synchronized (obj) {   // Acquire lock on obj
  obj.wait();          // Suspend and add this thread
                       // to obj’s wait set
}                      // Relinquish locks on obj
```

Other thread:

```
obj.notify();          // Awaken some waiting thread
```

wait() and notify()

Thread 1 acquires lock on obj

Thread 1 calls **`wait()`** on obj

Thread 1 releases lock on obj and adds itself to object's wait set.

Thread 2 calls **`notify()`** on obj (must have acquired lock)

Thread 1 is reawakened; it was in obj's wait set

Thread 1 reacquires lock on obj

Thread 1 continues from the **`wait()`**

wait() and notify()

Confusing enough?

`notify()` nondeterministically chooses one thread to reawaken (many may wait on the same object). So what happens where there’s more than one?

`notifyAll()` enables all waiting threads. Much safer.

Building a Blocking Buffer

```
class OnePlace {
  El value;

  public synchronized void write(El e) { .. }
  public synchronized El read() { .. }
}
```

Only one thread may read or write the buffer at any time

Thread will block on read if no data is available

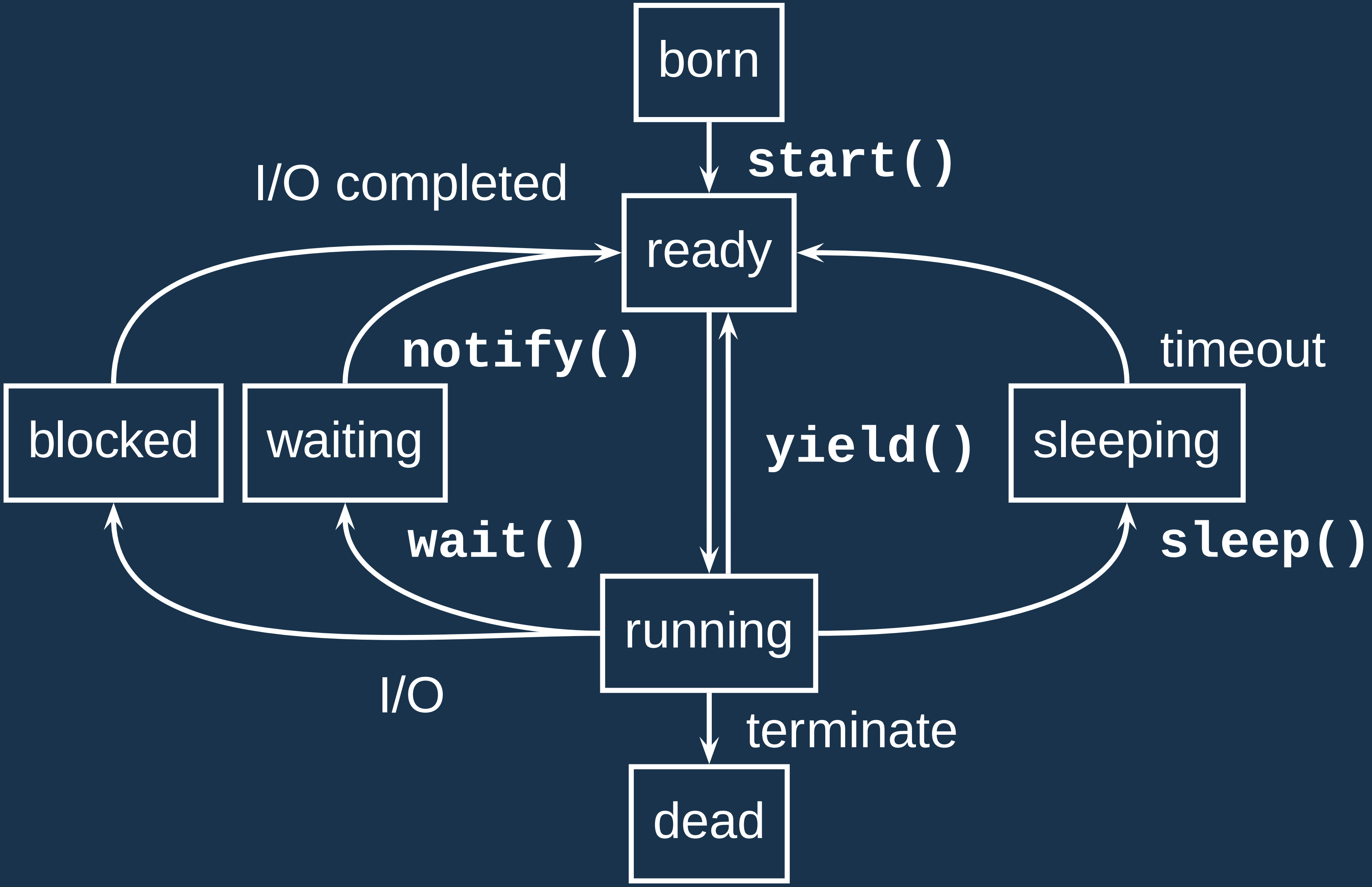
Thread will block on write if data has not been read

Building a Blocking Buffer

```
synchronized void write(El e)
   throws InterruptedException {
  while (value != null)
     wait();    // Block while full
  value = e;
  notifyAll();  // Awaken any waiting read
}

public synchronized El read()
   throws InterruptedException {
  while (value == null)
     wait();    // Block while empty
  El e = value; value = null;
  notifyAll();  // Awaken any waiting write
  return e;
}
```

Thread States



Other Approaches to Concurrency

co-begin/end

Statements in a Java block are composed sequentially

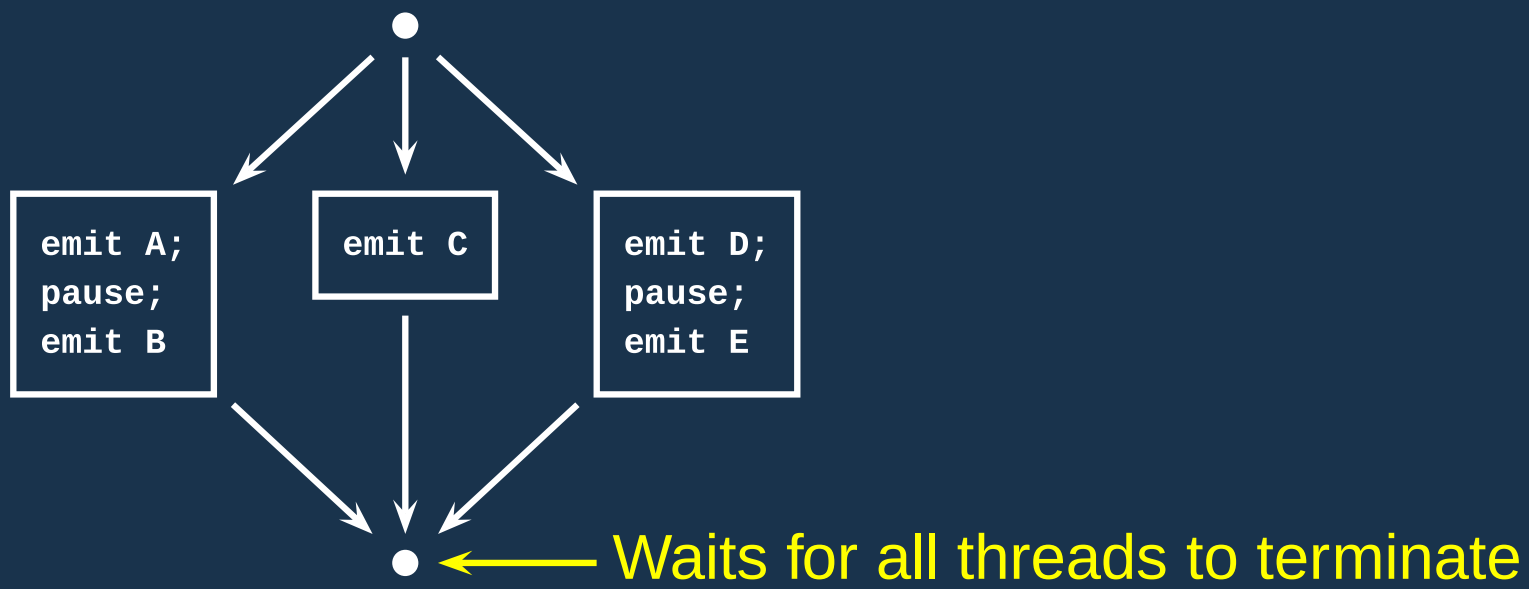
```
{
  a(); b(); c();
}
```

Other languages (e.g., Esterel) include concurrent composition:

```
  emit A; pause; emit B
||
  emit C
||
  emit D; pause; emit E
```

Concurrent Composition

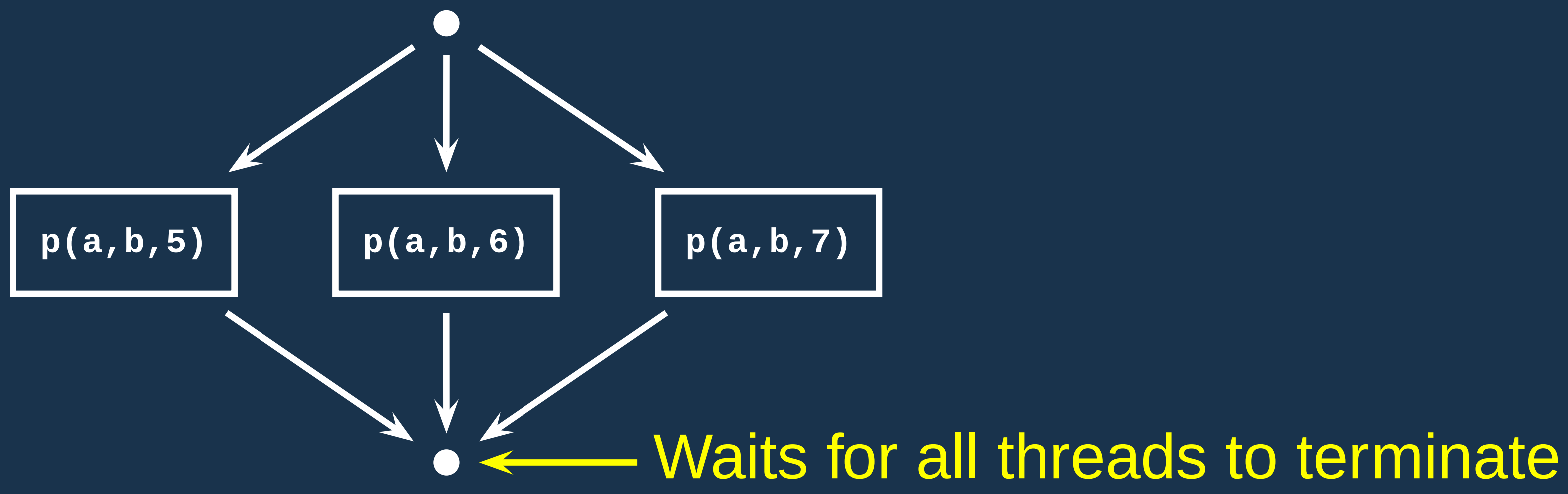
```
  emit A; pause; emit B
||
  emit C
||
  emit D; pause; emit E
```



Parallel Loops

SR (provides a parallel loop):

```
co (i := 5 to 7) ->
  p(a, b, i)
oc
```

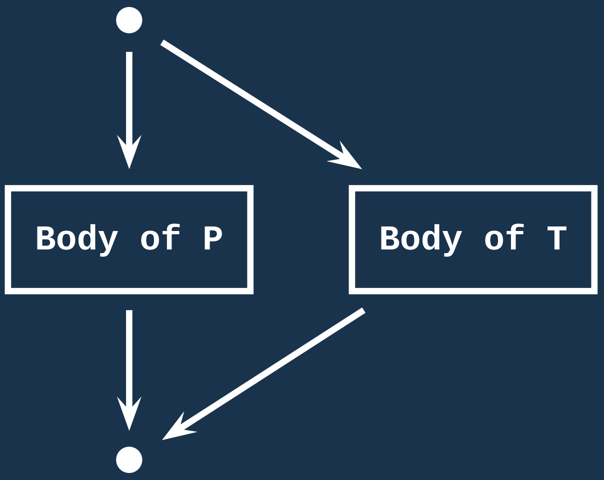


Launch-at-elaboration

A procedure can execute a task concurrently in Ada:

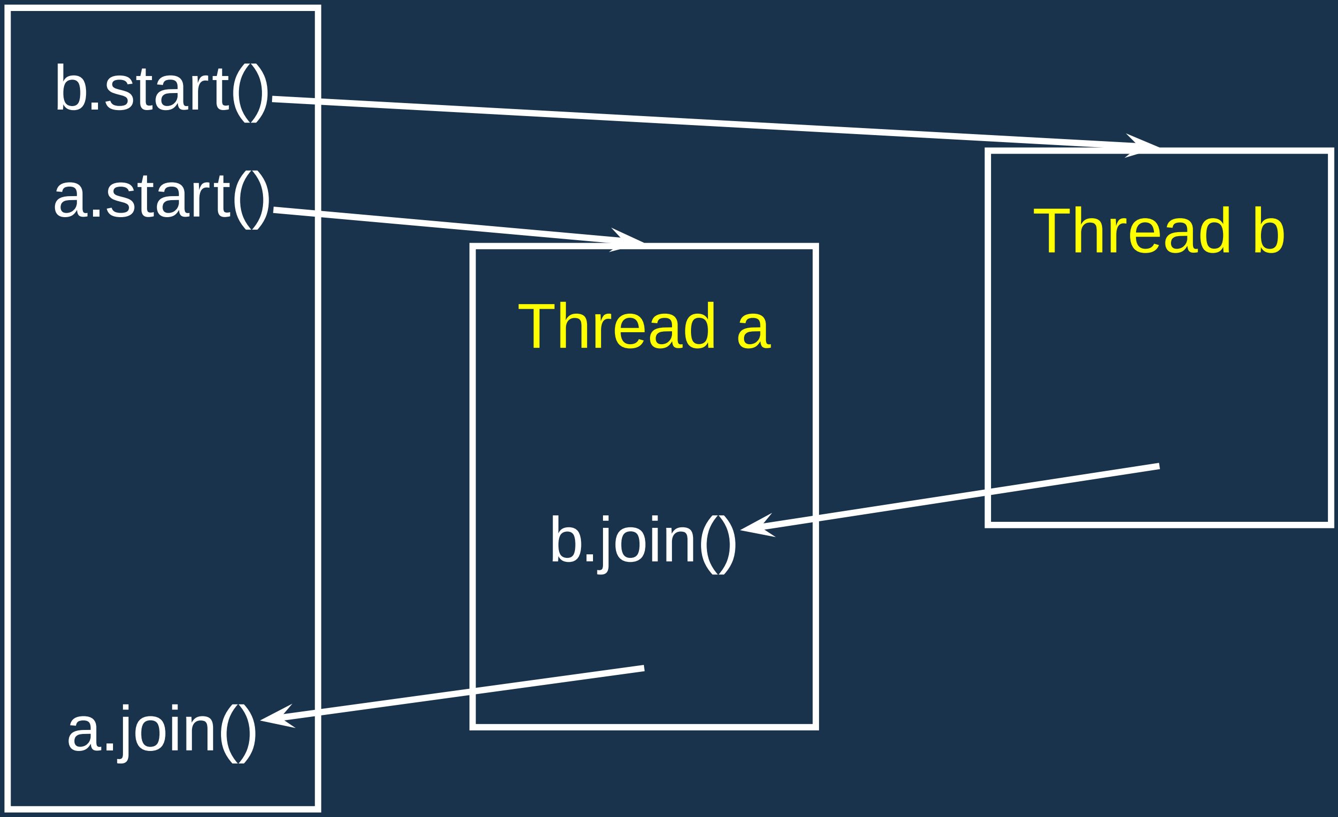
```
procedure P is
  task T is
        -- Body runs along with call of P
  end T;
begin
    -- Body of P
end P;
```

Invoking procedure P gives



Fork/Join

Java uses fork/join (actually start/join) to invoke and wait for threads. Permits nonnested behavior.



Implicit Receipt and the RPC Model

Normally, when you call a procedure in a program, that procedure is part of the same program:

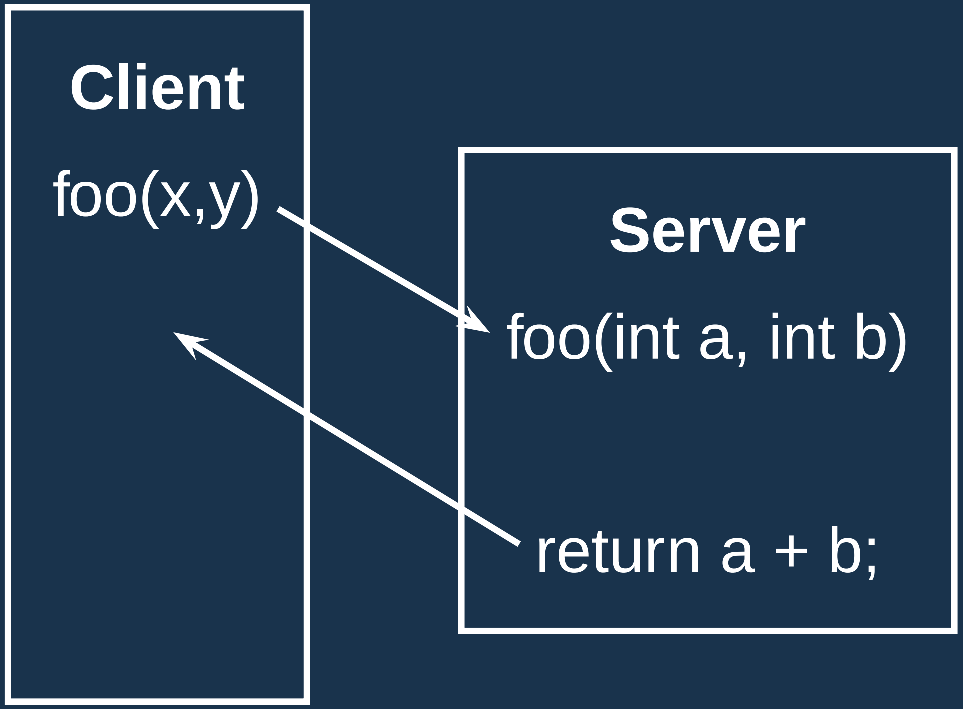
```
foo(x, y, z)
```

Remote procedure call modifies this to allow the procedure to be part of a different program on a different computer.

Rather than passing arguments on the stack and the return value in a register, RPC passes both over a network (e.g., using TCP).

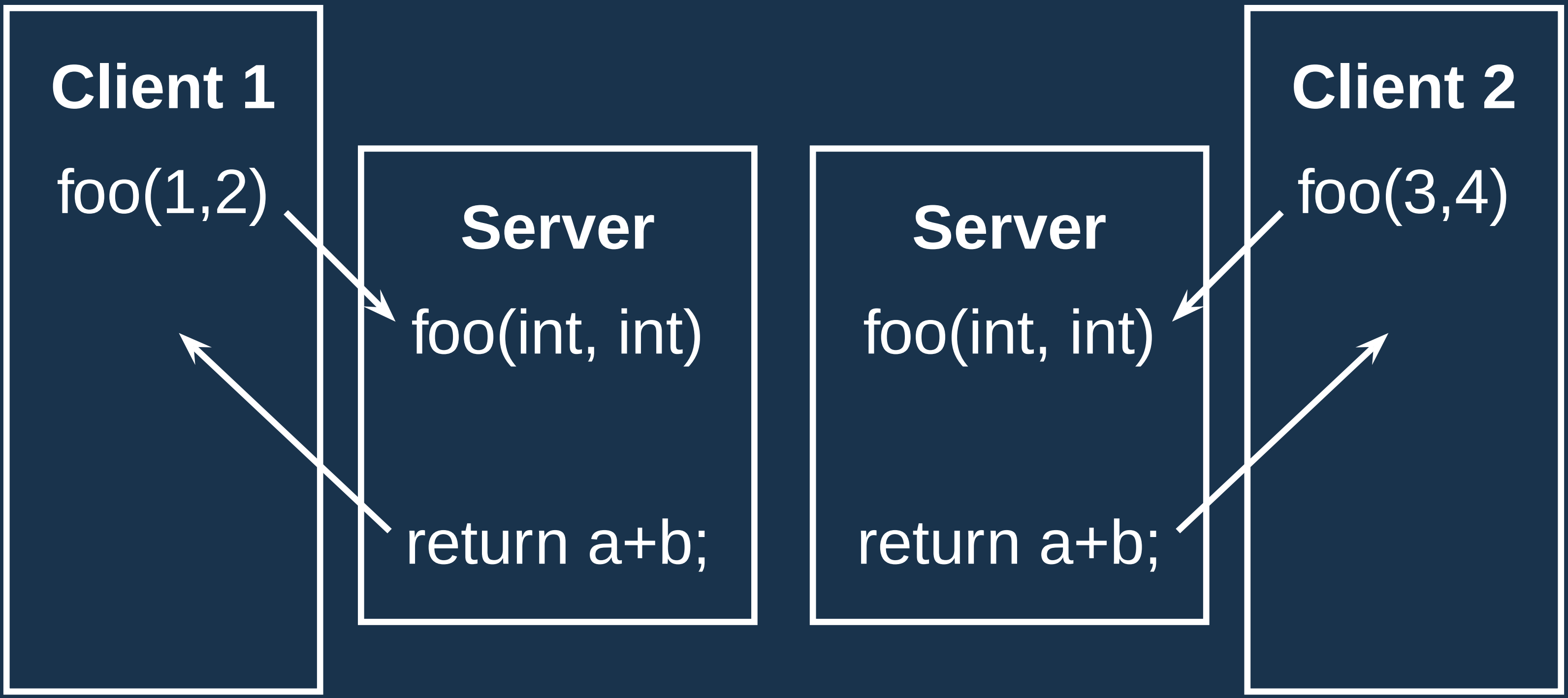
Implicit Receipt

This is a client/server model:



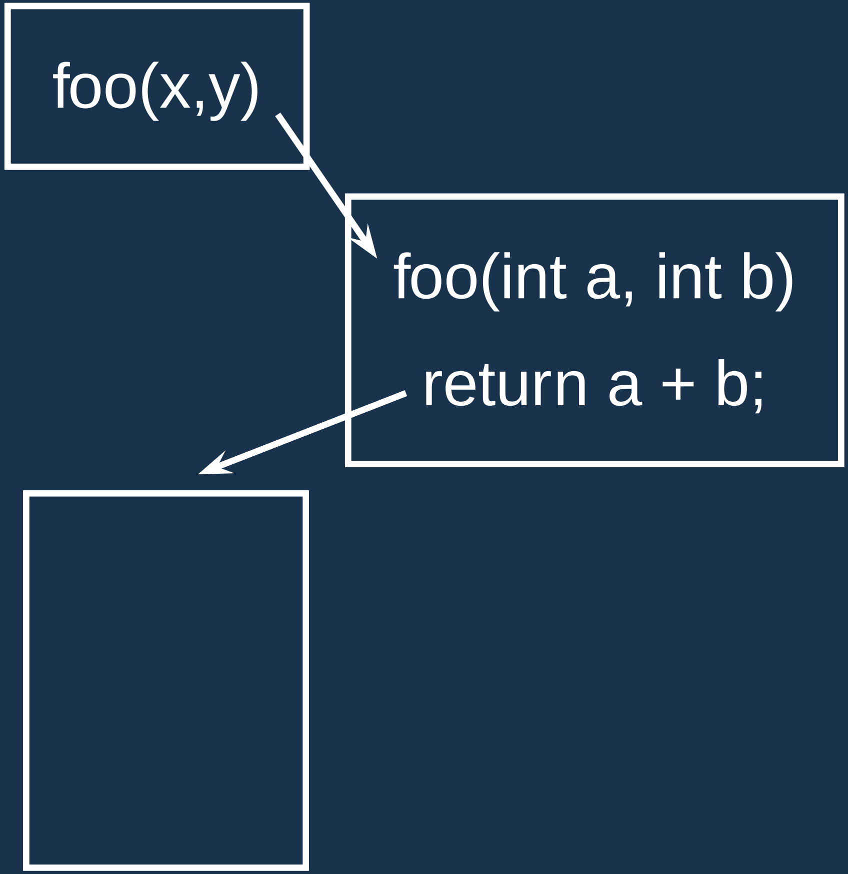
Implicit Receipt

The server generally allows multiple RPC requests at once. Each gets its own thread.



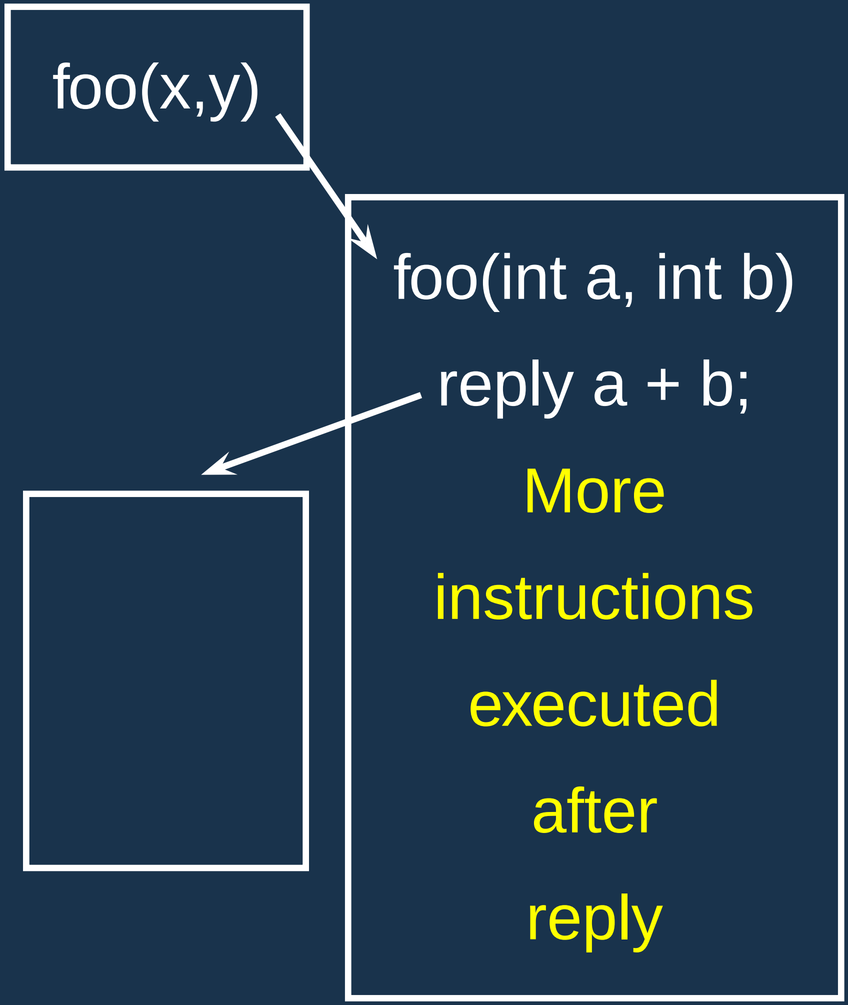
Early Reply

A procedure usually terminates when it returns.



Early Reply

But what if it didn't?


foo(x,y)
foo(int a, int b)
reply a + b;
More
instructions
executed
after
reply