CS5010: Lecture 9
Intro to Concurrency

Fall 2022

Brian Cross, Northeastern University

Additional credits: Ian Gorton & Abi Evans

Before we get started...

Clone the Code_from_Lectures repo from the course Github organization

https://github.ccs.neu.edu/cs5010seaF22/Code_From_Lectures

Then, open the Lecture9 project in the Evening_Lectures if you'd like to follow along.

Administrivia

- Homework #4
  - Group project
  - Due Monday, November 7^{th} by 11:59pm
  - Work in your team Group_<id1>_<id2> repo
- UML Draft Designs
- Codewalks 4 and 5
  - Combination of in-person and recorded
  - You'll want your team present for lectures to participate

Agenda

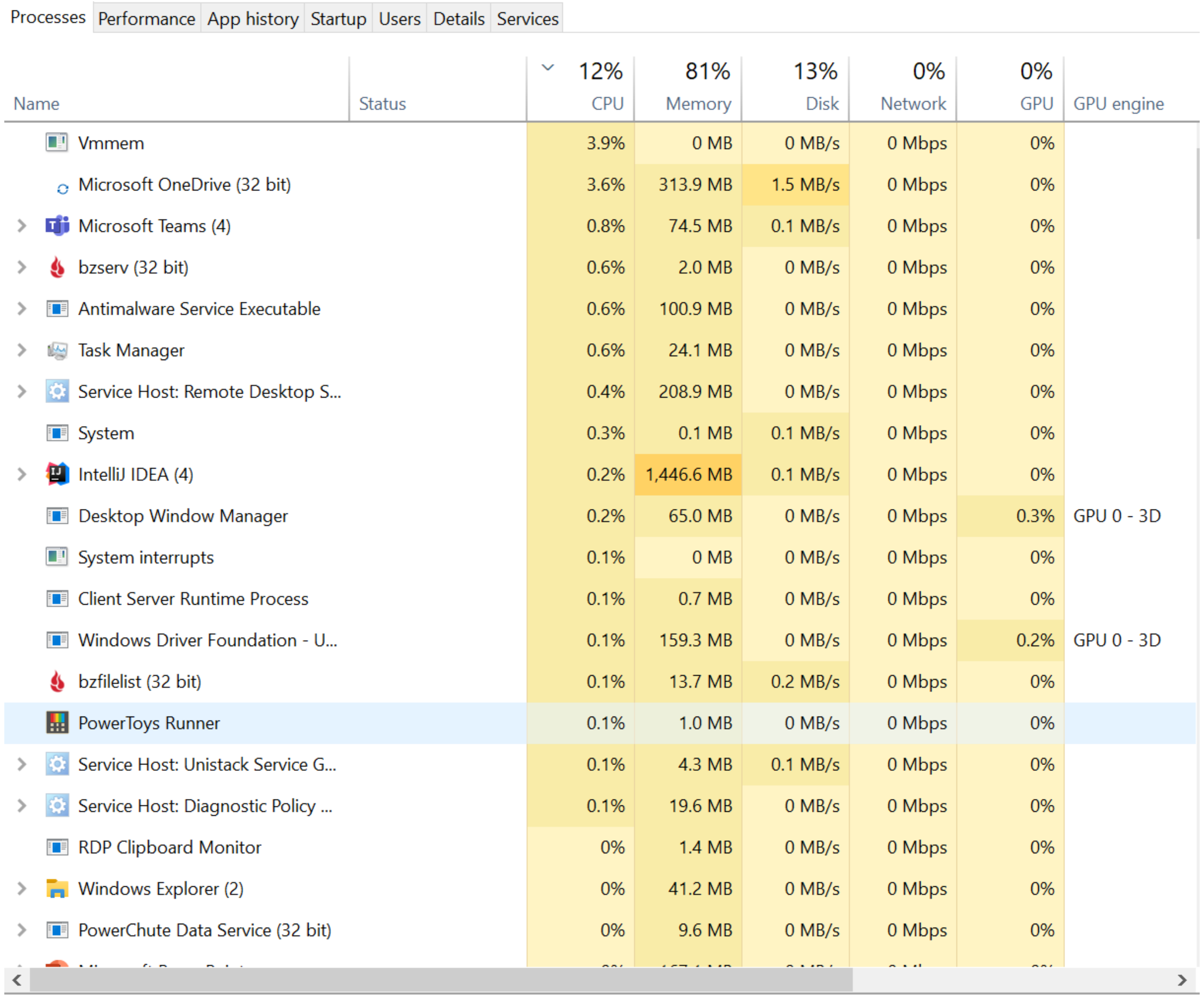
- Concurrency overview
- Simple threads in Java
- Race conditions
- Deadlock
- Producer-consumer problem
- Thread states
- Thread-safe collections

Concurrency

Concurrency

Doing multiple things at the same time

- Multiple applications
- Multiple processes within an application
- Distributed systems



Processes | Performance | App history | Startup | Users | Details | Services

Name	Status	12% CPU	81% Memory	13% Disk	0% Network	0% GPU	GPU engine
Vmmem		3.9%	0 MB	0 MB/s	0 Mbps	0%	
Microsoft OneDrive (32 bit)		3.6%	313.9 MB	1.5 MB/s	0 Mbps	0%	
Microsoft Teams (4)		0.8%	74.5 MB	0.1 MB/s	0 Mbps	0%	
bzserv (32 bit)		0.6%	2.0 MB	0 MB/s	0 Mbps	0%	
Antimalware Service Executable		0.6%	100.9 MB	0 MB/s	0 Mbps	0%	
Task Manager		0.6%	24.1 MB	0 MB/s	0 Mbps	0%	
Service Host: Remote Desktop S...		0.4%	208.9 MB	0 MB/s	0 Mbps	0%	
System		0.3%	0.1 MB	0.1 MB/s	0 Mbps	0%	
IntelliJ IDEA (4)		0.2%	1,446.6 MB	0.1 MB/s	0 Mbps	0%	
Desktop Window Manager		0.2%	65.0 MB	0 MB/s	0 Mbps	0.3%	GPU 0 - 3D
System interrupts		0.1%	0 MB	0 MB/s	0 Mbps	0%	
Client Server Runtime Process		0.1%	0.7 MB	0 MB/s	0 Mbps	0%	
Windows Driver Foundation - U...		0.1%	159.3 MB	0 MB/s	0 Mbps	0.2%	GPU 0 - 3D
bzfilelist (32 bit)		0.1%	13.7 MB	0.2 MB/s	0 Mbps	0%	
PowerToys Runner		0.1%	1.0 MB	0 MB/s	0 Mbps	0%	
Service Host: Unistack Service G...		0.1%	4.3 MB	0.1 MB/s	0 Mbps	0%	
Service Host: Diagnostic Policy ...		0.1%	19.6 MB	0 MB/s	0 Mbps	0%	
RDP Clipboard Monitor		0%	1.4 MB	0 MB/s	0 Mbps	0%	
Windows Explorer (2)		0%	41.2 MB	0 MB/s	0 Mbps	0%	
PowerChute Data Service (32 bit)		0%	9.6 MB	0 MB/s	0 Mbps	0%	

Concurrency

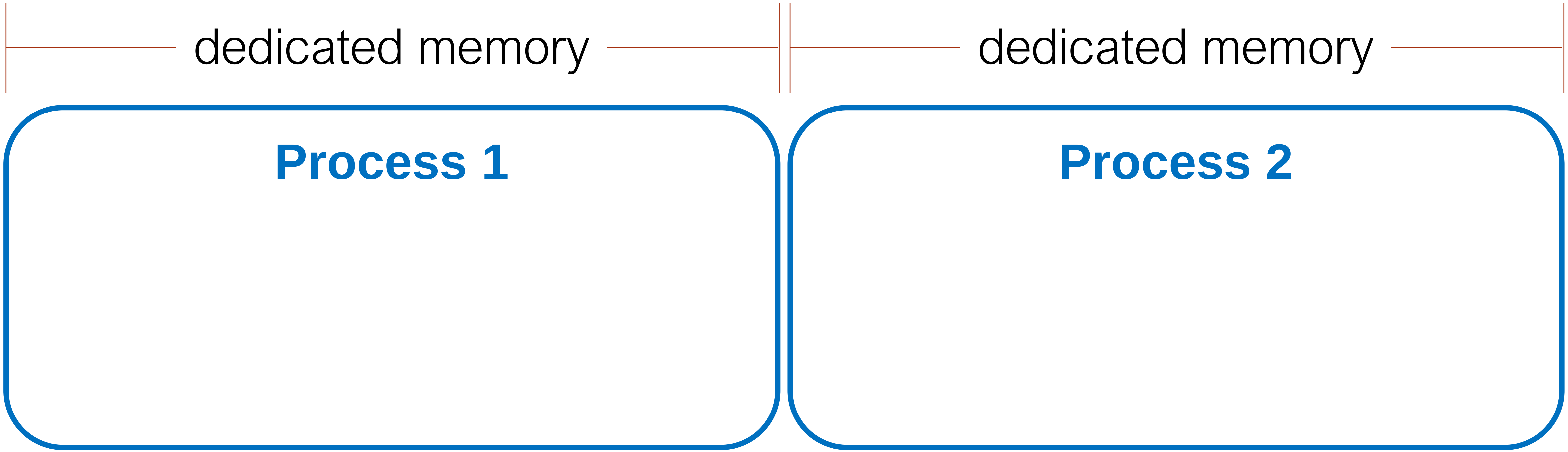
Key: One task should not block or slow another

- Tasks trigger/are triggered by events
- Event order is unpredictable
- Some tasks are time consuming e.g. reading large files
- System needs to stay ***responsive***

Processes and threads

Run a program → loads it in memory

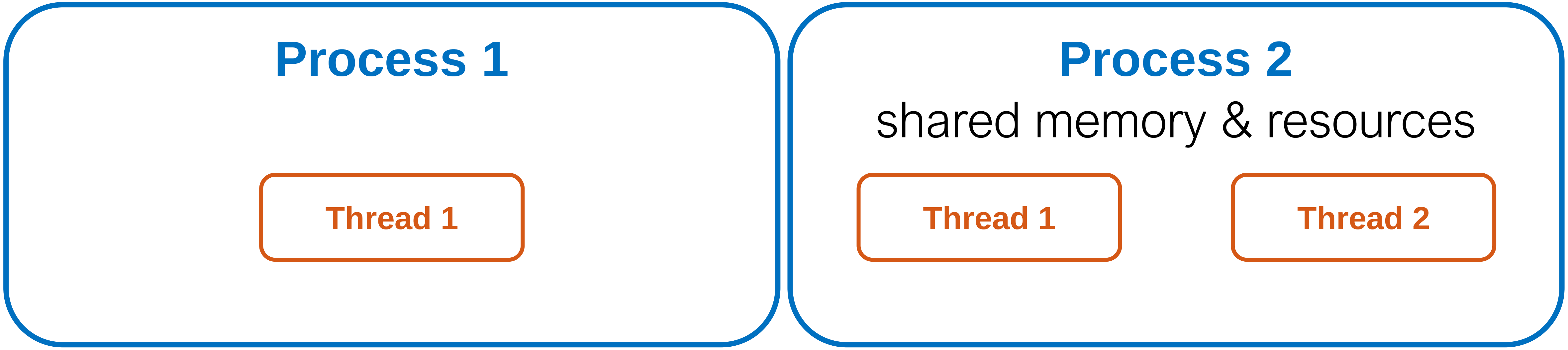
- Contains one or more **process**
  - Independent: can't directly access data / resources in other processes



Processes and threads

Run a program → loads it in memory

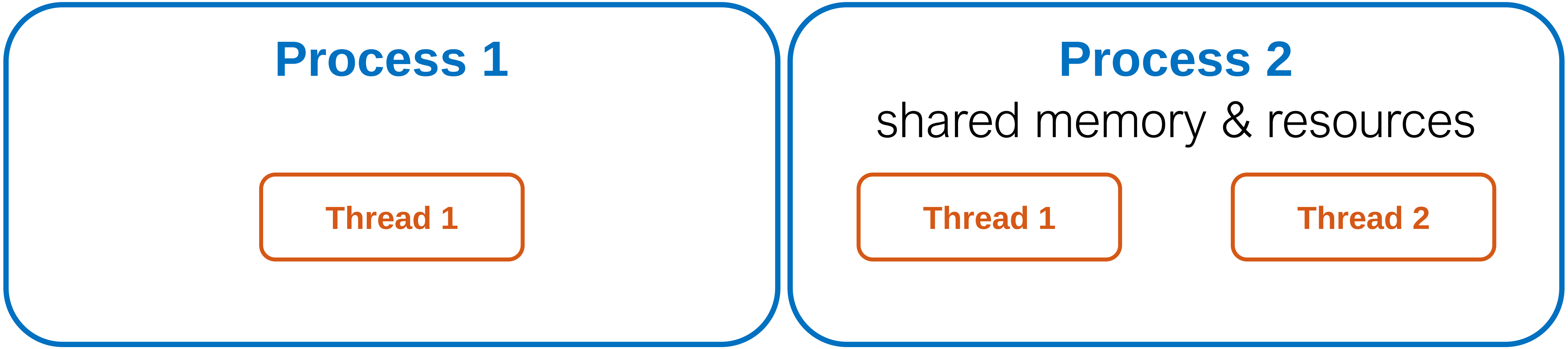
- Contains one or more **process**
- Each process has one or more **threads**



Processes and threads

Run a program → loads it in memory

- Contains one or more **process**
- Each process has one or more **threads**
- In Java, you'll mostly work with threads



Processes and threads example

All the processes and threads belonging to Firefox (with 8 tabs open)

Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	% GPU	GPU Time	PID	
followupd	0.0	0.51	2	0	0.0	0.00	757	ab
fmfd	0.0	11.55	3	0	0.0	0.00	587	ab
FirefoxCP WebExtensions	0.0	2:29.65	36	1	0.0	0.00	24242	ab
FirefoxCP Web Content	0.1	3:10.92	43	10	0.0	0.00	56660	ab
FirefoxCP Web Content	1.5	4:19.17	44	15	0.0	0.00	57258	ab
FirefoxCP Web Content	0.1	1:02.62	41	2	0.0	0.00	56982	ab
FirefoxCP Web Content	0.0	29.48	39	2	0.0	0.00	57331	ab
FirefoxCP Web Content	0.0	2:49.35	46	2	0.0	0.00	57073	ab
FirefoxCP Web Content	0.6	1:46.70	43	3	0.0	0.00	57055	ab
FirefoxCP Web Content	0.1	29:42.15	50	3	0.0	0.00	24234	ab
FirefoxCP Web Content	0.2	6:59.58	50	78	0.0	0.00	25548	ab
Firefox	0.9	1:04:28.47	73	49	0.6	1:07:51.17	24232	ab

Note the number of threads in each process!

Threads

- **Lightweight** compared to processes
  - Threads within the same process share the same address space (memory)
  - Each thread has its own stack to support independent execution
- Communication between threads is easier than communication between processes
- Threads enable multi-tasking but shared resources mean its easy to make a mess!

The main thread

Every application has at least one thread—the *main thread*

- You don't need to create it
- The main thread can create other threads
    - (These have to be created)

Simple threads in Java

Creating a thread in Java

Two options:

1. Create a **`Runnable`** object, pass it to a `java.lang.`**`Thread`** object
2. Subclass `java.lang.`**`Thread`**, which implements **`Runnable`**

Both options:

- Must override **`Runnable`**`'s` **`run()`** method (*where* differs for each approach)
- Call **`Thread.start()`** → calls the Runnable object's **`run()`** method

Creating a thread by implementing Runnable

Create a new class implementing interface Runnable...

```
public class SimpleThread implements Runnable {
   @override
   void run() {
      // Do stuff when the thread is started e.g.
      System.out.println(“I’m a thread.”);
   }
}
```

Creating a thread by implementing Runnable

Create a new class implementing interface Runnable...

```
public class SimpleThread implements Runnable {
   @override
  void run() {
     // Do stuff when the thread is started e.g.
     System.out.println("I'm a thread.");
  }
}
```

The "driver" of the thread

Creating a thread by implementing Runnable

...pass an instance of the Runnable object to Thread

```
public class SomeClass {
  public class void main() {
    Thread t1 = new Thread(new SimpleThread());
  }
}
```

Creating a thread by implementing Runnable

...pass an instance of the Runnable object to Thread

```
public class SomeClass {
  public class void main() {
    Thread t1 = new Thread(new SimpleThread());
  }
}
```

Built-in class

Instance of an object which implements `Runnable`

Creating a thread by implementing Runnable

Call `Thread.start()` to run the thread

```
public class SomeClass {
  public class void main() {
    Thread t1 = new Thread(new SimpleThread());
    t1.start();
  }
}
```

Calls the run() method of the object passed to the Thread constructor

In this example, `SimpleThread.run()`

Creating a thread by subclassing Thread

Create a class that extends `Thread` and override `run()`

```
public class MyThread extends Thread {
  @Override
  void run() {
    // Do stuff when the thread is started e.g.
    System.out.println("I'm also a thread.");
  }
}
```

Creating a thread by subclassing Thread

Instantiate it and call `Thread.start()` to run the thread

```
public class SomeClass {
  public class void main() {
    Thread t1 = new Thread(new SimpleThread());
    t1.start();

    MyThread t2 = new MyThread();
    t2.start();
  }
}
```

Calls the run() method of the Thread subclass

In this example, `MyThread.run()`

Which option to choose?

Implementing Runnable

- Slightly more code to write
- More flexible (Runnable object can still inherit another class)

- The best option most of the time

Extending Thread

- Less code
- Not flexible (can’t inherit any other class)

- Only for very simple applications

Reasons why threads are tricky #1

When there are multiple threads, exact order of execution is not guaranteed or consistent → interleaving

```java
public class RunnableThreadController {
    public static void main(String[] args) {
        System.out.println("This is the main thread starting. " + Thread.currentThread());
        Thread t0 = new Thread(new BasicRunnable());
        Thread t1 = new Thread(new BasicRunnable());
        Thread t2 = new Thread(new BasicRunnable());
        t0.start();
        t1.start();
        t2.start();
        int x = 10;
        int y = -5;
        int z = x + y;
        System.out.println("Time wasting... " + z);
        System.out.println("This is the main thread finishing. " + Thread.currentThread());
    }
}
```

`Thread.sleep()`

- Static method of the `java.lang.Thread` class
- Delays execution of the following code (within the thread)
- Must handle `InterruptedException`

Thread.sleep() example

From sleepy.SleepyRunnable.java

```
@Override
public void run() {
    while (true) {
        System.out.println("Hello from " + Thread.currentThread().getName() + ". Time for a nap...");
        try {
            Thread.sleep( millis: this.nap_time  * 1000);
        } catch (InterruptedException e) {
            System.out.println("I will stop napping then");
            return;
        }
        System.out.println(Thread.currentThread() + " is awake!");
    }
}
```

`Thread.sleep()` example

From sleepy.SleepyRunnable.java

```
@Override
public void run() {
    while (true) {
        System.out.println("Hello from " + Thread.currentThread().getName() + ". Time for a nap...");
        try {
            Thread.sleep( millis: this.nap_time  * 1000);
        } catch (InterruptedException e) {
            System.out.println("I will stop napping then");
            return;
        }
        System.out.println(Thread.currentThread() + " is awake!");
    }
}
```

Wait here for specified time...

...then execute what comes next

interrupt()

Force a thread to stop what it's doing → throws an InterruptedException

```
Thread t = new Thread(new MyThread());
t.start();
t.interrupt();
```

interrupt()

From sleepy.SleepyRunnable.java

```
@Override
public void run() {
    while (true) {
        System.out.println("Hello from " + Thread.currentThread().getName() + ". Time for a nap...");
        try {
            Thread.sleep( millis: this.nap_time  * 1000);
        } catch (InterruptedException e) {
            System.out.println("I will stop napping then");
            return;
        }
        System.out.println(Thread.currentThread() + " is awake!");
    }
}
```

Execute if `interrupt()` called on an instance of `SleepyRunnable`

Let’s take a look

SleepyRunnable

Joining threads with `thread.join()`

...forces one thread to wait until another thread completes its execution.

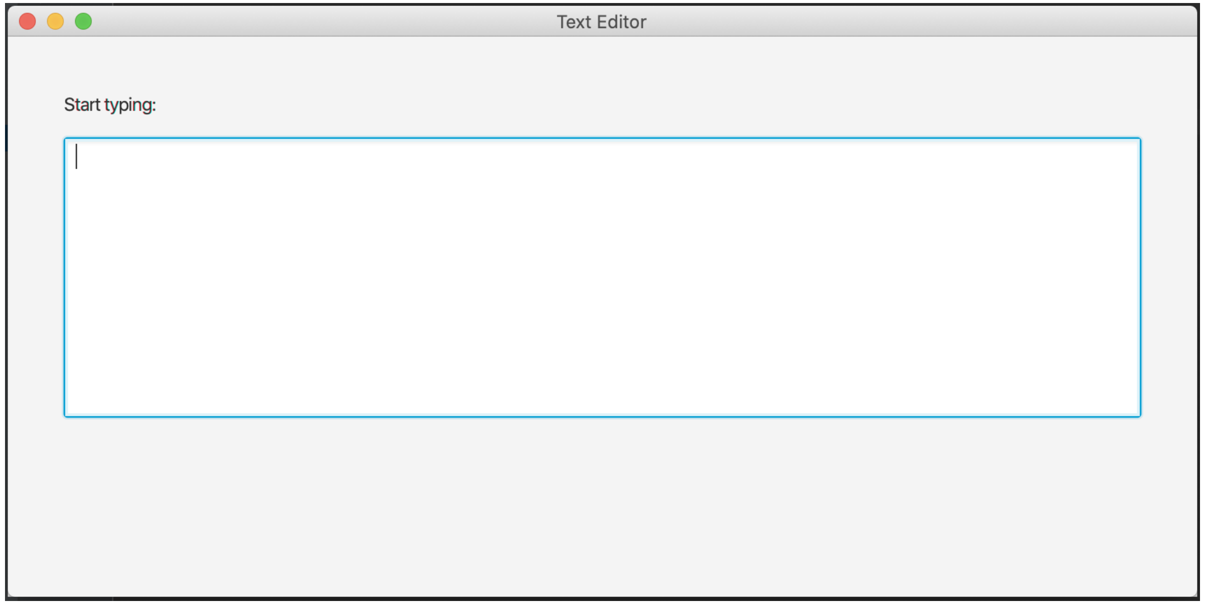
```
thread_to_wait_for.join();

Thread t1 = new Thread(new SomeThread());
Thread t2 = new Thread(new OtherThread());
Thread t3 = new Thread(new LastThread());
t1.start();
t2.start();
t2.join(); // will need to try-catch InterruptedException
t3.start(); // won't execute until t2 finishes
```

Example use case for threads and sleep

Basic auto-saving text editor

- Automatically save the contents of the editor every X seconds.



Basic auto-saving text editor

If we try using the main thread....

```java
public class Controller {
    @FXML
    TextArea textArea;

    @FXML
    public void initialize() { this.saveText(); }

    private void saveText() {
        while (true) {
            try {
                Thread.sleep( millis: 5000);
                System.out.println(this.textArea.getParagraphs());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Everything runs on the main thread

- (if we don't make new threads)

Sleeping this thread blocks anything from happening

Basic auto-saving text editor

Moving the autosave operation to its own thread keeps the main thread responsive

main thread

```java
public class Controller {
    @FXML
    TextArea textArea;


    @FXML
    public void initialize() {
        this.startSaver();
    }

    private void startSaver() {
        Thread saver = new Thread(new Saver(this.textArea));
        saver.start();
    }

}
```

autosave thread

```java
public class Saver implements Runnable {
    private TextArea textArea;

    public Saver(TextArea textArea) {
        this.textArea = textArea;
    }

    @Override
    public void run() {
        System.out.println("starting");
        while (true) {
            try {
                Thread.sleep(5000);
                System.out.println(this.textArea.getParagraphs());
            } catch (InterruptedException e) {
                return;
            }
        }

    }
}
```

Race conditions

Reasons why threads are tricky #2

Interleaving can cause **race conditions**

→When multiple threads share the same variables & change it at the same time

1. Read value from memory to register
2. Change value in register
3. Write register value back to memory:

    thread 1: x = x+6;

    thread 2: x = x+1

    The result after N operations?

Race conditions

Thread 1	Thread 2
Reads x into register	
Register value + 6	
Writes register value to x	
	Reads x into register
	Register value + 1
	Writes register value to x

Fine

Thread 1	Thread 2
Reads x into register	
Register value + 6	
	Reads x into register
	Register value + 1
	Writes register value to x
Writes register value to x	

Not fine

Race conditions

- Same program, different results
  - Depends on how CPU schedules execution
  - Different interleavings produce different outcomes

- Extremely hard to debug
  - Not reproducible
  - Extremely unpleasant when they occur in production systems!

Root cause: non-determinism

- Sequential programs exhibit deterministic behavior
  - We know what they going to do in which order
- Race conditions caused by **non-deterministic** behavior
  - We don’t know exactly what’s going to happen when
  - Non-determinism is not always a problem

Two kinds of non-determinism

Observable

Program may give different result (bad)

thread 1: x = x+6;

thread 2: x = x+1;

Non-observable

Program may execute differently, but result always the same (fine)

thread 1: a = 2; b = a =6;

thread 2: x = 9; y = x - 3

No shared variables → results are always the same

Another race condition example

A ticketing system

- Multiple simultaneous ticket requests for the same seat
  → if not properly handled, ticket could be oversold

Take a look at Code_from_Lectures > Lecture9 > racecondition

- **Can you see the problem?**

Ticketing system example

In TicketRace:

```java
public void takeSeat(String name) {
    if (!this.isTaken) {
        String message = "Taken by " + name;
        System.out.println(message);
        this.isTaken = true;
        this.customerName = name;
    }
}
```

In Customer (thread):

```java
@Override
public void run() {
    ticket.takeSeat(this.name);
}
```

Customer 0: not taken

Customer 1: not taken

Customer 0: takes seat

Customer 1: takes seat

Seat taken by: customer 1
(customer 0 is angry)

Let’s take a look

racecondition

Common causes for race conditions

Check-then-act

```java
public void takeSeat(String name) {
    if (!this.isTaken) {
        String message = "Taken by " + name;
        System.out.println(message);
        this.isTaken = true;
        this.customerName = name;
    }
}
```

Common causes for race conditions

Check-then-act

```java
public void takeSeat(String name) {
    if (!this.isTaken) {
        String message = "Taken by " + name;
        System.out.println(message);
        this.isTaken = true;
        this.customerName = name;
    }
}
```

Read-modify-write

See Code_from_Lectures > Lecture 9 > **raceconditon2**

- **Can you see the problem?**

Read-modify-write example

```java
public void incrementNumber() {
    int num = this.number;
    num++;
    System.out.println("waste time");
    this.number = num;
}
```

- Thread X: reads number
- Thread X: wastes some time
- Thread Y: reads number
- Thread X: writes new number
- Thread Y: wastes some time
- Thread Y: writes new number starting from *old* value

Let’s take a look

racecondition2

Why are race conditions hard to spot and debug?

- Both examples include extra code to slow processing down
- Simpler operations (e.g. `this.num++`) would work fine almost all of the time
- ...except when they don't

```java
public void takeSeat(String name) {
    if (!this.isTaken) {
        String message = "Taken by " + name;
        System.out.println(message);
        this.isTaken = true;
        this.customerName = name;
    }
}
```

```java
public void incrementNumber() {
    int num = this.number;
    num++;
    System.out.println("waste time");
    this.number = num;
}
```

Avoiding race conditions

Synchronization, atomic data types

Avoiding race conditions with locks

Use ”locks” to impose ordering constraints

- Lock shared variables → can only be accessed by one thread at a time
- Each thread wishing to access a variable;
  - Takes the lock—everyone else has to wait
  - Changes the variable
  - Releases the lock—other threads can access

Synchronized methods

Add `synchronized` to the shared method definition:

```
public synchronized void takeSeat(String name) {...}
public synchronized void incrementNumber() {...}
```

- Prevents multiple invocations of synchronized methods on the same object from interleaving
- Handles taking and releasing the “lock”
- **Critical section**: a block of code that can’t be accessed by more than one thread at a time

Monitor locks

- Each Java object has a **monitor**, which a thread can lock or unlock
- Synchronized methods automatically lock the monitor on invocation by a thread → the thread has the lock
- Any other threads attempting to lock the monitor are blocked until the lock is available

Let’s take another look

Adding **synchronized**

Synchronized: pros and cons

Pro:

- Very simple to use

Con:

- Can impact **liveness**—the ability of a program to do its work in a timely manner

Another option → Java's **atomic** data types

Atomic Variables

- Defined in `java.util.concurrent.atomic`
- "Lock-free thread-safe" versions of single variable types

```
Integer → AtomicInteger
Boolean → AtomicBoolean
```

...and more

Atomic Variables

Have unique methods...

```
private int number = 0;
→ private AtomicInteger number = new AtomicInteger(0);

this.number++;
→this.number.incrementAndGet();

return this.number;
→ return this.number.get();
```

Example: AtomicInteger

Run racecondition2.ReadModifyWrite.java

→unpredictable results

Run atomicsolution.ReadModifyWrite.java

→ consistent results

Let’s take a look

atomicsolution

Deadlock

Reasons why threads are tricky #3

Deadlock occurs when multiple threads are waiting on each other → no thread can move forward.

Classic example: the dining philosophers problem

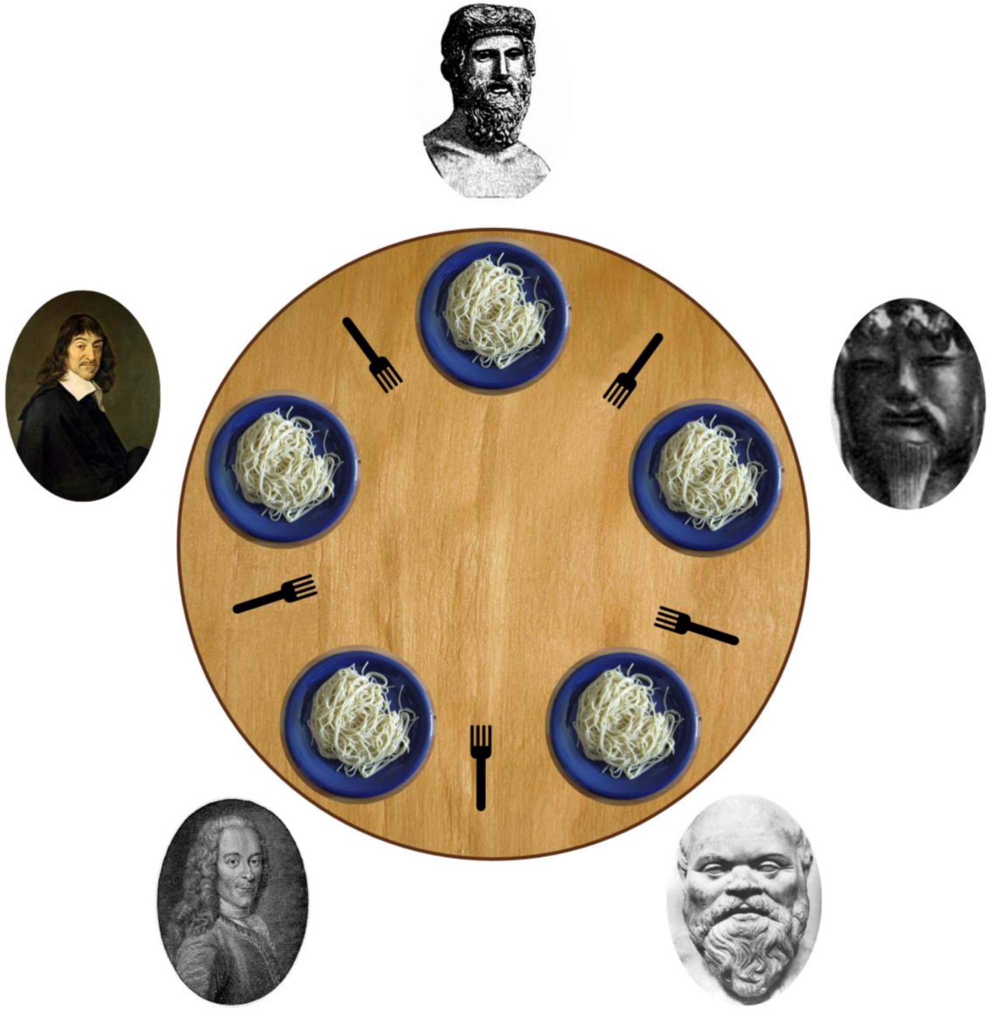
The dining philosophers problem

5 philosophers, 5 bowls of noodles &
5 chopsticks

Each philosopher must alternatively think and eat (forever).

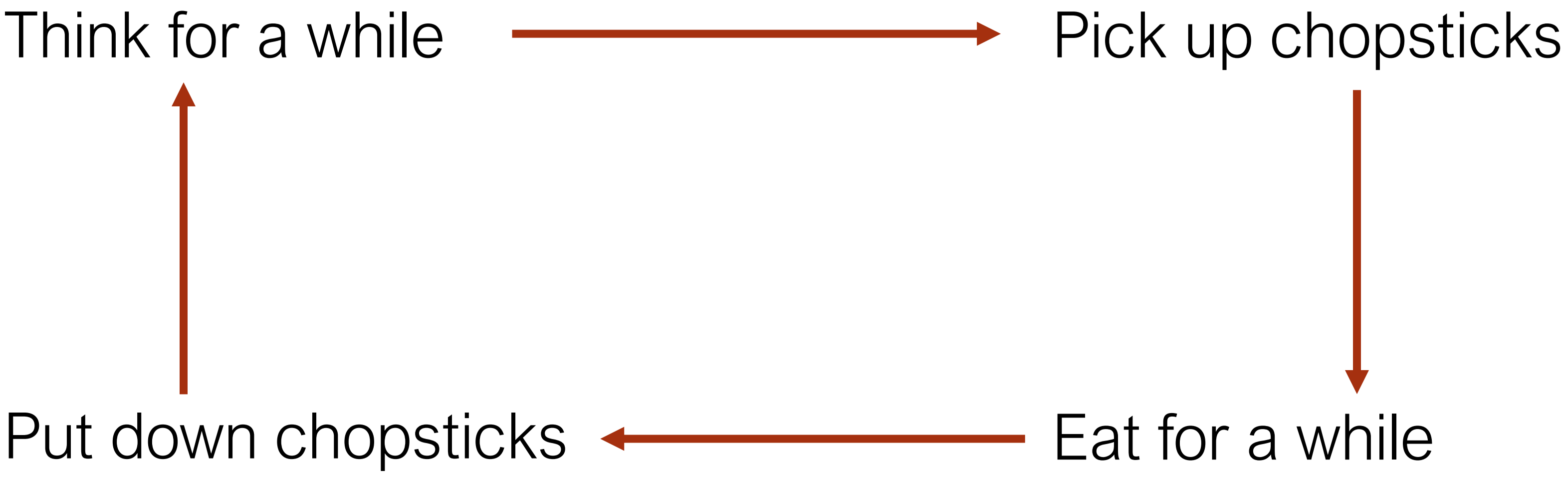
- can take a left or right chopstick if available
- BUT can only eat if BOTH left and right chopsticks are available

Goal: ensure no-one starves!



The dining philosophers problem

Philosopher behavior:



Pseudo-code for a philosopher

```
while (true) {
  think();
  pick_up_left_chopstick();
  pick_up_right_chopstick();
  eat();
  put_down_right_chopstick();
  put_down_left_chopstick();
}
```

Deadlock

Run:
Code_from_Lectures > Lecture9 > deadlockedphilosophers\Table.java

What’s gone wrong?

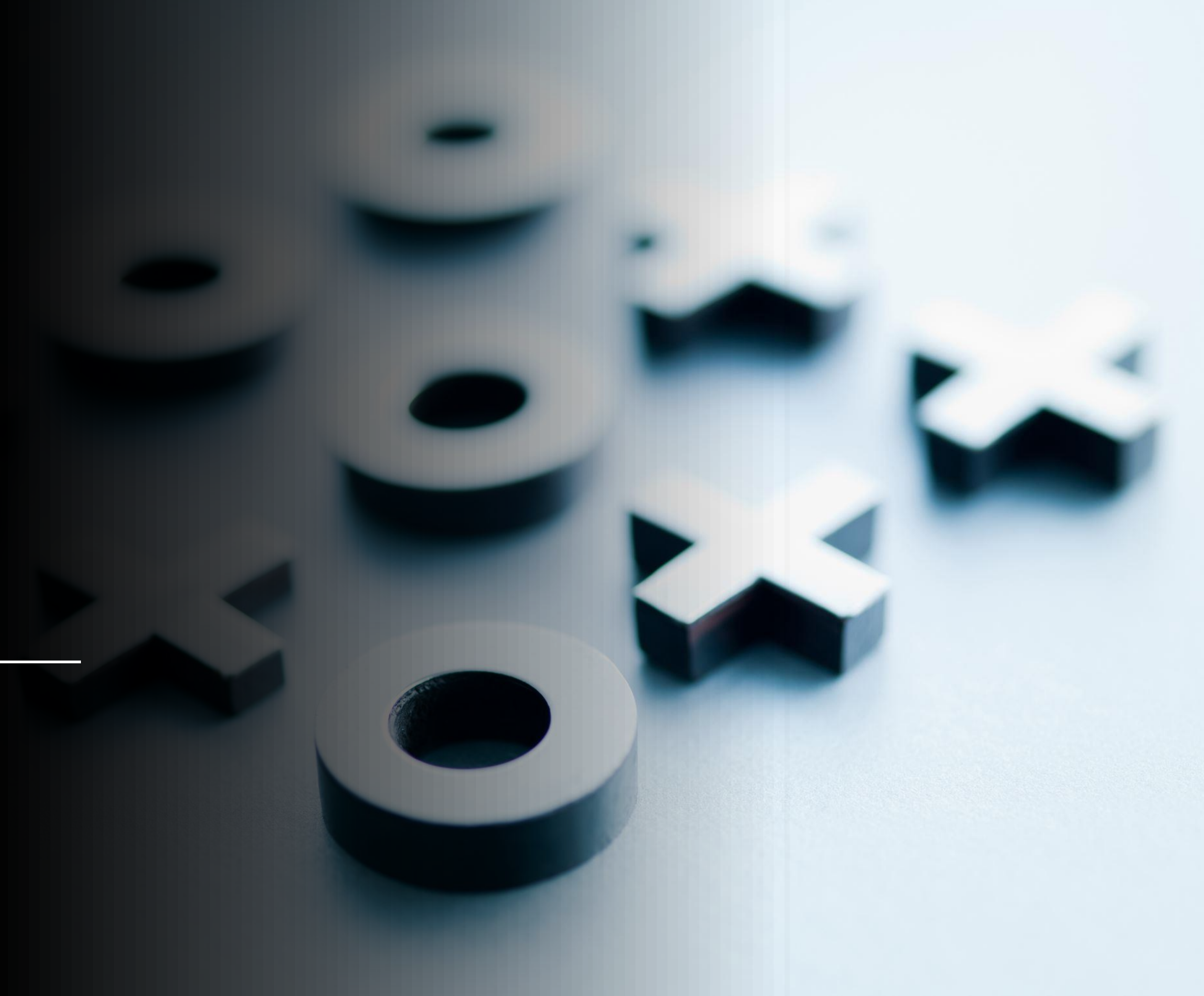
Deadlock

2 threads sharing access to 2 shared variables via locks:

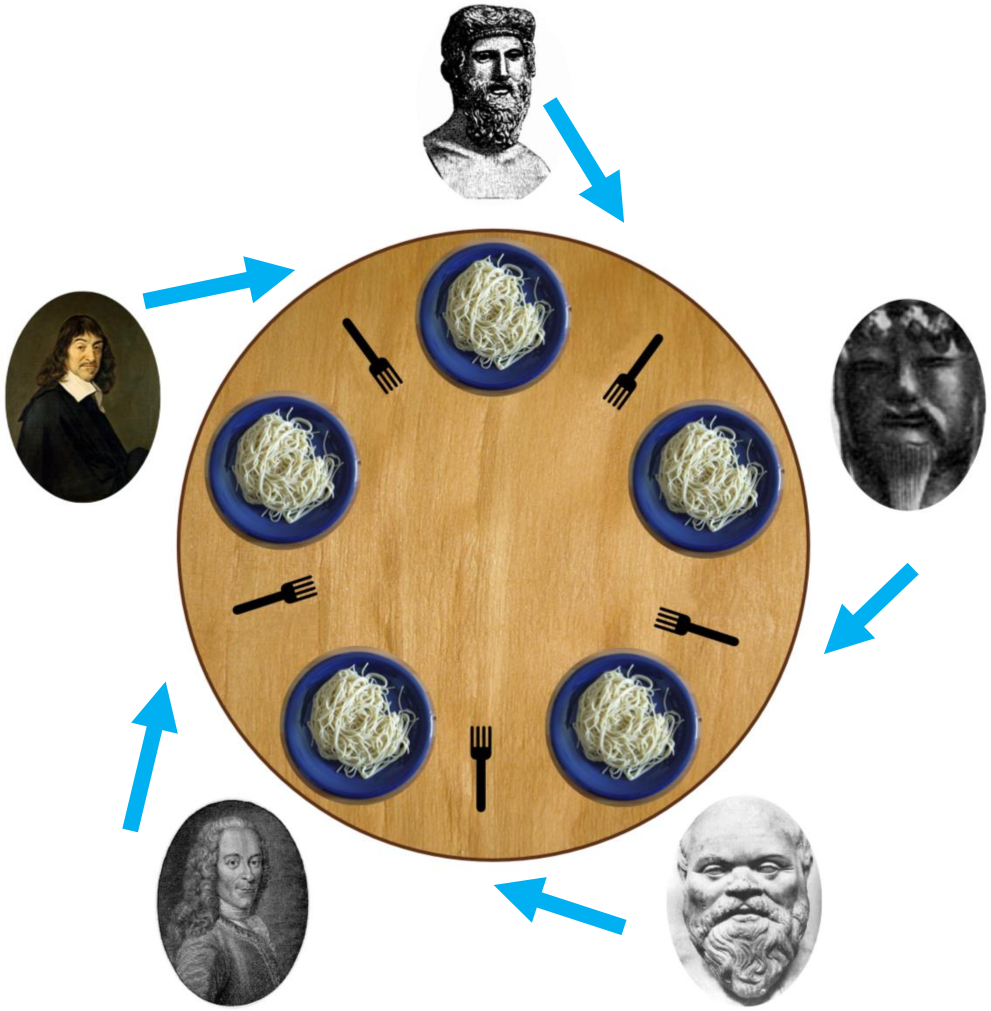
1. Thread 1: takes lock a
2. Thread 2: takes lock b
3. Thread 1: blocked from accessing b
4. Thread 2: blocked from accessing a

Let’s take a look

Deadlocked philosophers



Deadlocked philosophers



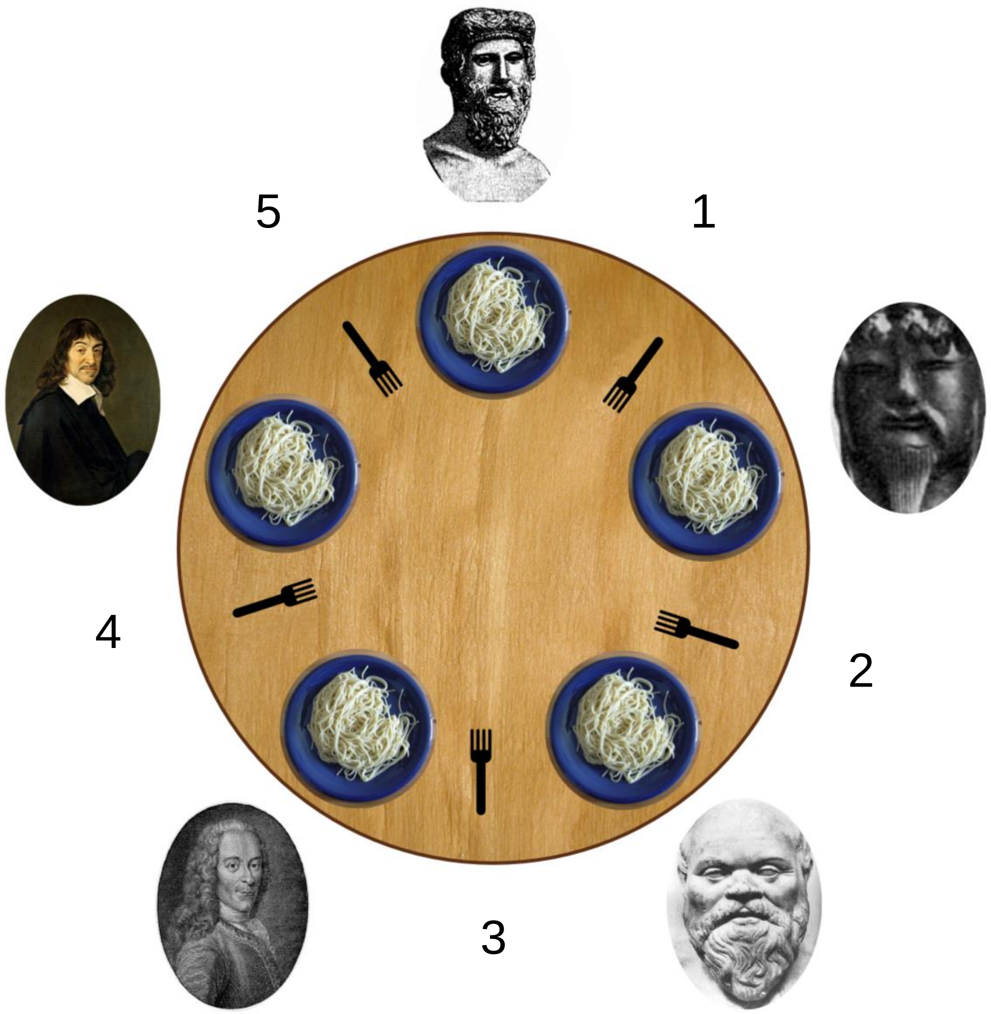
This is why concurrency is hard!

- Too few ordering constraints → race conditions
- Too many ordering constraints → deadlocks
- Hard/impossible to reason about based on modularity
  - Need to think about what all threads could do in all possible orders
- Thorough testing is impossible
  - Infinite number of possible interleavings
  - Controlled by system scheduler and events, not the program

One solution

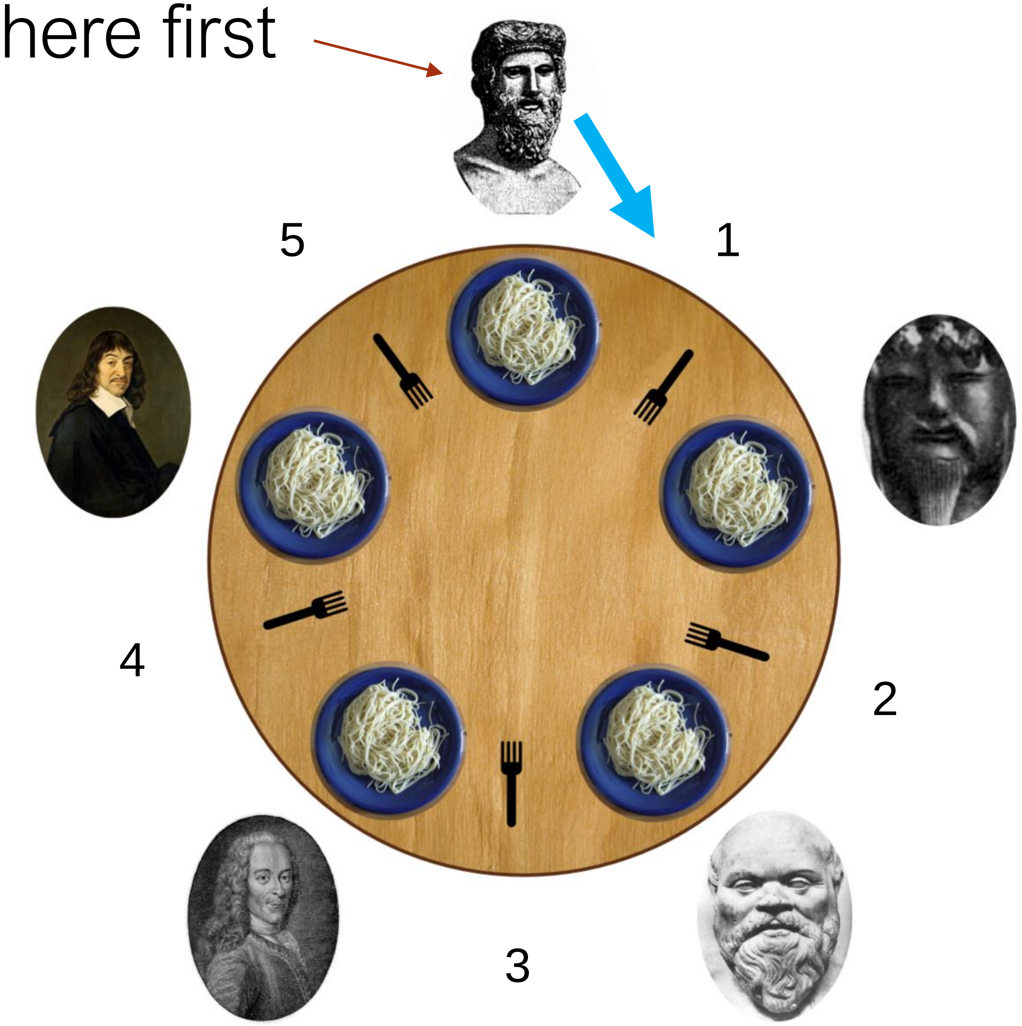
Number the chopsticks

Each philosopher picks up their lowest-numbered chopstick first

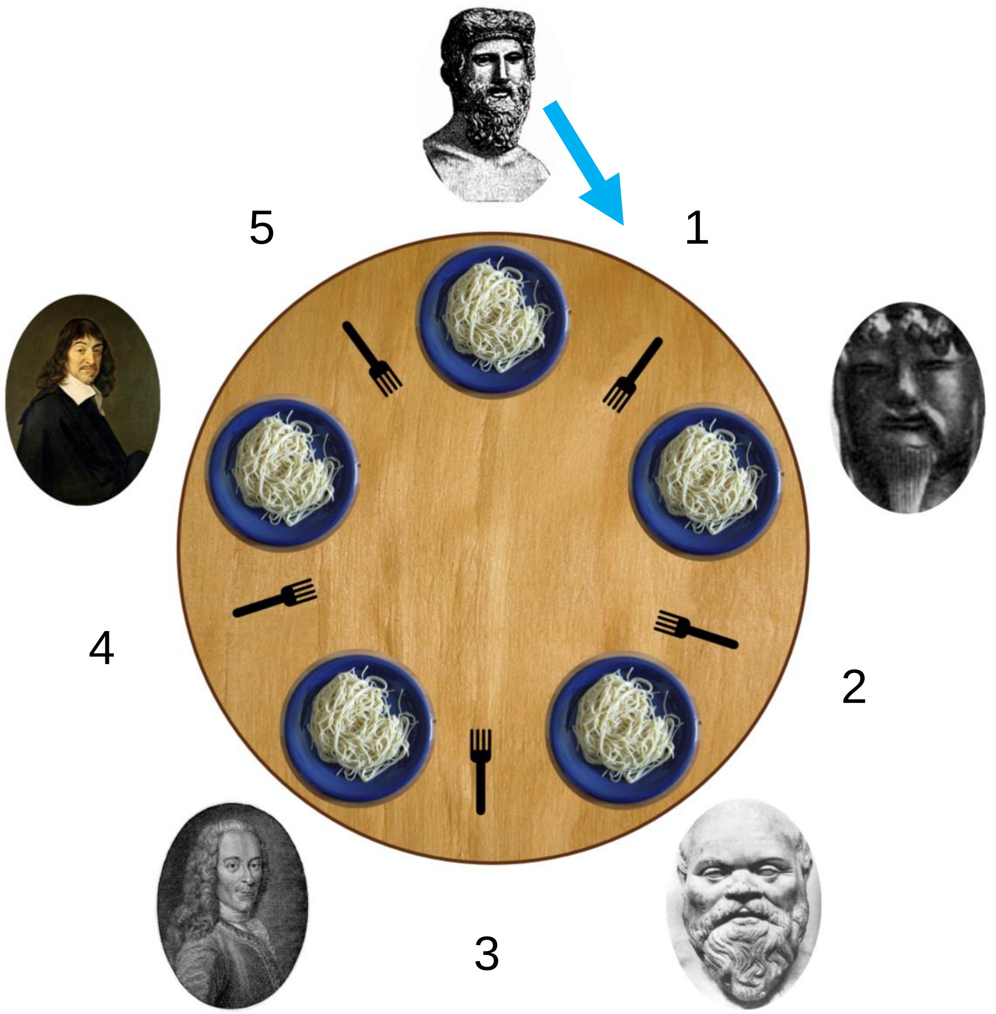


One solution: numbered chopsticks

Gets there first



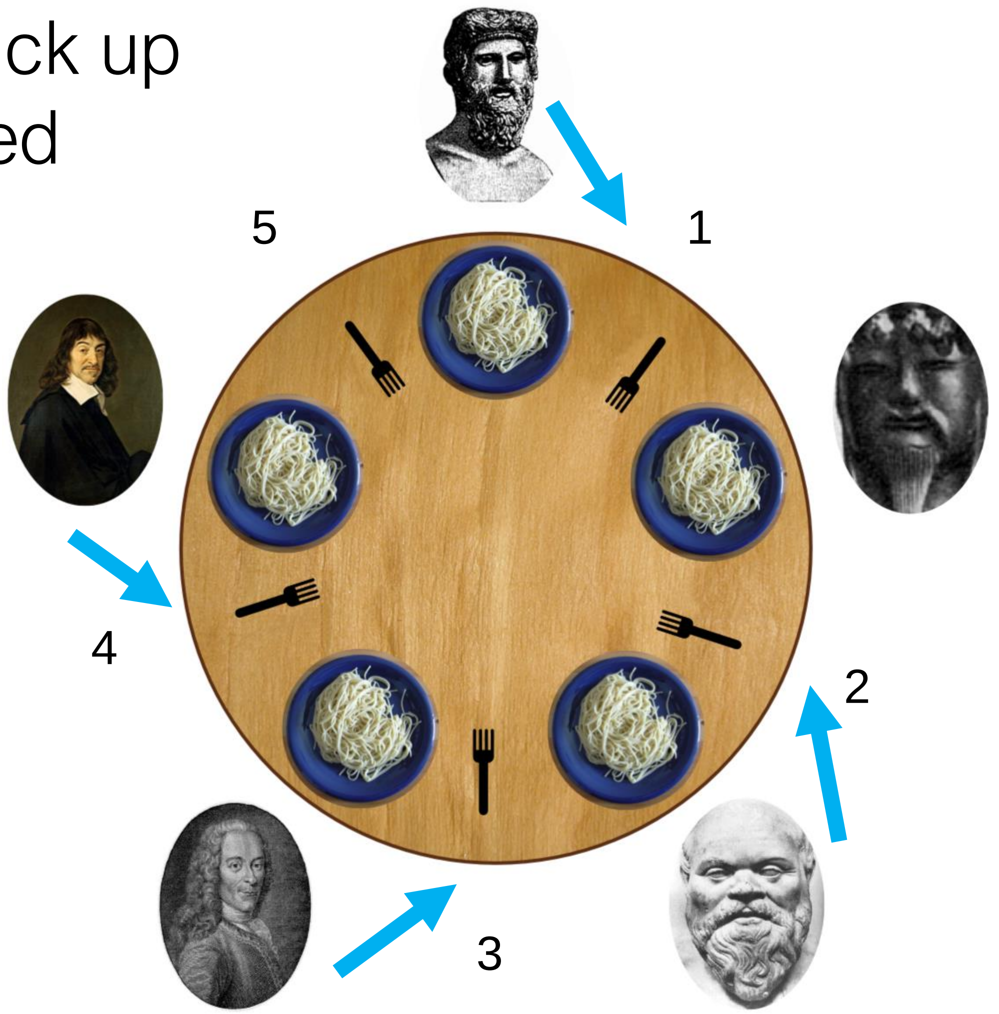
One solution: numbered chopsticks



Lowest-numbered chopstick is taken → waits

One solution: numbered chopsticks

Everyone else can pick up their lowest-numbered chopstick...

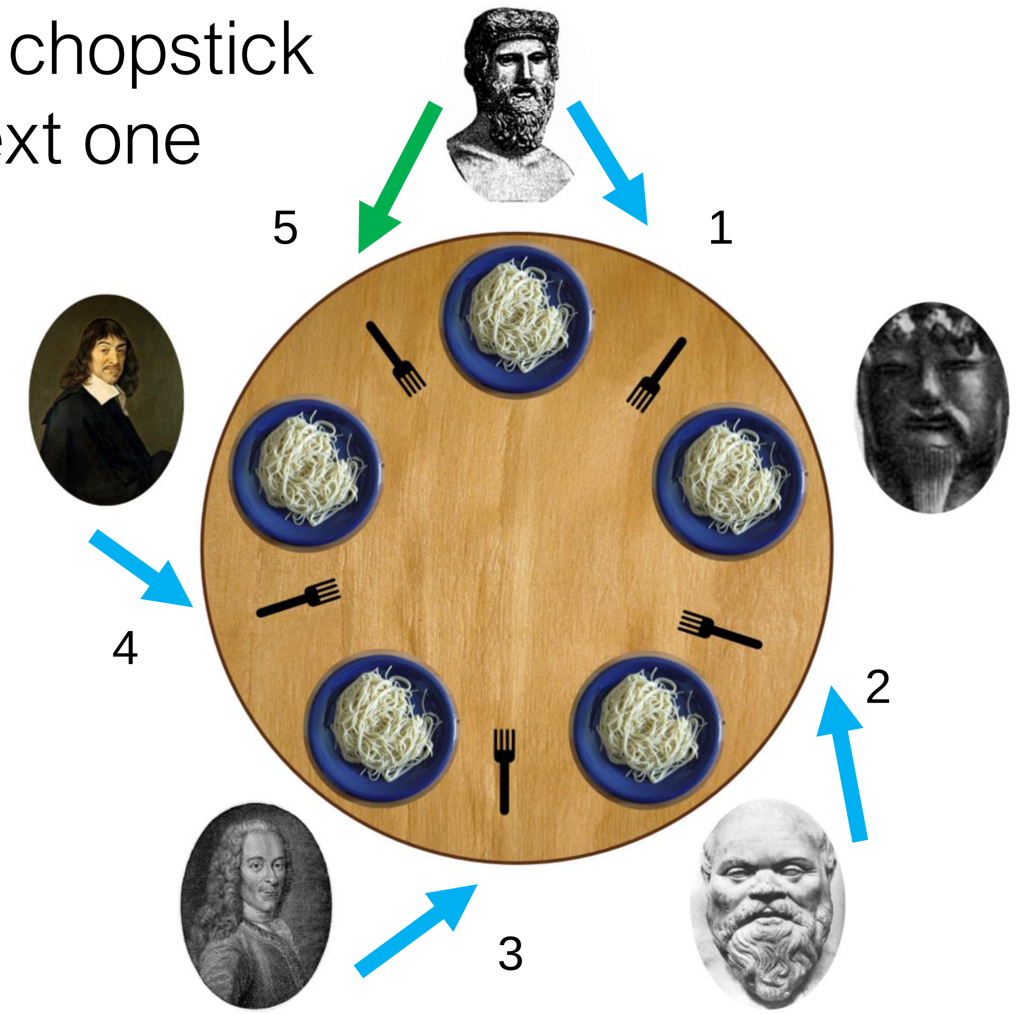


Lowest-numbered chopstick is taken → waits

One solution: numbered chopsticks

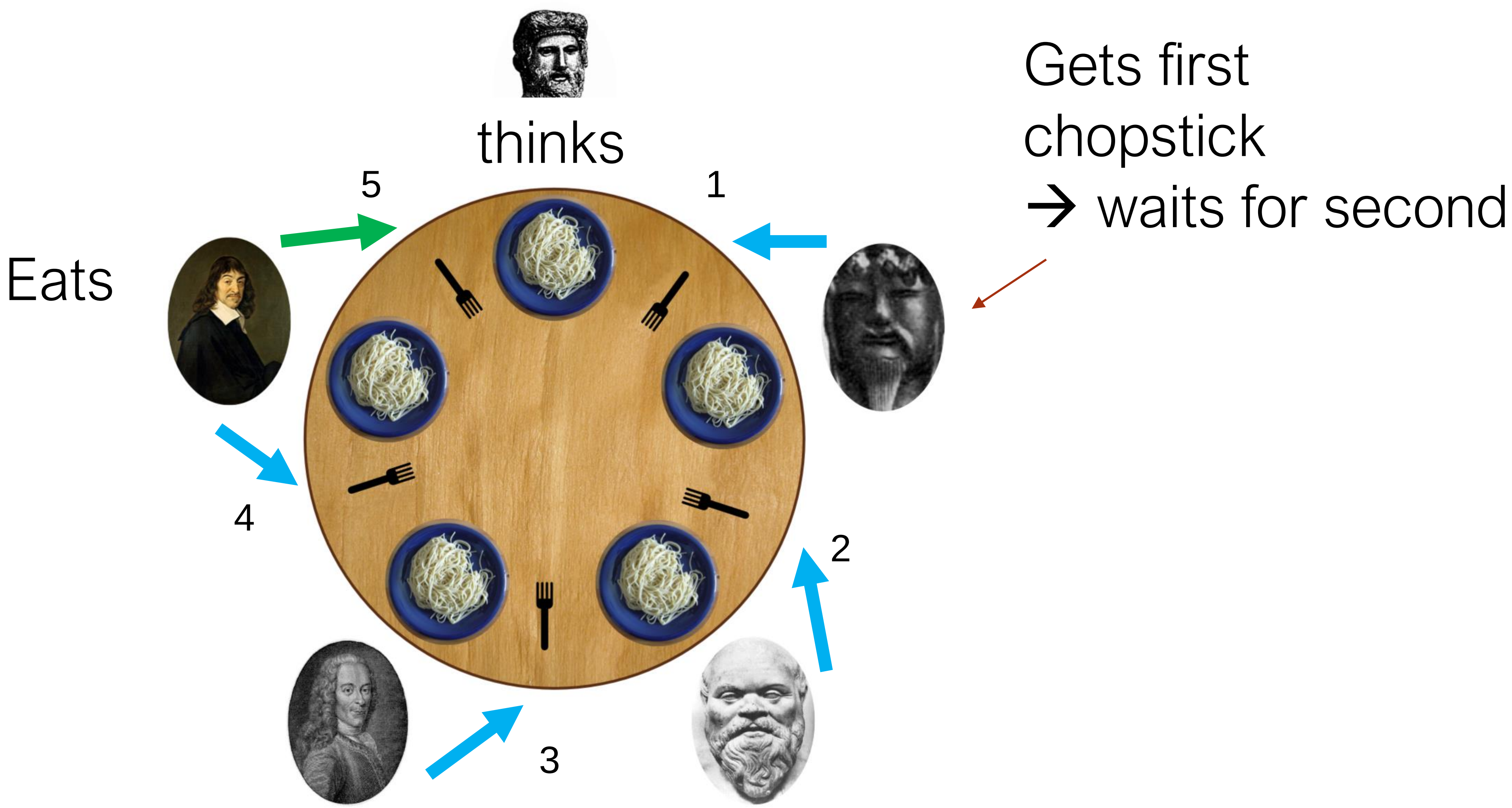
Philosophers with a chopstick try to pick up the next one

- All but one must wait



Lowest-numbered chopstick is taken → waits

One solution: numbered chopsticks

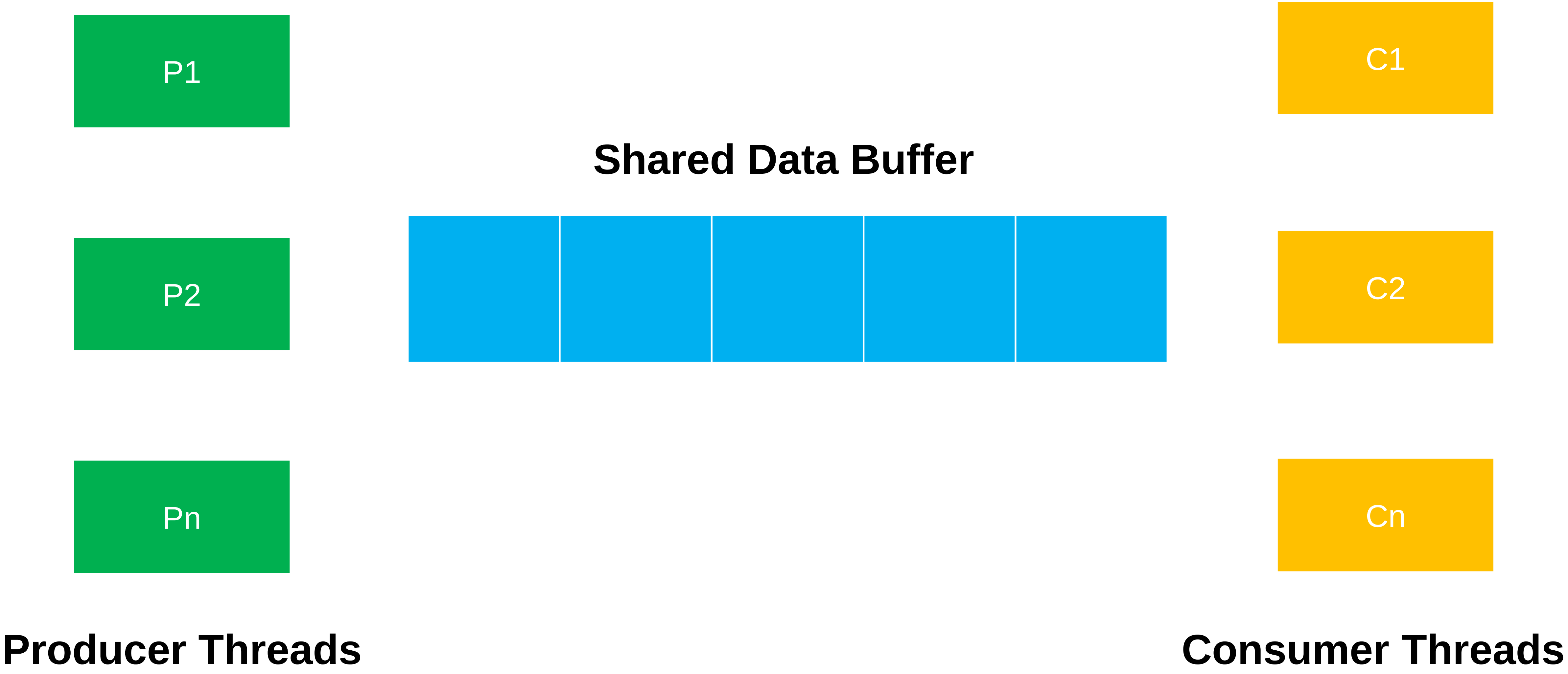


Let’s take another look

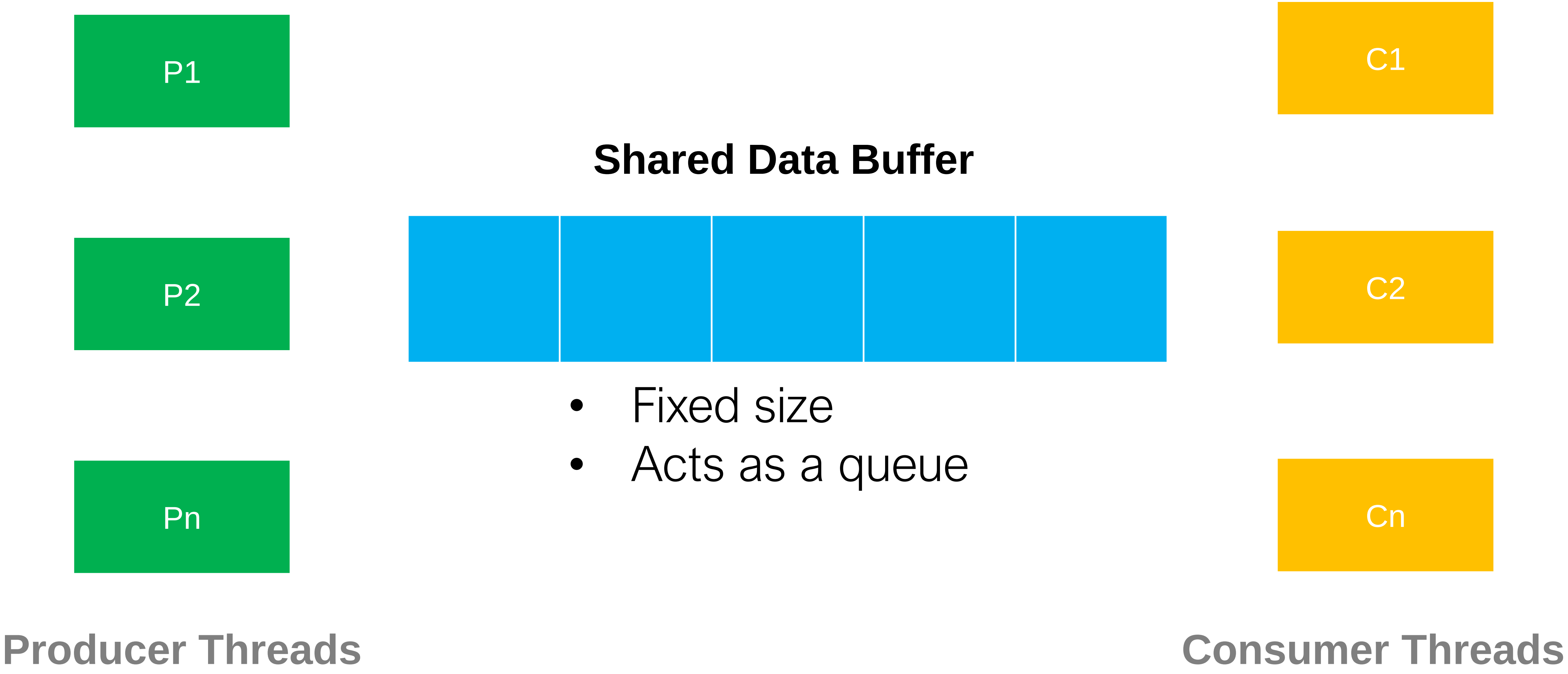
fed philosophers

Producer Consumer Problem

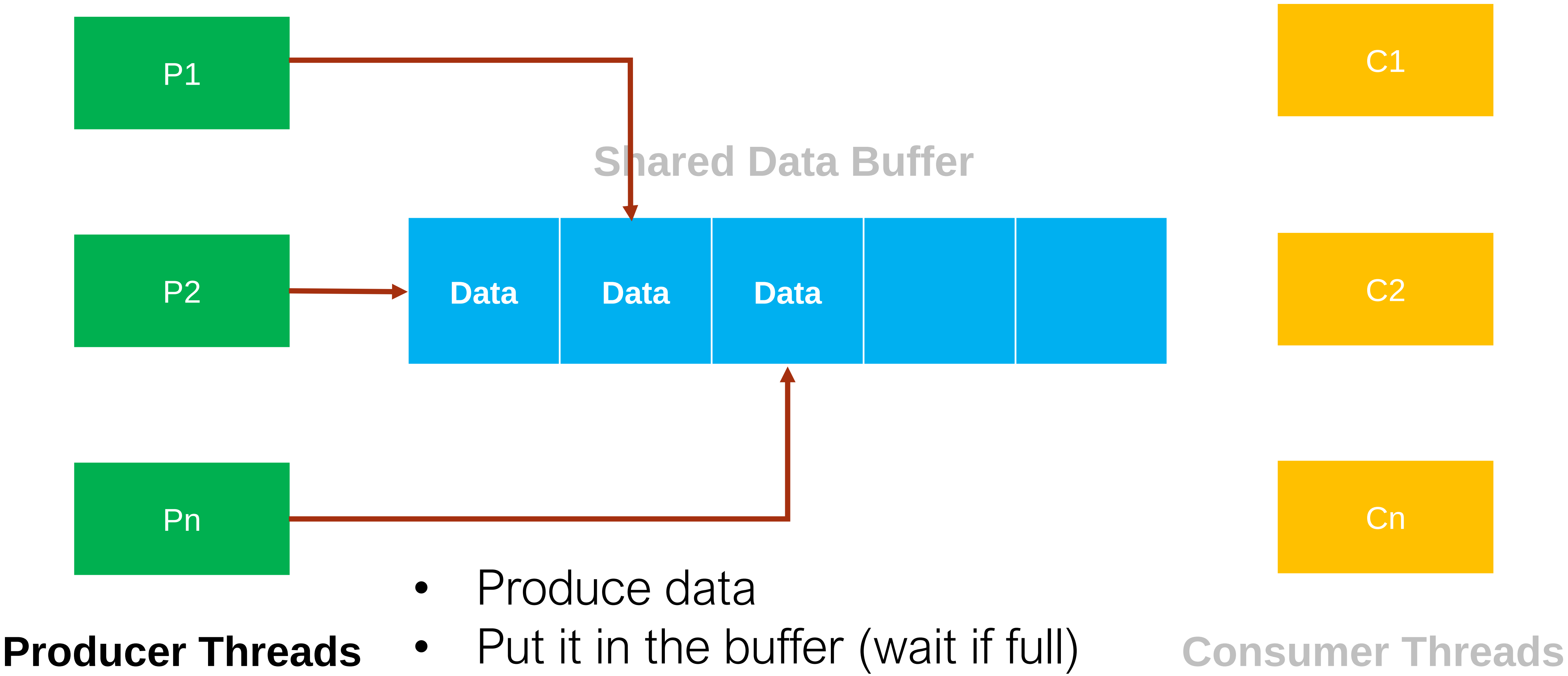
Producers and consumers



Producers and consumers

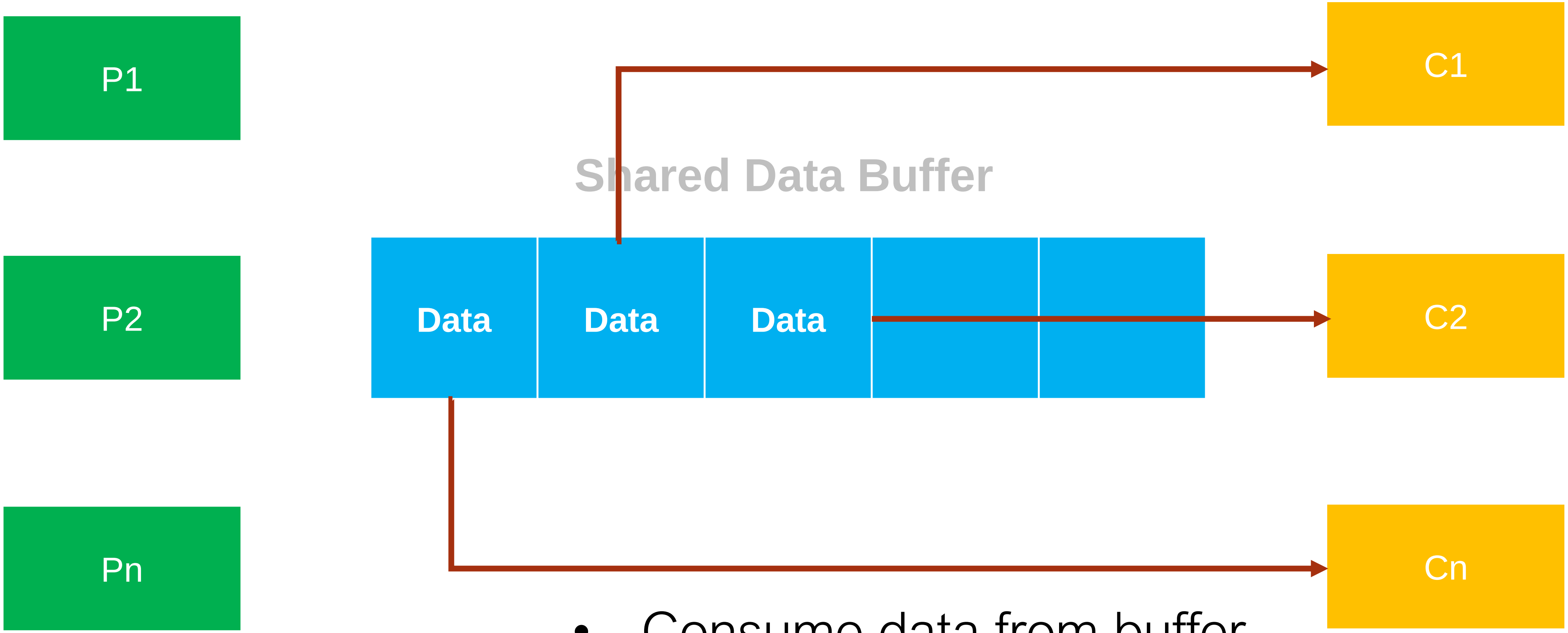


Producers and consumers



- Produce data
- Put it in the buffer (wait if full)
- Repeat

Producers and consumers



- Consume data from buffer
- Wait if empty
- Repeat

Producer Threads

Consumer Threads

Use case: event logging

Capture mouse clicks

Capture password fails

Capture exceptions

Producer Threads

Shared Data Buffer

Key: producer / consumer don’t care what the other does with the data.

Log everything

Log interactions

Log problems

Consumer Threads

Shared buffer implementation

- Some kind of data storage e.g. an integer, a list etc.
- Method to allow producer to add data
- Method to allow consumer to remove data

- Producer and consumer objects need reference to the shared buffer

Producer Consumer: first try pseudocode

```
producer() {
  while(true) {
    item = produceItem();
    if (bufferFull()) {
      sleep();
    }
    putIntoBuffer(item)
    if (!bufferEmpty()) {
      wakeup(consumer);
    }
  }
}
```

```
consumer() {
  while(true) {

    if (bufferEmpty()) {
      sleep();
    }
    consumeItem()
    if (!bufferFull()) {
      wakeup(producer);
    }
  }
}
```

Potential for deadlock...

- Consumer notices empty buffer, moves inside the if block

Potential for deadlock...

- Consumer notices empty buffer, moves inside the if block
- Consumer interrupted (context switch) <u>before</u> calling sleep

Potential for deadlock...

- Consumer notices empty buffer, moves inside the if block
- Consumer interrupted (context switch) <u>before</u> calling sleep
- Producer creates an item, puts it in the buffer

Potential for deadlock...

- Consumer notices empty buffer, moves inside the if block
- Consumer interrupted (context switch) <u>before</u> calling sleep

- Producer creates an item, puts it in the buffer
- Because the buffer was empty, producer tries to wake up consumer

Potential for deadlock...

- Consumer notices empty buffer, moves inside the if block
- Consumer interrupted (context switch) <u>before</u> calling sleep
- Producer creates an item, puts it in the buffer
- Because the buffer was empty, producer tries to wake up consumer
- But...consumer wasn't sleeping, wake up call is lost

Potential for deadlock...

- Consumer notices empty buffer, moves inside the if block
- Consumer interrupted (context switch) <u>before</u> calling sleep

- Producer creates an item, puts it in the buffer
- Because the buffer was empty, producer tries to wake up consumer

- But...consumer wasn't sleeping, wake up call is lost
- Consumer goes to sleep

Potential for deadlock...

- Consumer notices empty buffer, moves inside the if block
- Consumer interrupted (context switch) <u>before</u> calling sleep
- Producer creates an item, puts it in the buffer
- Because the buffer was empty, producer tries to wake up consumer
- But...consumer wasn't sleeping, wake up call is lost
- Consumer goes to sleep
- Producer loops until the buffer fills, then goes to sleep

Guards

Producer-Consumer style implementations require "guards" on the buffer

- Buffer accepts incoming data unless full → waits until room
- Buffer allows data to leave unless empty → waits until there is data

Java guards (AKA monitors)

wait() and notify() statements → communicate to threads synchronized on the same object

E.g.:

- Thread A calls synchronized method of buffer
- Buffer isn't ready → wait() → thread A waits
- Thread B changes buffer using different method
- Buffer is ready to complete thread A's request → notifyAll() → all threads waiting on method continue

Example: Code_from_Lectures > Lecture9 > producercomsumerguarded

Let’s take a look

Producer consumer guarded

Takeaways: the shared buffer

(This example shows one approach of several)

Buffer implemented in its own class:

- Has a data structure for storing things sent from/to threads
- Data structure has a size limit
- Methods for putting data and getting data are `synchronized`
- General approach to putting and getting:
  - if buffer is full/empty → `wait()`
  - do job
  - `notifyAll()`

Takeaways: producer and consumer

Producer and consumer are both threads (implementing Runnable or subclassing Threads)

- Each has a reference to the same buffer instance
  - e.g. if a custom class: pass it via the constructor
- The `run()` method starts the process of putting/getting
  - All checking/waiting handled by the buffer...no need to do it here

Thread states

Thread states

- New thread
  - Created but not started
- Runnable state
  - Started and running or ready to run
- Blocked/waiting state
  - Temporarily inactive due to another thread
- Dead state
  - run() terminates

Blocked/waiting state

A thread is not runnable if one of the following occurs:

- sleep() is invoked
- suspend() is invoked
- wait() is invoked
  - waits for notification of a free resource
  - waits for completion of another thread
  - waits to acquire a lock on an object
- blocked on an I/O request

Thread resumption

- If a thread is asleep:
  - Sleep period elapses or it is interrupted
- If a thread is suspended:
  - Its resume() method must be called
- If a thread is waiting:
  - The object owning the shared resource must relinquish it by calling either notify() or notifyAll()
- If waiting on I/O, I/O must complete

Thread priority

- Every Java thread has a priority
  - High priority threads get scheduled more frequently than lower priority threads
- Java threads inherit priority from parent...
  - MIN_PRIORITY (1)
  - NORM_PRIORITY (5) Default
  - MAX_PRIORITY (10)
- ...unless set using `thread.setPriority(priority)`

Thread scheduling

- Scheduler chooses the runnable thread with the highest priority
- When there are multiple threads to choose from → picks one. The chosen thread runs until:
  - a higher priority thread becomes runnable
  - it yields or completes
  - its time allotment has expired

Reentrancy

Every Java object has a lock associated with it

- Known as the intrinsic lock
- AKA *monitor* or *mutex* locks

Synchronized methods exploit this intrinsic lock

- Lock acquired by executing thread before entering a synchronized block
- Lock released automatically when the thread exits the synchronized block

Reentrancy

Intrinsic locks are **reentrant***:*

- If a thread tries to acquire a lock it already holds, it succeeds

Reentrancy facilitates encapsulation of locking behavior and simplifies OO concurrent code

Thread safety

- Thread safety requires the internal state of an object to be protected from concurrent updates
  - i.e. can’t be changed by more than one thing at a time
- What if an object has no state that persists...?
- Or cannot be modified by a calling thread?
- Is this thread-safe?

Thread safety

Stateless and immutable objects are always thread-safe

Thread safe collections

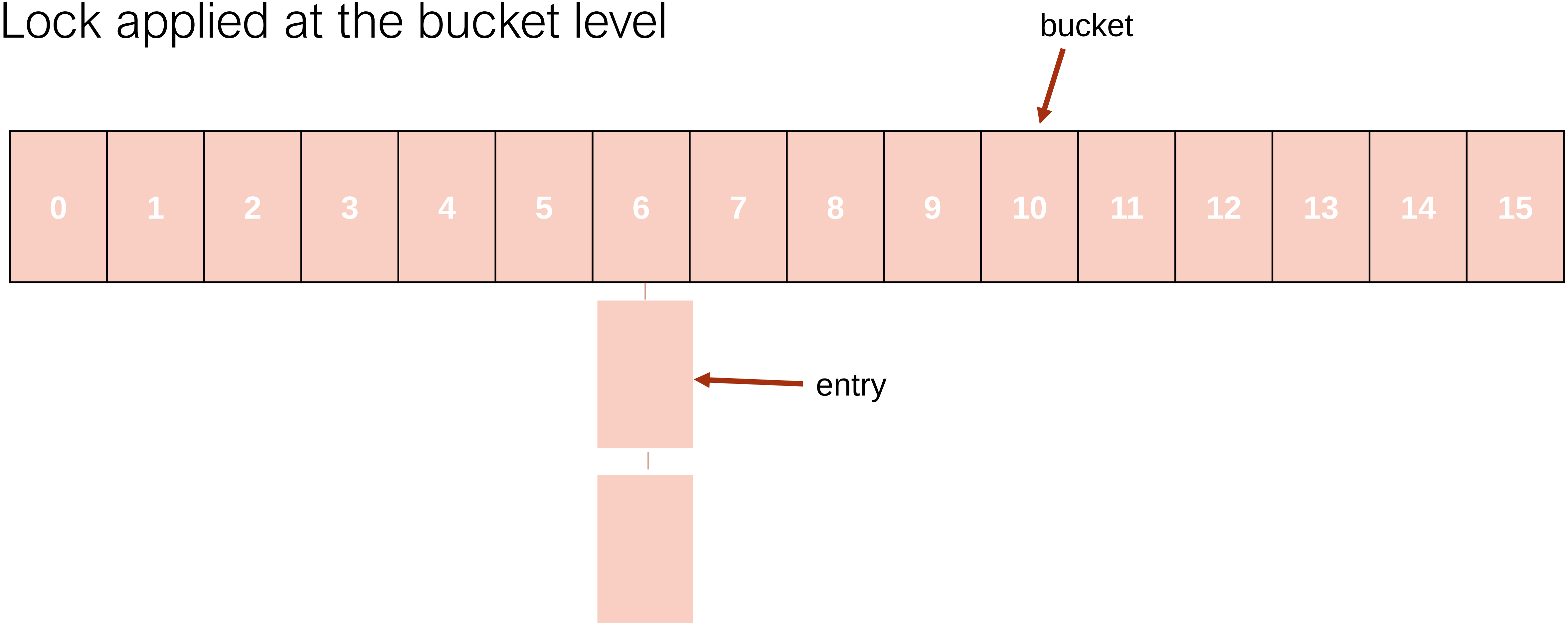
Thread-safe collection classes

- Standard collection class are NOT thread-safe
  - i.e. multiple threads can modify a standard collection at the same time
- `java.util.concurrent` package provides thread-safe collections, including:
  - **BlockingQueue**: queue that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue
  - **ConcurrentMap**: thread-safe subinterface of `java.util.Map`
  - **ConcurrentHashMap**: thread-safe version of `HashMap`

ConcurrentHashMap

A HashMap is divided into buckets

Lock applied at the bucket level



BlockingQueue

Saves the buffer from having to wait() and notifyAll()

Blocking Queue

Produce data
Put it in the queue (wait if full)
Repeat

Wait if empty
Retrieve from queue
Repeat

Using a BlockingQueue

All producer/consumer threads that will interact with the shared queue need a reference to it. E.g.:

```
public class Producer implements Runnable {
  private BlockingQueue<Integer> buffer;
  public Producer(BlockingQueue<Integer> buffer) {
    this.buffer = buffer;
  }
  ... produce things...
}
```

Using a BlockingQueue

All producer/consumer threads that will interact with the shared queue need a reference to it. E.g.:

```
public class Consumer implements Runnable {
  private BlockingQueue<Integer> buffer;
  public Consumer(BlockingQueue<Integer> buffer) {
    this.buffer = buffer;
  }
  ... consume things...
}
```

Using a BlockingQueue

Producers and consumers of the same BlockingQueue need a reference to the *same* instance...e.g.:

```
public class Driver {
  public static void main(String[] args) {
    ArrayBlockingQueue<Integer> shared_buffer
                  = new ArrayBlockingQueue<Integer>(10);
    Thread p = new Thread(new Producer(shared_buffer));
    Thread c = new Thread(new Consumer(shared_buffer));
  }
}
```

Takeaways: BlockingQueue example

(Again, one of many possible configurations)

- No need for a separate buffer class
- No need for:
  - synchronized put/get methods
  - wait()
  - notifyAll()

→ BlockingQueue handles everything

Let’s take a look

Producer consumer
blockingqueue

Recap

- Concurrency is fundamental to software systems
- Introduces problems of race conditions and deadlocks
- Synchronization required as a solution
- Threads move through various states during the lifetime
- Scheduler makes decisions on which thread to run based on state and priority

Questions?

Implementation exercise

Producer-consumer: file I/O

Use a producer-consumer approach to read a csv file and print it to the terminal. Try at least one of the following:

- Using a guarded buffer (`synchronized`, `wait()`, and `notifyAll()`)
- Using a thread-safe collection e.g. BlockingQueue