Concurrency

Concurrency

Concurrency in Real Life

- Concurrency is the simultaneous occurrence of events.

- Most complex tasks that occur in the physical world can be broken down into a set of simpler activities
  - Building a house: bricklaying, carpentry, plumbing, electrical installation, roofing
  - Some of them can overlap and take place concurrently

Concurrency in Computing

- Computing on the Internet: independent, autonomous agents trying to achieve individual and shared goals.

- Even on our local machines, we take it for granted that we can do more than one thing at a time.
  - We continue to work in a word processor, while other applications download files, manage the print queue, and stream audio.
  - Even a single application is often expected to do more than one thing at a time.

Concurrent Programming

- The activity described by a computer program can also be divided into simpler activities (subprograms)

  - **Sequential programs:** Subprograms do not overlap in time, they are executed one after another
    - In 15-110 we have been writing sequential programs.

  - **Concurrent programs**: Subprograms may overlap in time, their executions proceed concurrently
    - In 15-110 we will not write concurrent programs but we will learn about what makes them tricky.

Why Do We Need It?

- Everything happens at once in the world. Inevitably, computers must deal with that world.
  - For example, traffic control, airline seat reservation, process control, banking
- Performance gain from multiprocessing hardware
  - For example, Google, Yahoo, divide each query into thousands of little queries and use thousands of small computers.
  - For example, a supercomputer with thousands of processors can compute a weather prediction much faster than a single processor.
- Increased application throughput for applications sharing computational resources. Throughput is the amount of work that a computer can do in a given time period.
  - When one application is waiting for I/O another can continue its execution.

Caution

- The advantages of concurrency may be offset by the increased complexity of concurrent programs.
  - We will be giving some examples of what may go wrong in concurrent programming.

- Notorious cases of erroneous concurrent software:
  - Therac-25 computerized radiation therapy machine
  - Mars Rover “Spirit”

DIFFERENT FLAVORS OF CONCURRENCY

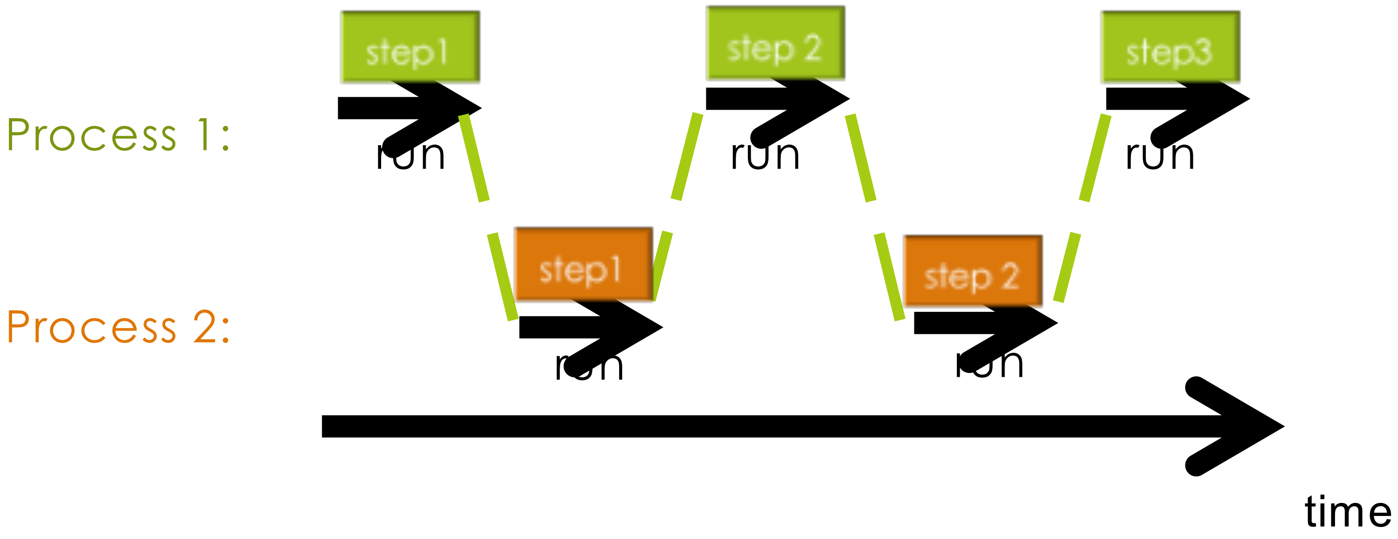
A Useful Abstraction: Process

- Process: A program in execution
  - Program along with its data in memory, open files, open communication channels etc.

- Concurrency involves multiple processes running simultaneously on multiple processors or on a single processor time-sharing the processor.

Sharing a Processor

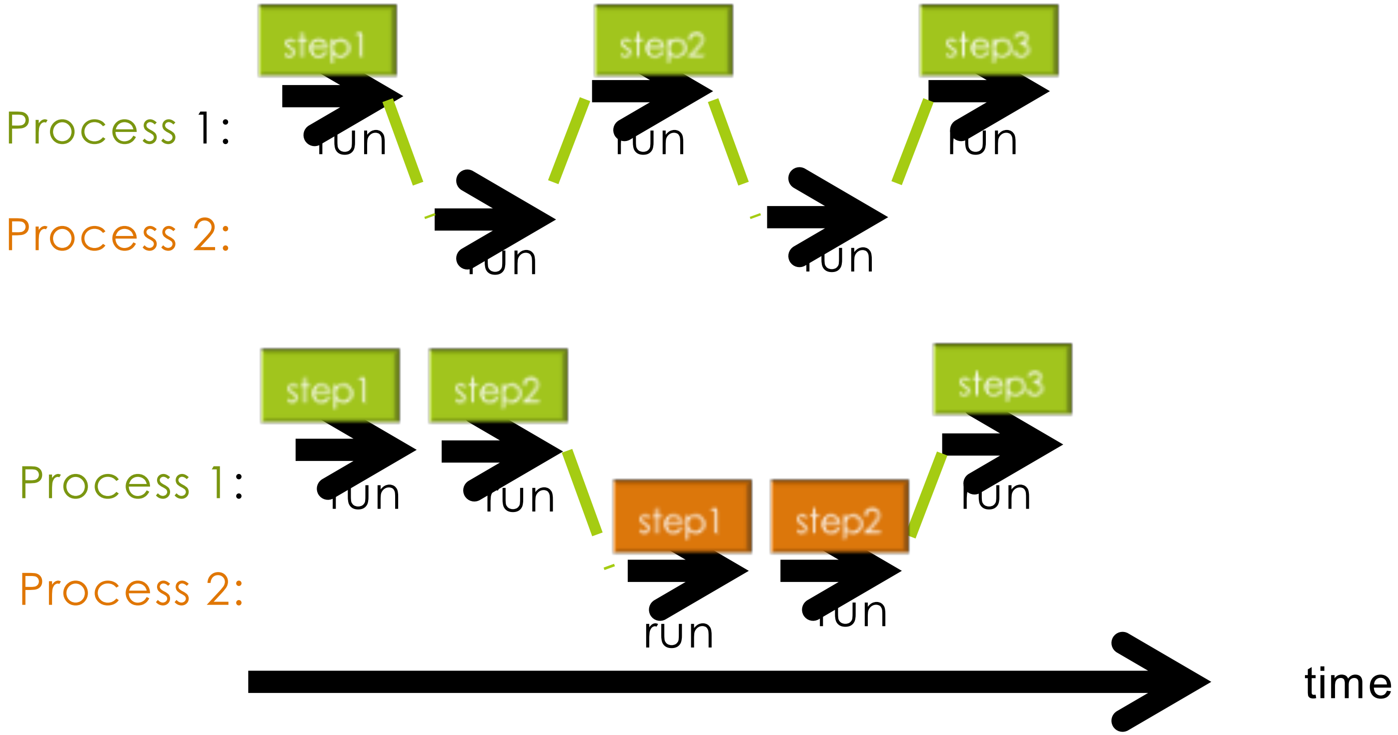
If only one processor (CPU) is available, the only way to run multiple processes is by switching between them.



Only one process is using the CPU at a given time even though they look like they are running in parallel to an observer.

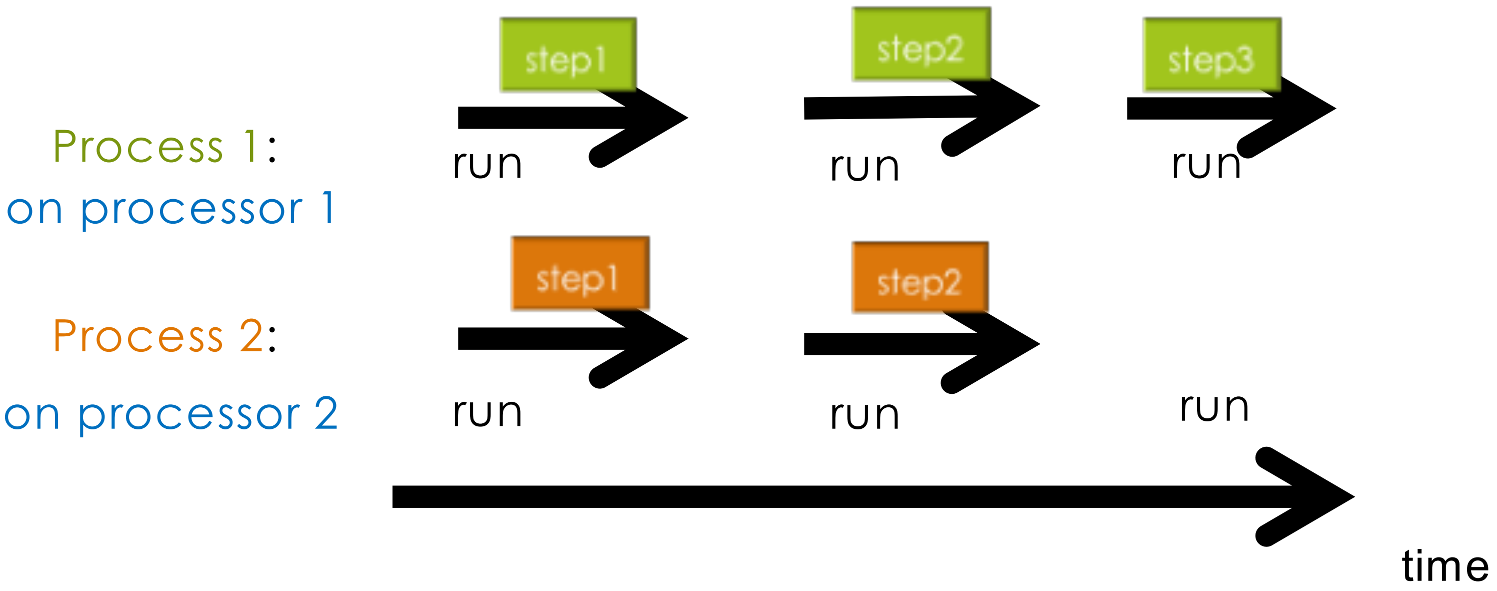
Scheduling

The order in which the steps are run is determined by a scheduler. There are many possibilities.



Multiple Processors

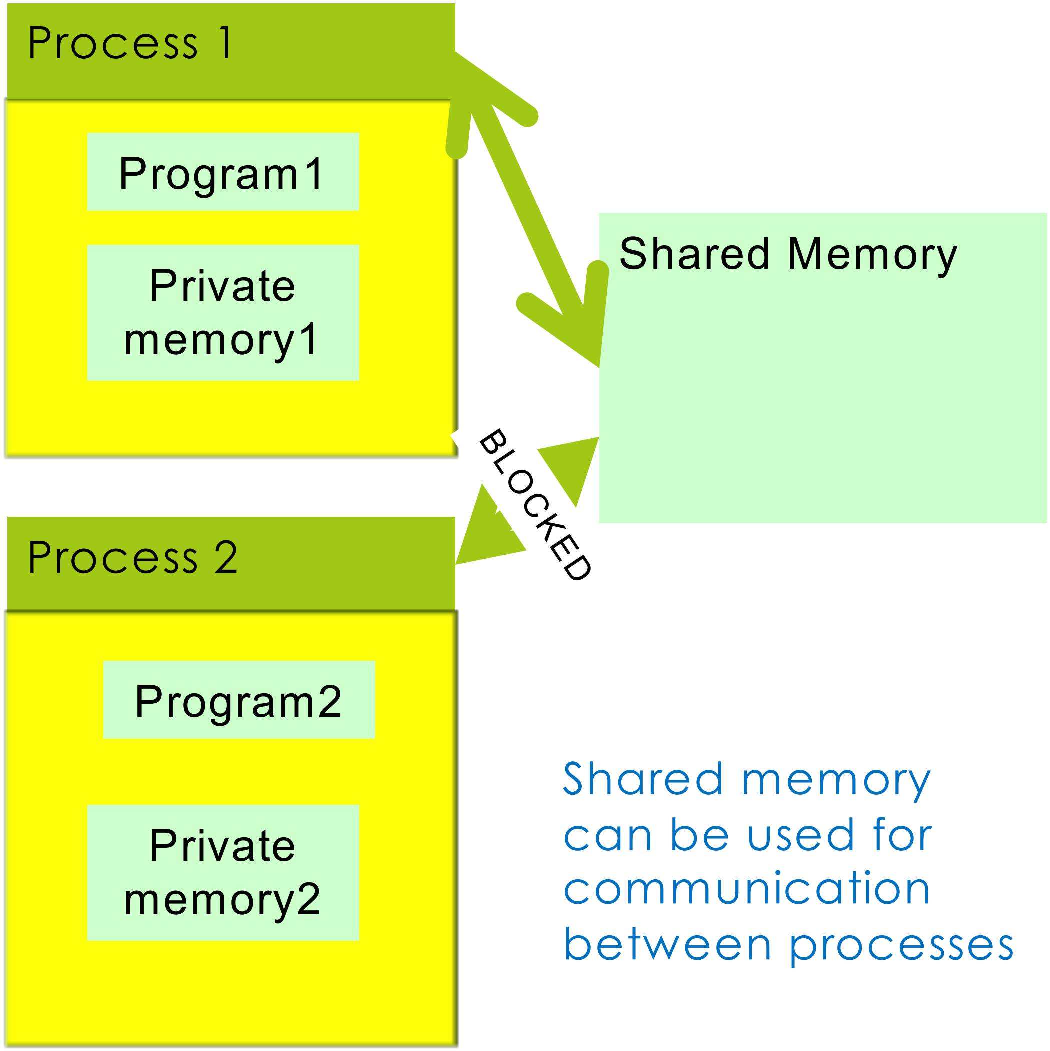
If you have multiple CPUs, you may execute multiple processes in parallel (simultaneously). Really!



Sharing Memory

- Processes may share resources such as memory

- For example, only one processor at a time may execute an instruction that touches the shared memory. The memory hardware makes the others wait.

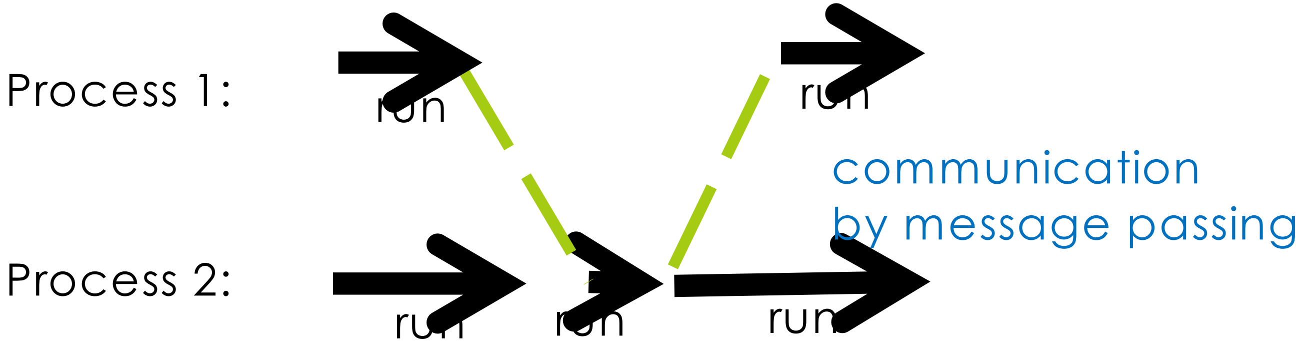


Shared memory can be used for communication between processes

Distributed Computing

Processes may run on distributed systems

- For example, a cluster of workstations, communicating via sockets



Process 1: run run

Process 2: run run run

communication
by message passing

Some steps are executed simultaneously but some are dependent on another

CONCURRENT PROGRAMMING

“Thinking Parallel”

- **Hardware supports parallelism.** Nowadays, we have multiple processors in most computing environments such as multicore machines, clusters.

- **Programmers do not always support parallelism.** Algorithms do not fully utilize parallelism provided by hardware.

- **Many programming languages offer “multithreading” libraries** to support concurrent programming:
  - Structuring programs where there are logically separate, naturally independent control flows.
  - What is really needed is development of new languages that will enable programmers to express parallel algorithm designs.

- We will not focus on parallel algorithms. We will focus on issues that arise from concurrent execution of sequential processes that cooperate to achieve a common goal.

Threads

- What most programmers think of when they hear about concurrent programming today.
- We will use Python threads to illustrate some challenges with concurrent programming.
- Thread: a (somewhat) independent computation running inside a program
- Shares resources with the main program (memory, files, network connections etc.)

Thread Basics

```
>>> python3 -i example.py
```

statement

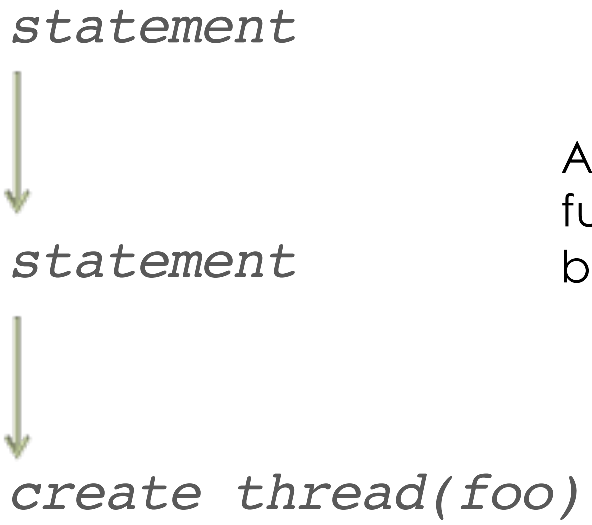
statement

statement

Python launches the “main” thread of the program. Control flows from one statement to another.

Thread Basics

```
>>>  python3 -i example.py
```

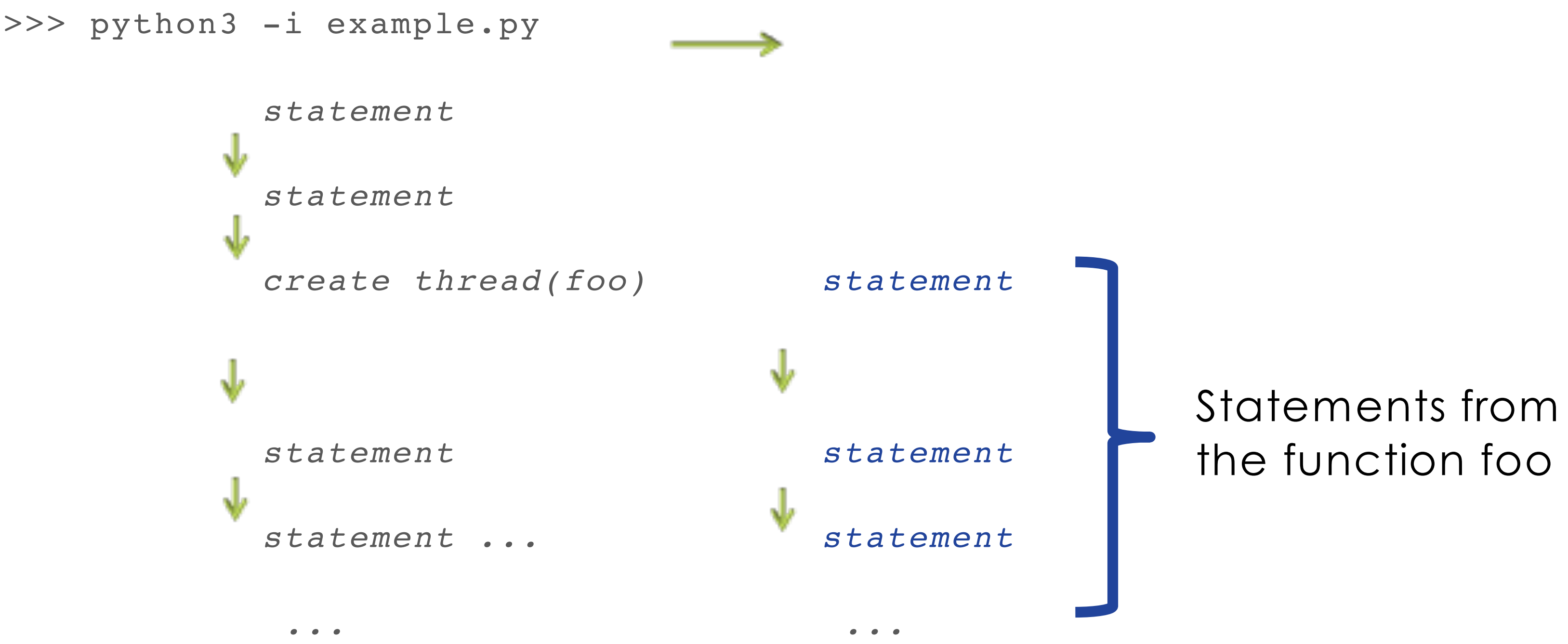


```
statement
    ↓
statement
    ↓
create thread(foo)
```

Assume that `foo` is a function that has already been defined.

```
def foo():
    statement
    statement
    ...
```

Thread Basics



```
>>> python3 -i example.py
```

statement

statement

create thread(foo)

statement

statement ...

...

statement

statement

statement

...

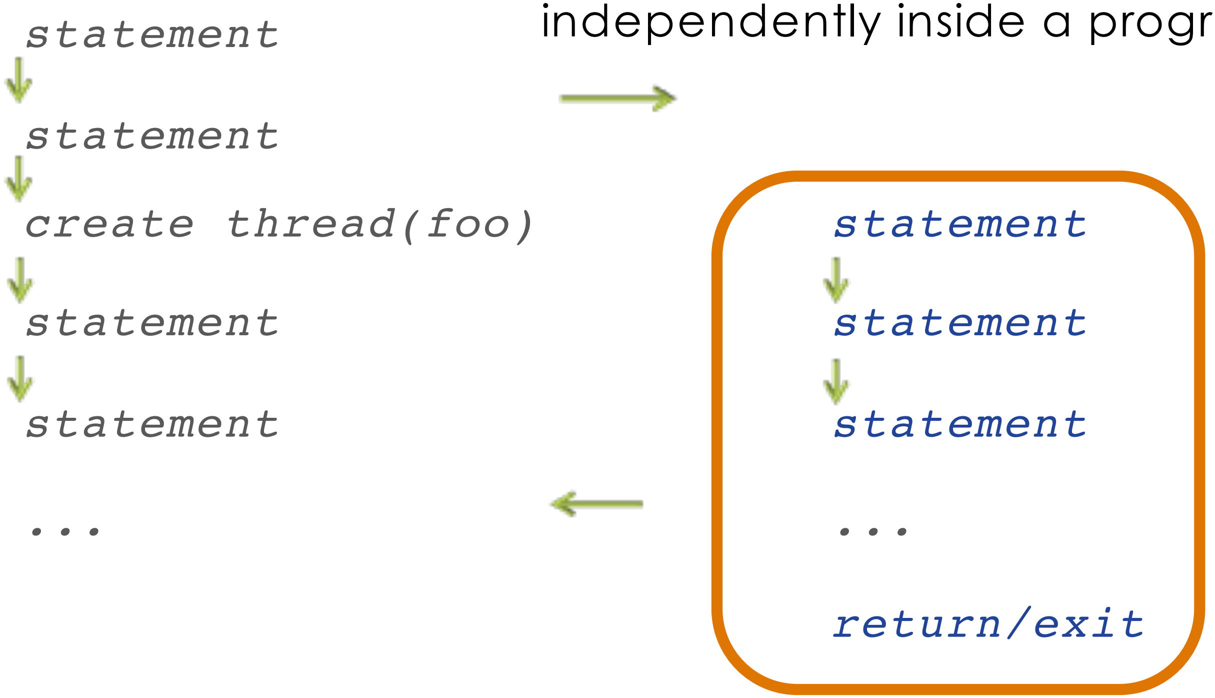
Statements from the function foo

Concurrent execution of the "main" thread **and** the function foo()

Thread Basics

```
>>> python3 -i example.py
```

Thread is like a “process” that runs independently inside a program



```
statement
statement
create thread(foo)
statement
statement
...
```

```
statement
statement
statement
...
return/exit
```

Functions as Threads

The Python module `threading` allows you to create Thread objects or use functions as threads.

Below is a function that is used as a thread.

```
import threading

def countdown(count):
    while count != 0:
        count = count-1
    return
t1= threading.Thread(target=countdown, args=(10,))
t1.start()
do your own thing
t1.join()
```

Joining a Thread

- Once you start a thread it runs independently.
- Use `t.join()` to wait for a thread `t` to exit

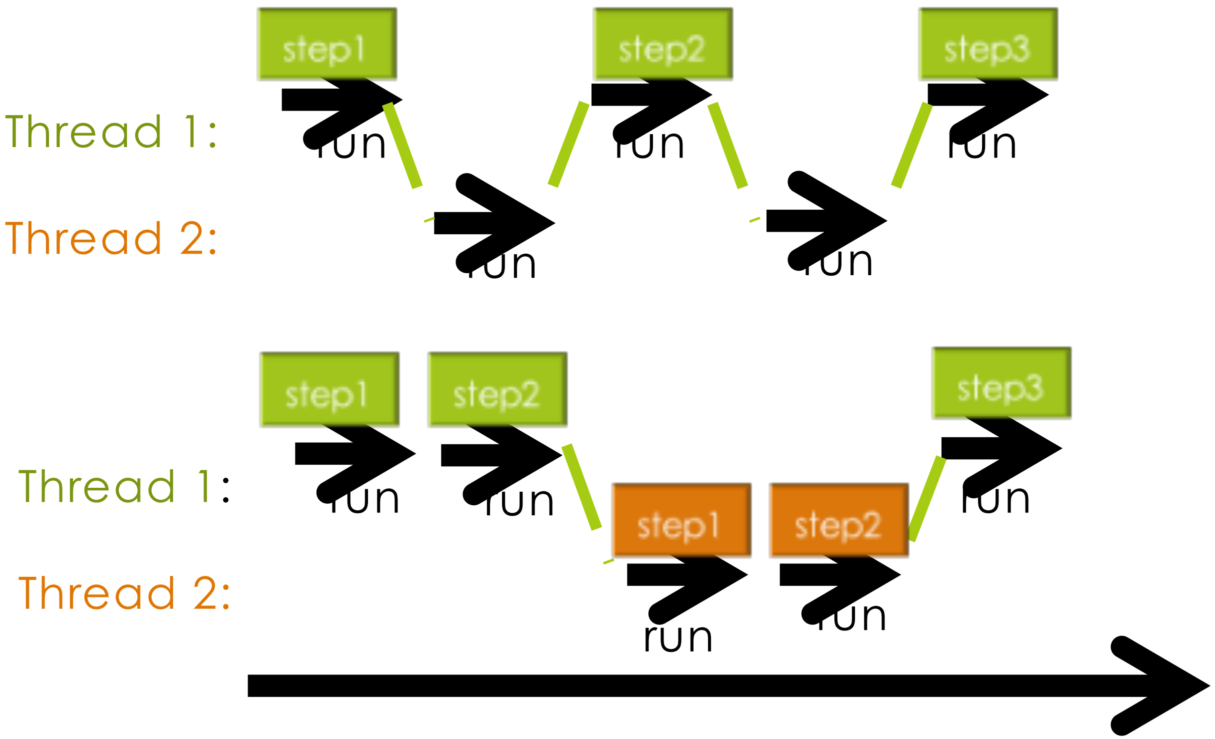
```
 t.start() # launch a thread t

# do other work
...

# wait for thread t to finish and exit
t.join()
```

Access to Shared Data

- Threads **share all of the data** in your program.
- **We cannot assume anything about scheduling** (the order of steps in an execution).
- **Operations that we think of as a single step are often *non-atomic*** (take several steps and might be interrupted).

Thread Scheduling

Thread 1 and Thread 2 are separate threads.



The dashed lines indicate the points in time at which a switch occurs.

We cannot assume anything about when these switches will occur.

Threads Sharing Data

- Consider a shared resource (variable x in this example)

```
              x = 0
```

Thread 1	**Thread 2**
`...`	`...`
`x = x + 1`	`x = x - 1`
`...`	`...`

Example

```
import threading
x = 0
def inc():
    global x
    for i in range(1000000):
         x = x + 1

def dec():
    global x
    for i in range(1000000):
         x = x - 1

t1 = threading.Thread(target = inc)
t2 = threading.Thread(target = dec)
t1.start()
t2.start()
t1.join()
t2.join()
print(x)
```

Example

```
import threading
x = 0
def inc():
    global x
    for i in range(1000000):
         x = x + 1

def dec():
    global x
    for i in range(1000000):
         x = x - 1

t1 = threading.Thread(target = inc)
t2 = threading.Thread(target = dec)
t1.start()
t2.start()
t1.join()
t2.join()
print(x)
```

Caution: Global variables should be used sparingly. They can be modified and read in a variety of places in the code. They make it hard to read, test and debug code.

Example

Run it several times. It may produce a different number each time. Why?

```
import threading
x = 0
def inc():
    global x
    for i in range(1000000):
         x = x + 1

def dec():
    global x
    for i in range(1000000):
         x = x - 1

t1 = threading.Thread(target = inc)
t2 = threading.Thread(target = dec)
t1.start()
t2.start()
t1.join()
t2.join()
print(x)
```

Low-level Atomic Steps

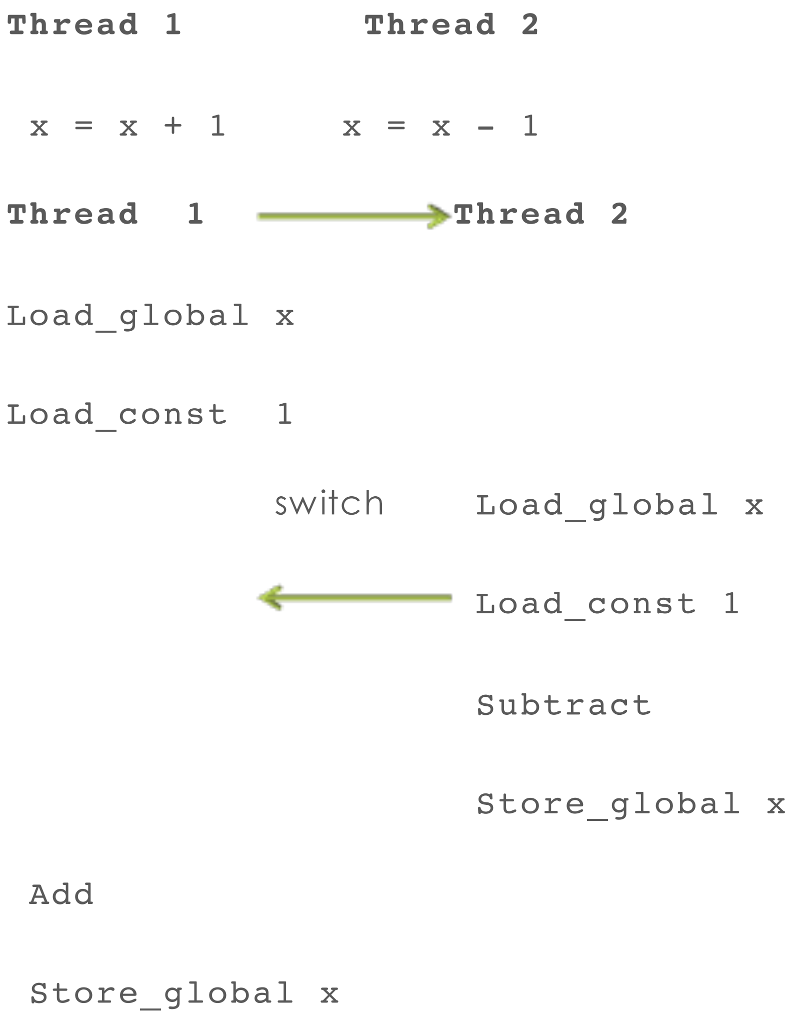
```
Thread 1          Thread 2

    ...                ...

  x = x + 1         x = x - 1

   ...                ...
```

We thought of addition and subtraction as one indivisible step but Python divided their execution into smaller steps

Low-level interpreter execution:

```
Thread  1             Thread 2

Load_global x         Load_global x

Load_const 1          Load_const  1

Add                   Subtract

Store_global x        Store_global x
```

A Possible Interleaving of Steps



```
Thread 1          Thread 2

 x = x + 1       x = x - 1

Thread  1          Thread 2

Load_global x

Load_const  1

            switch    Load_global x

                      Load_const 1

                      Subtract

                      Store_global x

 Add

 Store_global x
```

One of several possible interleavings of steps actually Took place

Think of starting execution at a state with x = 0. Can you see why the final value would be 1, not 0?

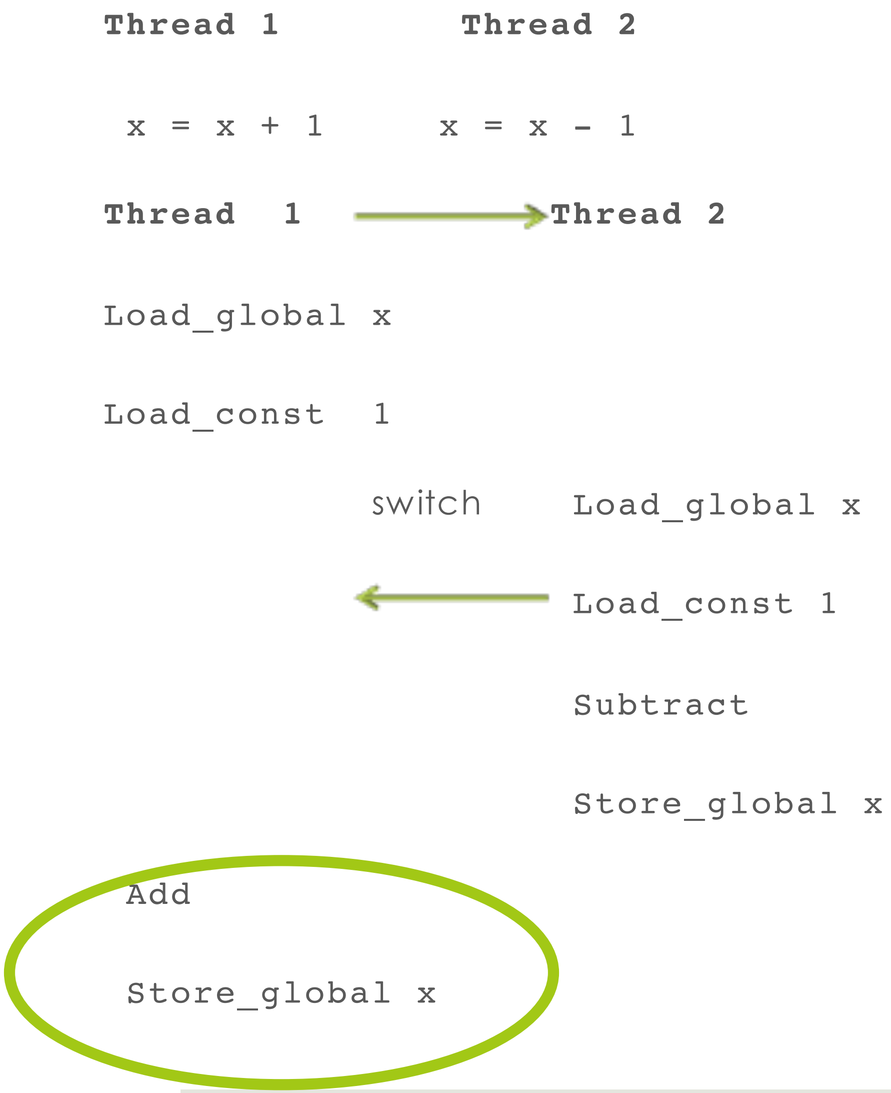
Not what the programmer intended

```
Thread 1        Thread 2

 x = x + 1     x = x - 1

Thread  1  ────────→ Thread 2

Load_global x

Load_const  1

            switch   Load_global x

         ←────────   Load_const 1

                     Subtract

                     Store_global x

 Add

 Store_global x
```

Operations performed on a stale value of x (i.e. 0) after x has been updated to -1 by Thread 1

- Knock, Knock
- Race Condition!
- Who’s There?

Race Condition



```
Thread 1            Thread 2

 x = x + 1        x = x - 1

Thread  1              Thread 2

Load_global x

Load_const  1

            switch    Load_global x

                      Load_const 1

                      Subtract

                      Store_global x

 Add

 Store_global x
```

Race condition: two or more threads operating on the same data object without proper synchronization

The output is dependent on the timing of uncontrollable events such as scheduling decisions of the underlying system

Concurrent programming is hard.

- Only a tiny percentage of practicing programmers can do it.
- It requires art and mathematics.
  - It's like digital hardware design.
  - It needs proofs.
- Conventional debugging doesn't work.
  - If you stop the program to observe, you change the behavior.
  - Testing is futile because the number of possible execution sequences for the same input explodes.

Summary

- Sequential vs. concurrent programming paradigms
  - Advantages of using concurrency: utilizing resources more efficiently, dealing with concurrent events in the computational environment
  - Challenges in concurrent programming
    - Synchronization between different tasks and access to shared data is a major source of complexity
    - Need to consider all possible executions
    - Difficulty of replicating errors
- We will NOT do any programming with threads. We looked at it only to illustrate the concepts of process scheduling, interleaving of actions, and race conditions.

Concurrency is hard…

Recap

- Process: program in execution. Unit of sequential execution.
- We can structure programs so that they can be executed as a set of concurrent processes
  - On a single processor
  - On multiple processors
- Processes may coordinate their actions using
  - Shared memory
  - Message passing
- A race condition is a situation in which multiple processes read and write a shared data item and the final result depends on the order of execution.

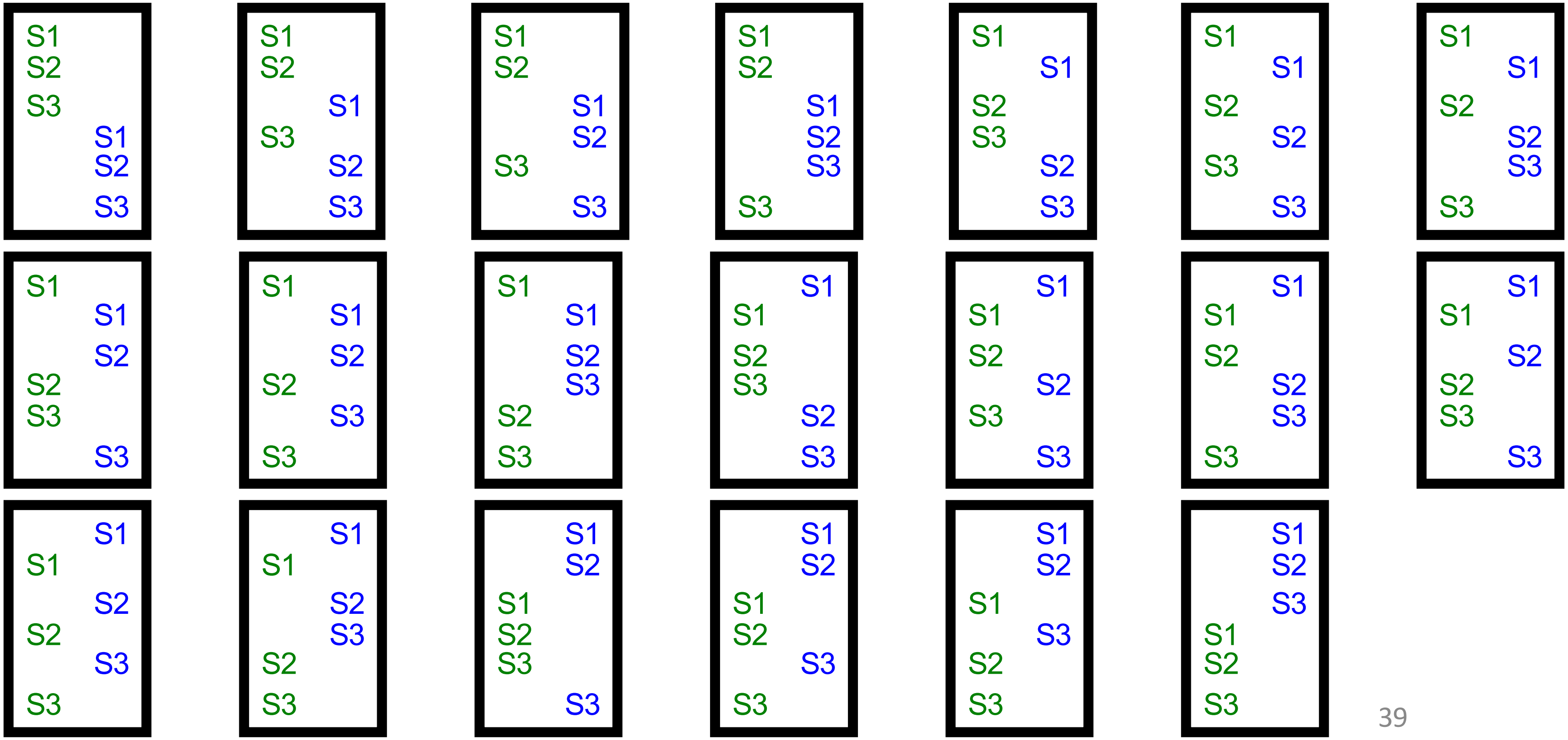
There are many ways to execute two processes concurrently.

The green process executes steps S1 S2 S3 in the given order.
The blue process executes steps S1 S2 S3 in the given order.

S1
S2
S3

S1
S2
S3

Several possible interleavings of steps.



S1	
S2	
S3	
	S1
	S2
	S3

S1	
S2	
	S1
S3	
	S2
	S3

S1	
S2	
	S1
	S2
S3	
	S3

S1	
S2	
	S1
	S2
	S3
S3	

S1	
	S1
S2	
S3	
	S2
	S3

S1	
	S1
S2	
	S2
S3	
	S3

S1	
	S1
S2	
	S2
	S3
S3	

S1	
	S1
	S2
S2	
S3	
	S3

S1	
	S1
	S2
S2	
	S3
S3	

S1	
	S1
	S2
	S3
S2	
S3	

	S1
S1	
S2	
S3	
	S2
	S3

	S1
S1	
S2	
	S2
S3	
	S3

	S1
S1	
S2	
	S2
	S3
S3	

	S1
S1	
	S2
S2	
S3	
	S3

	S1
S1	
	S2
S2	
	S3
S3	

	S1
S1	
	S2
	S3
S2	
S3	

	S1
	S2
S1	
S2	
S3	
	S3

	S1
	S2
S1	
S2	
	S3
S3	

	S1
	S2
S1	
	S3
S2	
S3	

	S1
	S2
	S3
S1	
S2	
S3	

Assumption

- In the rest of the lecture we will use some programs to illustrate concepts such as race conditions, interference, and deadlock. For the purposes of this lecture **we assume that a single line of program is executed atomically:** you can think of one line of code as corresponding to one step in the previous slide whose execution cannot be broken down into smaller steps.

Critical Sections

- Often, a process really needs exclusive access to some data.
- A **critical section** is a sequence of steps that have exclusive access to the shared resource
  - If multiple processes are sharing a resource only one should be executing its critical region
- Real Life Examples where critical sections are needed
  - Crossing a traffic intersection
  - A bank with many ATMs

Critical Section Example

- Consider a bank with multiple ATM's.
- At one, Mr. J requests a withdrawal of $10.
- At another, Ms. J requests a withdrawal of $10 from the same account.
- The bank's computer executes:
  1. For Mr. J, verify that the balance is big enough.
  2. For Ms. J, verify that the balance is big enough.
  3. Subtract 10 from the balance for Mr. J.
  4. Subtract 10 from the balance for Ms. J.

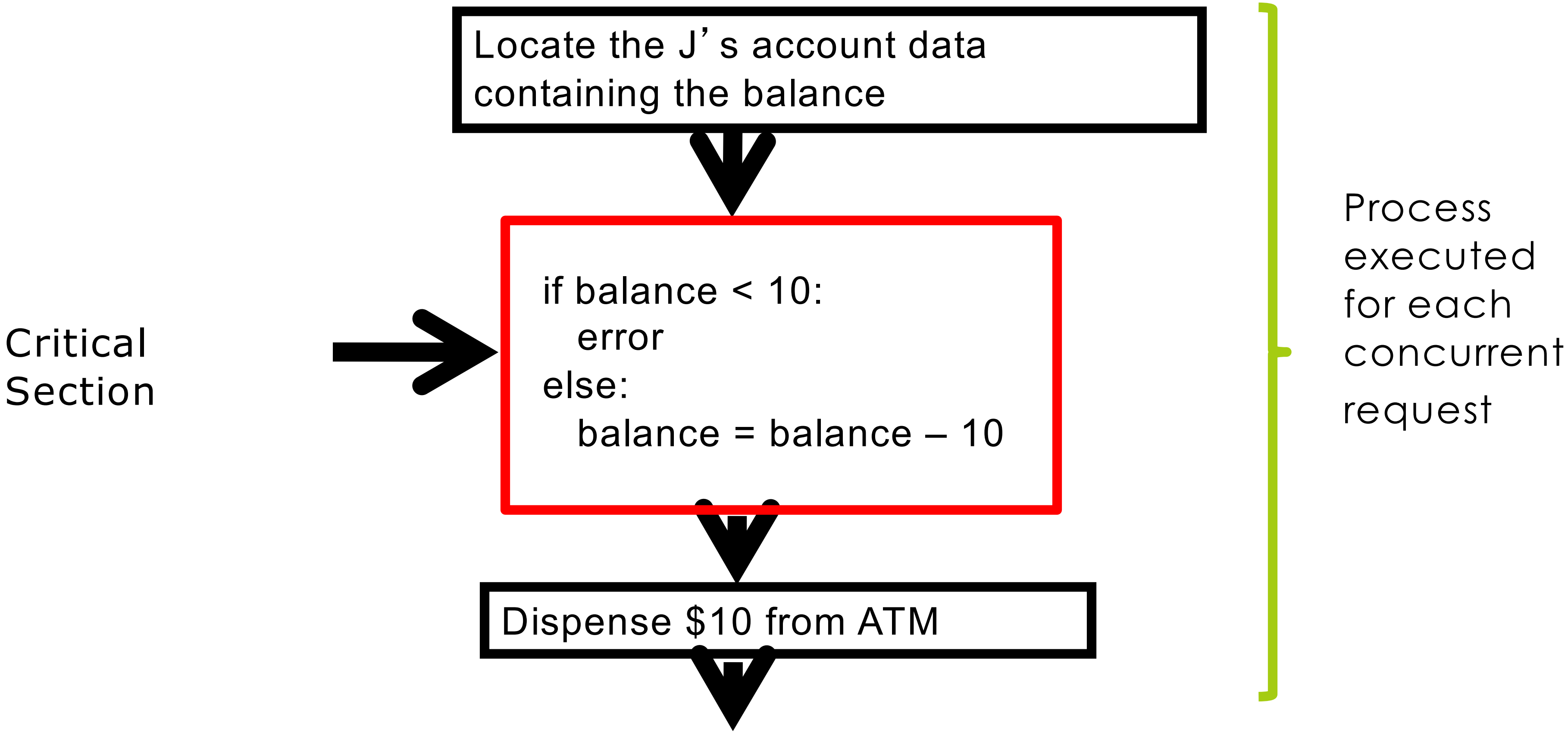
- The balance went negative if it was less than $20!

Vocabulary Reminder

- Race condition: A behavior in concurrent processing where proper functioning depends on the timing of other uncontrollable events

- A critical section is a piece of code that accesses a shared resource that must **not** be concurrently accessed by more than one process

Critical Sections in a Program



Locate the J's account data containing the balance

Critical Section

```
if balance < 10:
   error
else:
   balance = balance – 10
```

Dispense $10 from ATM

Process executed for each concurrent request

What can we do to prevent one processor from entering the critical section while another is in it?

Careful Driver Method: Don't enter the intersection unless it's empty.

In shared memory: **free = True #initially unlocked**

```
# Process 1
while True :
  Non-Critical_Section
  while not free:
    pass
  free = False
  Critical_Section
  free = True
```

```
# Process 2
while True :
  Non-Critical_Section
  while not free:
   pass
  free = False
  Critical_Section
  free = True
```

code that does not touch shared memory

code that touches shared memory

Interference is possible!

Careful Driver Method: Don’t enter the intersection unless it’s empty.

In shared memory: **free = True #initially unlocked**

```
# Process 1
while True :
  Non-Critical_Section
  while not free :
    pass
  free = False
  Critical_Section
  free = True
```

```
# Process 2
while True :
  Non-Critical_Section
  while not free :
      pass
  free = False
  Critical_Section
  free = True
```

If these two processes leave their non-critical sections at precisely the same time, then strictly alternate lines, they will both end up in the Critical_Section.

Computers vs. Real Life

- The careful driver method works in real life because
  - The number of times in your life you cross the intersection is low. Twice a day for forty years is about 29,000.
  - The chance of two drivers arriving at the intersection simultaneously is low.
  - Cars move slowly enough that if you don't see anyone coming, you'll get across before anyone comes.

The Stop Sign Method

STOP
STOP
©2000 How Stuff Works

1. Signal your intention (by stopping).
2. Wait until cross road has no one waiting or crossing.
3. Cross intersection.
4. Renounce intention (by leaving intersection).

The Stop and Look Method

```
# Shared Memory
 free0 = True   # P0 is not stopped at sign
 free1 = True   # P1 is not stopped at sign
```

```
# Process 0
while True :
   Non-Critical_Section
   free0 = False
   while not free1 :
       pass
   Critical_Section
   free0 = True
```

```
# Process 1
while True :
   Non-Critical_Section
   free1 = False
   while not free0:
       pass
   Critical_Section
   free1 = True
```

This version of the code does not suffer from interference. Does it now guarantee safe execution of the critical section?

The Stop and Look Method

```
# Shared Memory
 free0 = True   # P0 is not stopped at sign
 free1 = True   # P1 is not stopped at sign
```

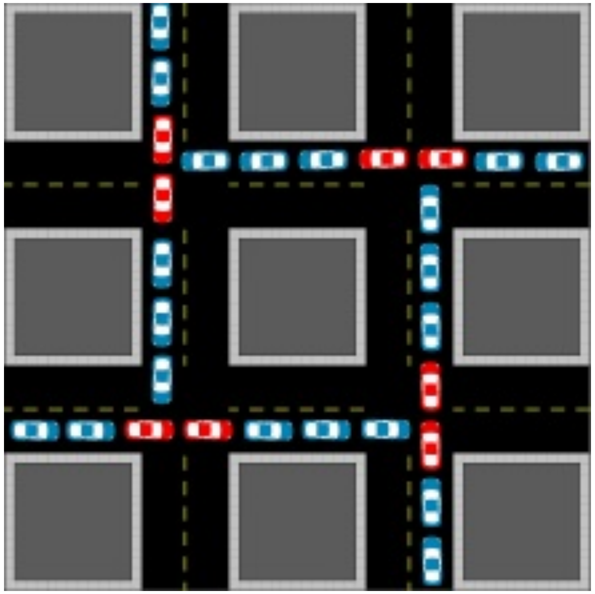
```
# Process 0
while True :
   Non-Critical_Section
   free0 = False
   while not free1:
       pass
   Critical_Section
   free0 = True
```

```
# Process 1
while True :
   Non-Critical_Section
   free1 = False
   while not free0:
       pass
   Critical_Section
   free1 = True
```

Once again, if the two processes exit the non-critical section at the same time and strictly alternate lines they will end up stuck in their while loops. This is called Deadlock.

Deadlock

- Deadlock is the condition when two or more processes are all waiting for some shared resource, but no process actually has it to release, so all processes to wait forever without proceeding.
- It’ s like gridlock in real traffic.



The Stop Sign Method with Tie Breaking

STOP
STOP
©2000 How Stuff Works

1. Signal your intention (by stopping).
2. Wait until cross road has no one else waiting or crossing.
3. If two of you are both waiting, yield to the car to your right.
4. Cross intersection.
5. Renounce intention (by leaving intersection).

Peterson's algorithm avoids all bugs!

```
free0 = True
free1 = True
priority = 0
```

```
# Process 0
  while True :
    Non-Critical_Section0
    free0 = False
    priority = 1
    while not free1 and
          priority==1:
       pass
    Critical_Section0
    free0 = True
```

```
#  Process 1
  while True :
    Non-Critical_Section1
    free1 = False
    priority = 0
    while not free0 and
          priority==0:
        pass
    Critical_Section1
    free1 = True
```

Peterson’s algorithm avoids all bugs!

```
free0 = True
free1 = True
priority = 0
```

```
# Process 0
  while True :
    Non-Critical_Section0
    free0 = False
    priority = 1
    while not free1 and
          priority==1 :
       pass
    Critical_Section0
    free0 = True
```

```
#  Process 1
  while True :
    Non-Critical_Section1
    free1 = False
    priority = 0
    while not free0 and
          priority==0 :
       pass
    Critical_Section1
    free1 = True
```

Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section (free1 == True) or if P1 has given priority to P0 by setting priority to 0 (priority == 0).

A Probabilistic Approach

- There is a conceptually easier way to solve synchronization problem by embracing probable thinking. We just use the stop sign approach but wait for a random amount of time when a conflict occurs.

Types of HeisenBugs*

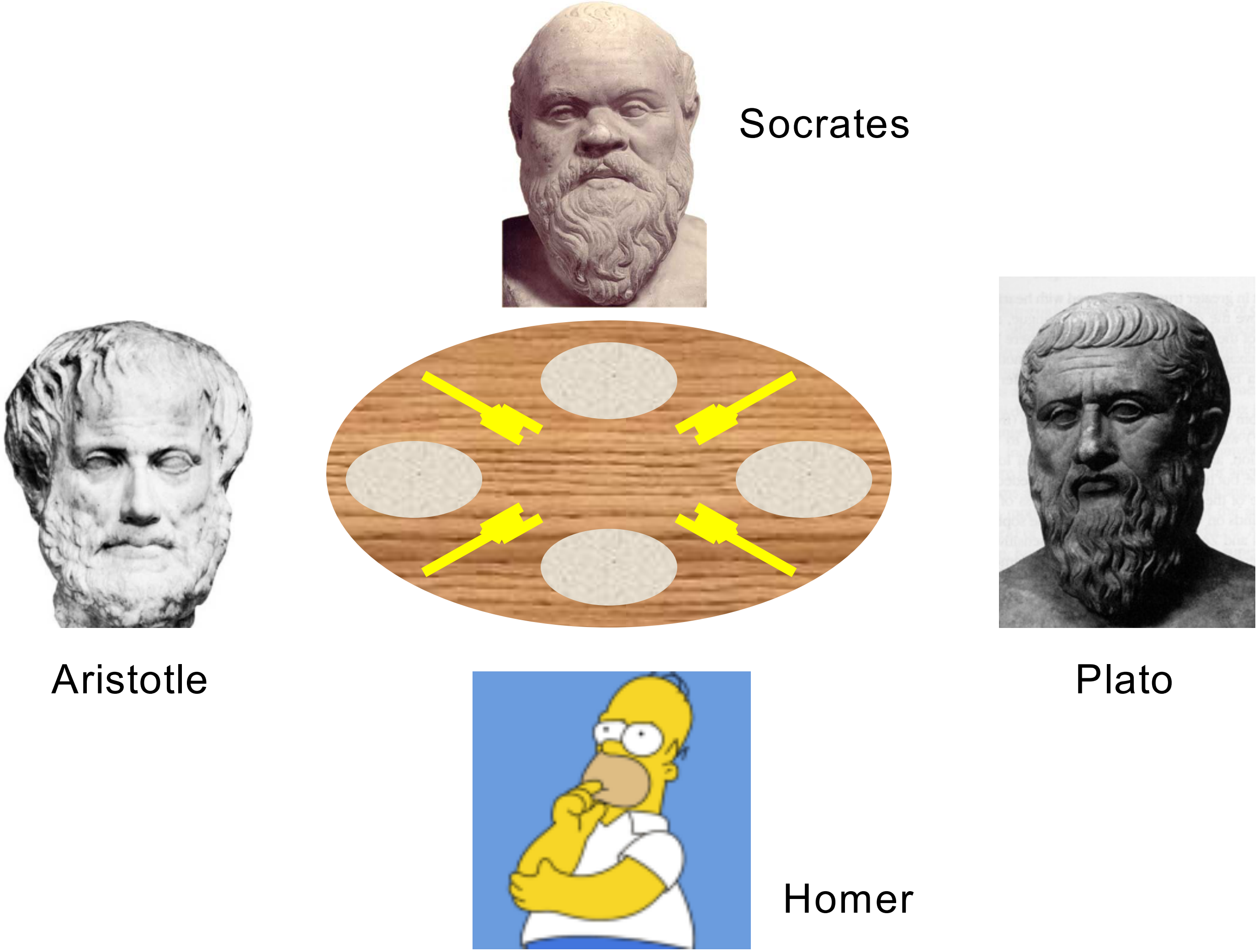
In decreasing order of seriousness:

1. Interference: multiple process in critical section.
2. Deadlock: two processes idle forever, neither entering their critical or non-critical sections.
3. Starvation: one process needlessly idles forever while the other stays in its non-critical section.
4. Unfairness: a process has lower priority for no reason.

Note: We did not discuss 3 and 4 in detail. You can learn more about them in the future.

* In computer programming jargon, a heisenbug is a software bug that seems to disappear or alter its behavior when one attempts to study it. Source: Wikipedia

Dining Philosophers' Problem



Socrates

Aristotle

Plato

Homer

The Dining Philosophers

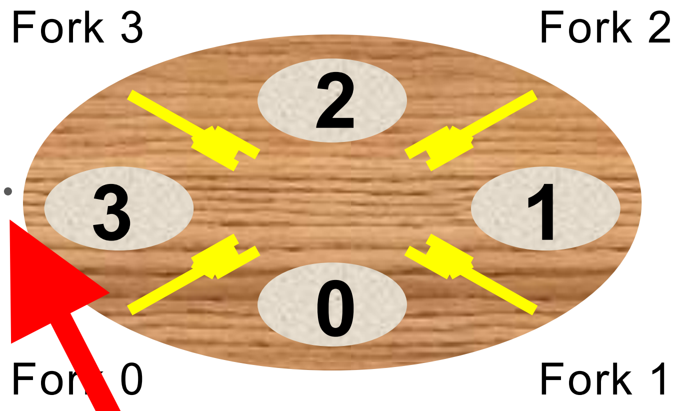
- Each philosopher thinks for a while, then picks up his left fork, then picks up his right fork, then eats, then puts down his left fork, then puts down his right fork, thinks for a while...
  - We assume here that each philosopher thinks and eats for random times, and a philosopher cannot be interrupted while he picks up or puts down a single fork.
- Each fork models a "resource" on a computer controlled by an OS.
- Original problem was proposed by Edsgar Dijkstra.

Dining Philosophers' Problem

- There are N philosophers.
- Philosopher i does the following:

N=4

1. THINK
2. Pick up fork i.
3. Pick up fork (i+1) modulo N.
4. EAT
5. Put down fork i.
6. Put down fork (i+1) modulo N.
7. Go to step 1.



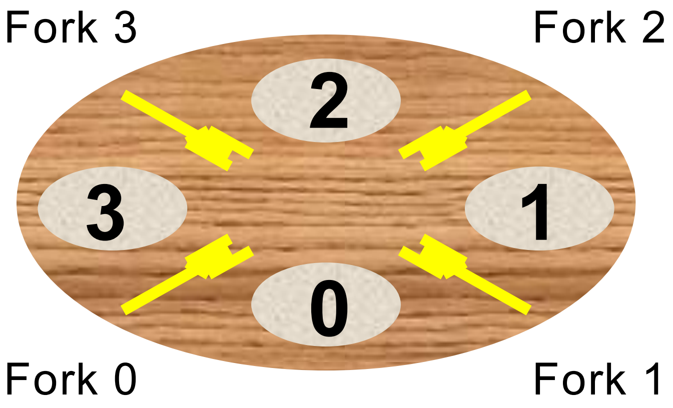
NOTE: (i+1) modulo N = i+1 , if 0 ≤ i < N-1
(i+1) modulo N = 0, if i = N-1

Dining Philosophers' Problem

- There are N philosophers.
- Philosopher i does the following:

1. THINK
2. Pick up fork i.
3. Pick up fork (i+1) modulo N.
4. EAT
5. Put down fork i.
6. Put down fork (i+1) modulo N.
7. Go to step 1.

N=4



How can deadlock occur here?

Removing the Deadlock

- Philosopher i does the following:

```
1. THINK
2. If i is not equal to N-1:
        a. Pick up fork i
        b. Pick up fork i + 1
3. If i is equal to N-1:
        a. Pick up fork 0
        b. Pick up fork N - 1
```

This philosopher picks up the right fork first

```
4. EAT
5. If i is not equal to N-1:
        a. Put down fork i
        b. Put down fork i + 1
6. If i is equal to N-1:
        a. Put down fork 0
        b. Put down fork N – 1
```

This philosopher picks up the right fork first

```
7. Go to step 1
```

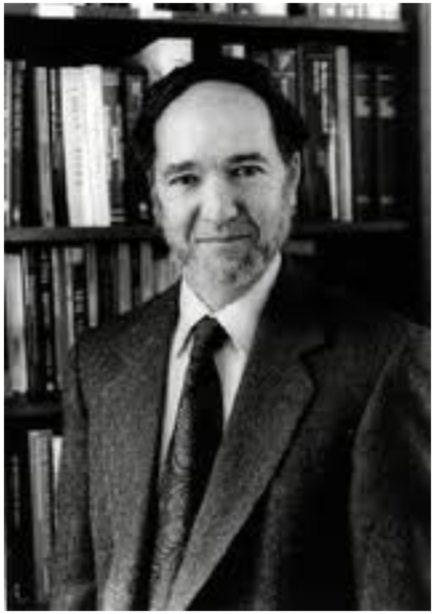
Afterthoughts

Some counter-intuitive ideas about bugs and risks.

This man removed all the traffic lights and signs!

It reduced accidents by 50% and reduced congestion. Pedestrian and driver anxiety increased.

Why did Jared Diamond sleep under a tree when his aborigine companion wouldn’t?



There is only a 1/10,000 chance of a particular tree falling on any given night. But aborigines sleep in the woods every night so if they tempted fate every night for 40 years there is only a 23% chance of not dying that way, i.e. (1-1/10,000)^(40*365) = 0.23.

Why is a 1% chance of a bug biting better than a 0.1% chance?

- If there is a 1% chance of error, the bug will show up during 100 days of testing.

- If there is a 0. 1% chance, the bug will show up after three years when the system is deployed.