Lecture 2 Supplement: Nondeterminism and QA

Wonsun Ahn

University of Pittsburgh

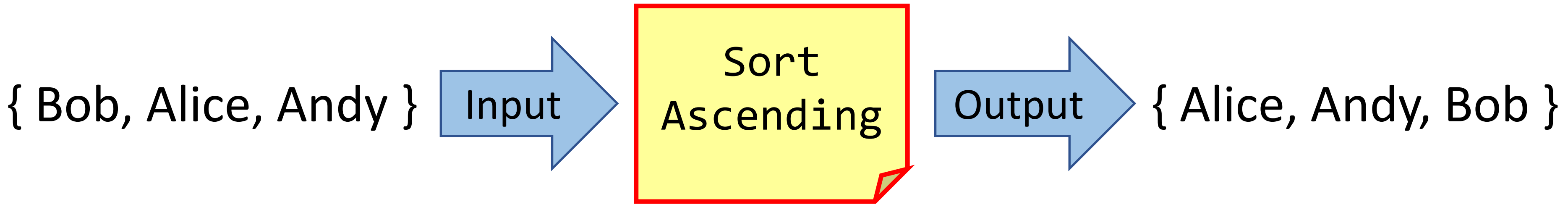
Learning Goals

- Learn what **nondeterminism** is and why it is an issue with SW testing
- Learn **4 sources** of nondeterminism
- Learn **3 tools** that combat nondeterminism

- To understand this fully, it helps to know:
  - C programming
  - Concept of program stack and heap
  - Concept of threads and parallel execution
  - If you don’t, I will go slowly on some concepts so don’t worry
- These slides extend Lecture 2: Testing Theory

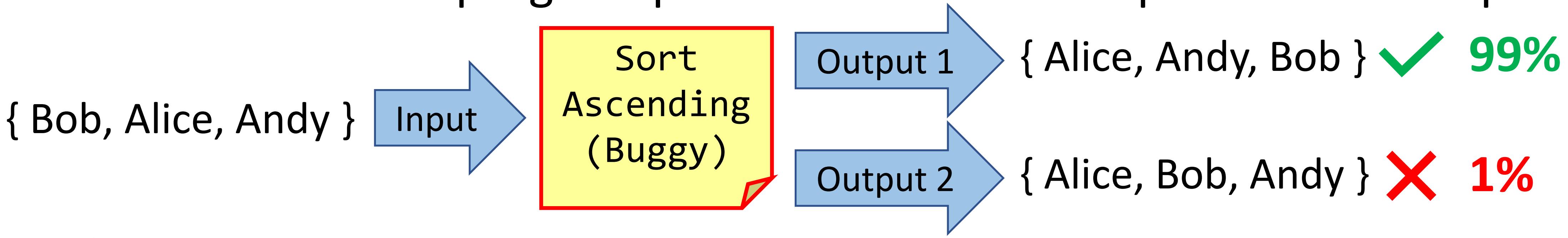
What is Nondeterminism?

- When the output of a program is not determined by its input
- A deterministic program produces the same output on the same input:



{ Bob, Alice, Andy } Input → Sort Ascending → Output { Alice, Andy, Bob }

- A nondeterministic program produces different outputs on same input!



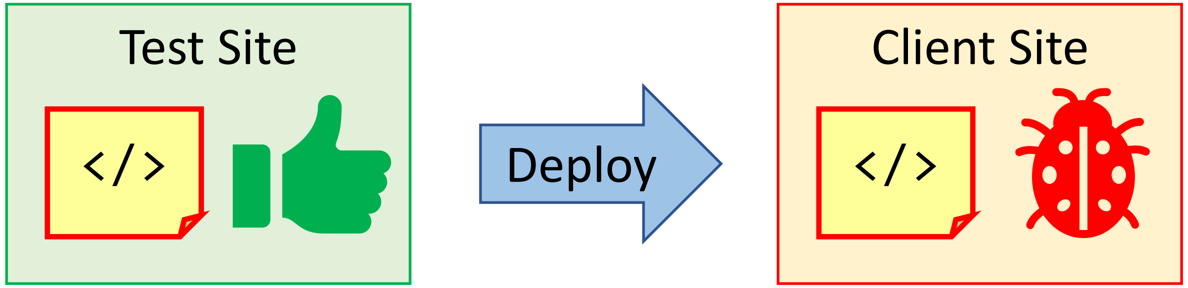
{ Bob, Alice, Andy } Input → Sort Ascending (Buggy)

Output 1 { Alice, Andy, Bob } ✓ 99%

Output 2 { Alice, Bob, Andy } ✗ 1%

Nondeterminism Makes Testing Hard

- Surprise defects
  - Defect not revealed during testing suddenly pops up during usage



Test Site

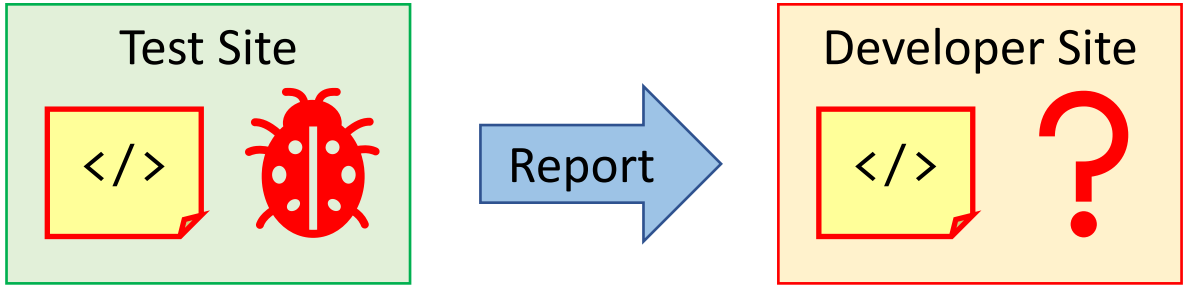
</>

Deploy

Client Site

</>

- Unreproducible defects
  - Defect revealed during testing doesn’t show up when trying to debug it



Test Site

</>

Report

Developer Site

</>

So What To Do?

- Depends on what kind of nondeterminism it is.

1. Nondeterminism by mistake
    - Coder never intended nondeterminism; nondeterminism itself is the defect

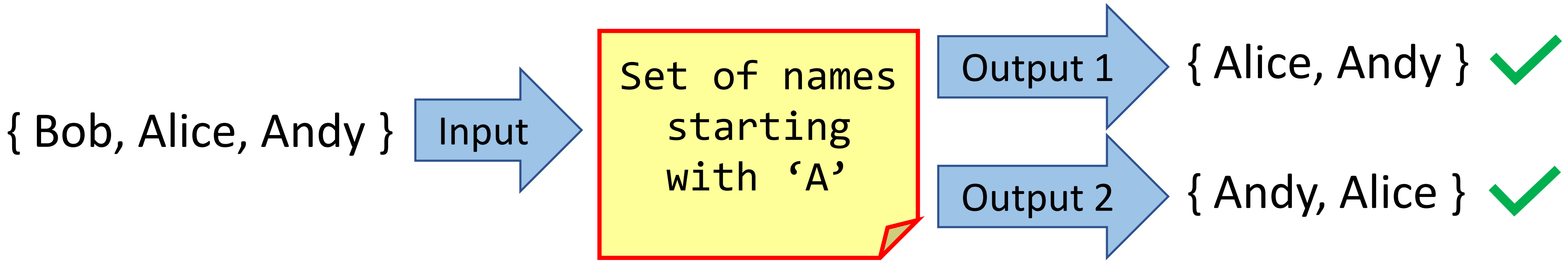
    → Stamp out the nondeterminism!

2. Nondeterminism by design
    - Coder intended the nondeterminism

    → Must somehow deal with the nondeterminism.

Why Nondeterminism by Design?

- Consider the following nondeterministic program:



{ Bob, Alice, Andy }

Input

Set of names starting with ‘A’

Output 1

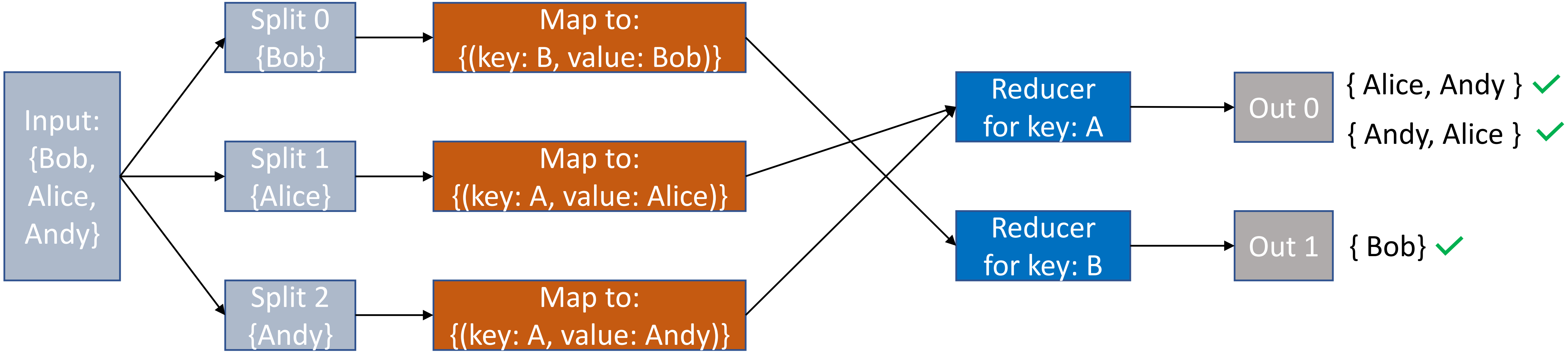
{ Alice, Andy } ✓

Output 2

{ Andy, Alice } ✓

  - Both outputs are correct since there is no ordering constraint in a set
- Less constraints usually means the program can run faster!
  - A straightforward loop checking each name is deterministic but slow
  - But what if we used parallel MapReduce to speed up the program?

MapReduce Implementation of Filter Names



Map:
String → (first letter of String, String)

Reduce:
Strings → concatenated CSV

- Reducer concatenates in the order of arrival → nondeterministic output!
- All non-commutative reducers like concatenation have this property
- For determinism, must constrain order of mapping → slows down program!

Nondeterminism is Inherent in Parallelism

- “Nondeterminism in MapReduce considered harmful? An empirical study on non-commutative aggregators in MapReduce programs.”
    - Xiao, Tian et al., International Conference on Software Engineering, 2014.
    - https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/icsecomp14seip-seipid15-p.pdf

- Most other parallel frameworks suffer from nondeterminism as well
    - Including POSIX Threads, Java Threads, OpenMP, MPI, CUDA, TensorFlow
    - To allow flexibility in execution order for maximal performance

Why Nondeterminism by Design?

1. To make programs go faster through parallel execution
   - Sometimes all outputs are equally correct → nondeterminism is not a problem
   - Sometimes, for optimization problems, some outputs are “better” than others
     - Still, nondeterministic output is okay as long as it is better than deterministic output (given the same amount of time to run; i.e. nondeterministic approaches optimum faster)
     - True for NP-hard problems (e.g. integer linear programming, constraint solving, …)

2. To intentionally introduce randomness (random number generation)
   - Video games – to introduce random events into the game
   - Cryptography – to make cryptographic keys unpredictable

Outline

- Nondeterminism by mistake
  - Memory errors (examples / solutions)
  - Datarace errors (examples / solutions)
- Nondeterminism by design
  - Thread interleaving (examples / solutions)
  - Random number generation (examples / solutions)
- Summary

Nondeterminism by Mistake

It’s a Mistake – Stamp out from your Code!

- Due to erroneous code
  - Memory errors
  - Datarace errors

- Runtime behavior of program is undefined or barely defined
  - Called errors because they are illegal in language specification

- Undefined behavior can hardly be intentional by design
  - These behaviors need to be banished!

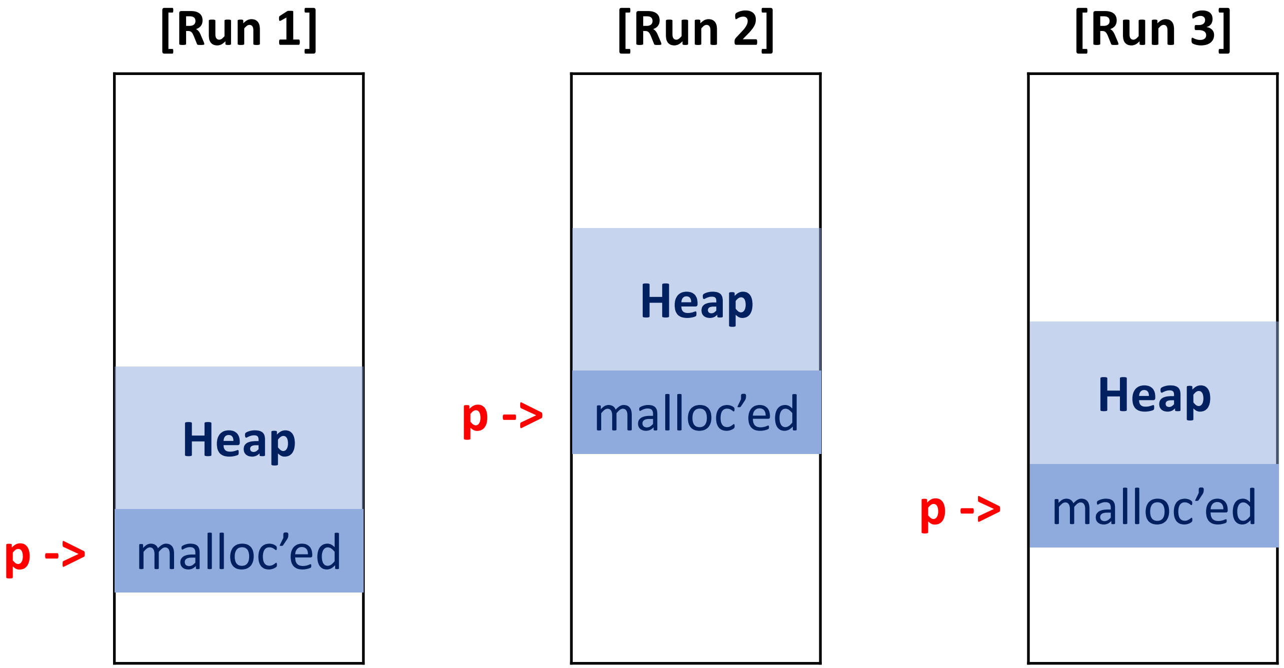
Outline

- Nondeterminism by mistake
  - Memory errors (examples / solutions)
  - Datarace errors (examples / solutions)
- Nondeterminism by design
  - Thread interleaving (examples / solutions)
  - Random number generation (examples / solutions)
- Summary

It’s Very Easy to Make a Random C Program

```
int main() {
   char *p = malloc(8);
   printf("p = %p\n", p);
   free(p);
   return 0;
}
```

```
bash-4.2$ ./heap
p = 0x10b2010
bash-4.2$ ./heap
p = 0x257e010
bash-4.2$ ./heap
p = 0x13a7010
```

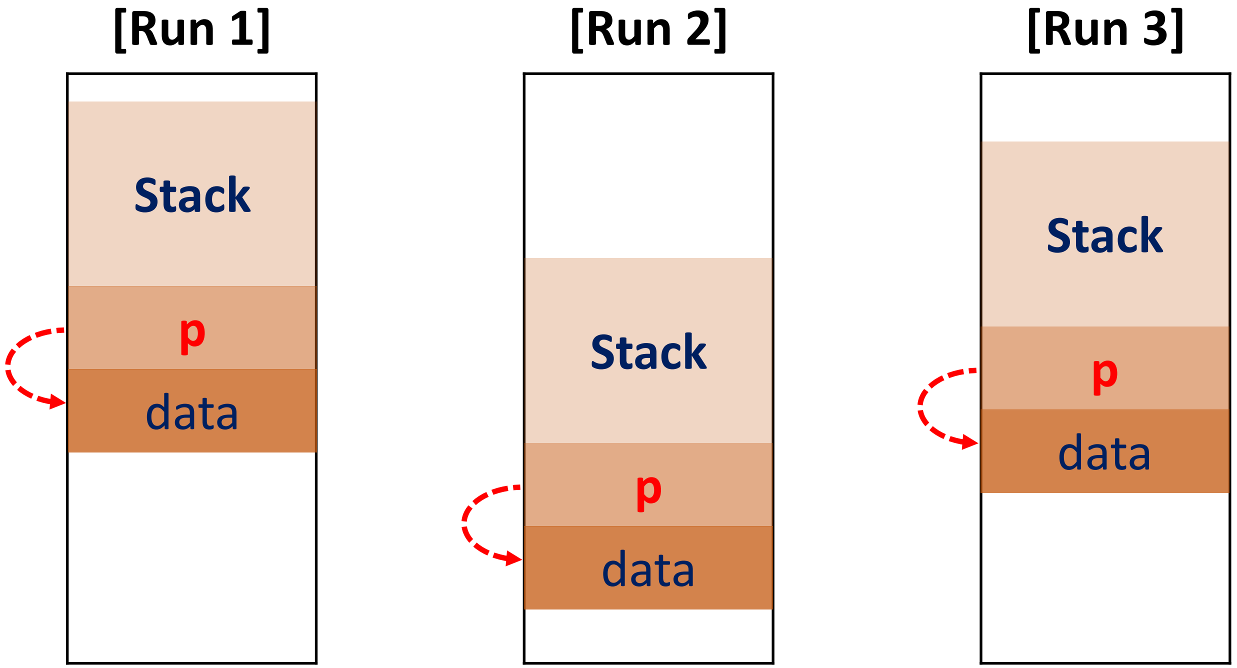


- Why is malloc returning a random address?
- Address Space Layout Randomization (ASLR)
- To prevent hackers from guessing memory layout

Stack Addresses are Random Too

```
int main() {
   char *p;
   char data[8];
   p = data;
   printf("p = %p\n", p);
   return 0;
}
```

```
bash-4.2$ ./stack
p = 0x7ffff5443188
bash-4.2$ ./stack
p = 0x7ffedfb740f8
bash-4.2$ ./stack
p = 0x7fffc21002f8
```



- `p` is now pointing to a stack location
- Why is `data[8]` at a random address?
- ASLR is also applied to the stack

Does it Matter?

- Aren’t these contrived examples?
- Pointer addresses are almost never part of program output anyway
    - Unless you are printing them out for debugging or diagnostic purposes
    - Most of the times you output the data stored inside those locations
      (e.g. You output the data inside a data structure node not the node address)
- But addresses can leak out to program output *by mistake*
- Specifically when you have *memory errors*

Memory Errors

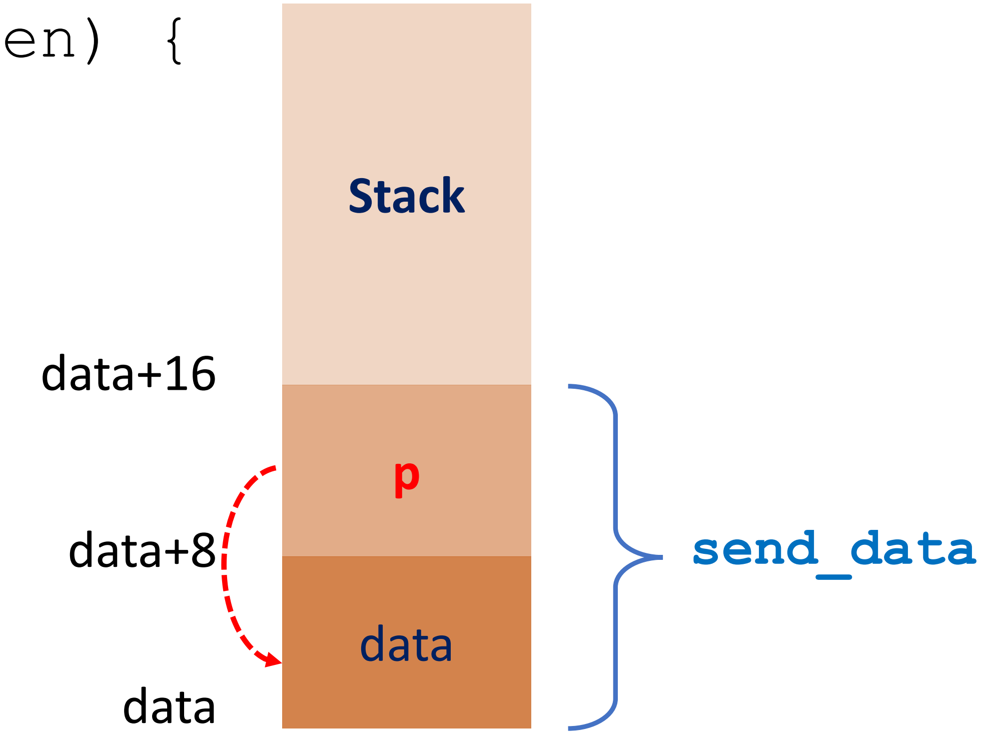
- Errors that access an illegal memory location are the culprits
  - Buffer overflow: access beyond the bounds of an array
  - Dangling pointer: access to already freed memory pointed to by pointer

- If illegal location contains an address, it can leak out to the output!

- Only happens in languages like C/C++ with direct access to memory
  - Does not happen in memory-managed languages like Java or Python
  - C/C++ is used to write most system code so still a big problem

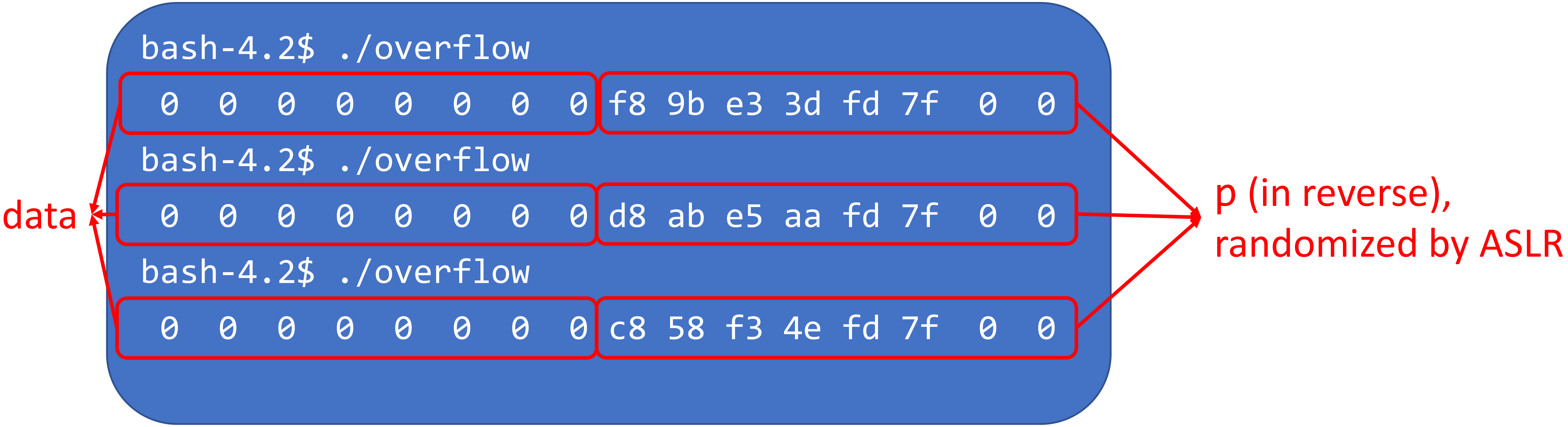
Buffer Overflow Example

```
void send_data(char *data, int len) {
  for (int i=0; i < len; i++)
    printf("%2hhx ", data[i]);
  printf("\n");
}
int main() {
  char *p;
  char data[8] = {0};
  p = data;
  send_data(data, 16);
  return 0;
}
```



p is sent along with data!

Buffer Overflow Output



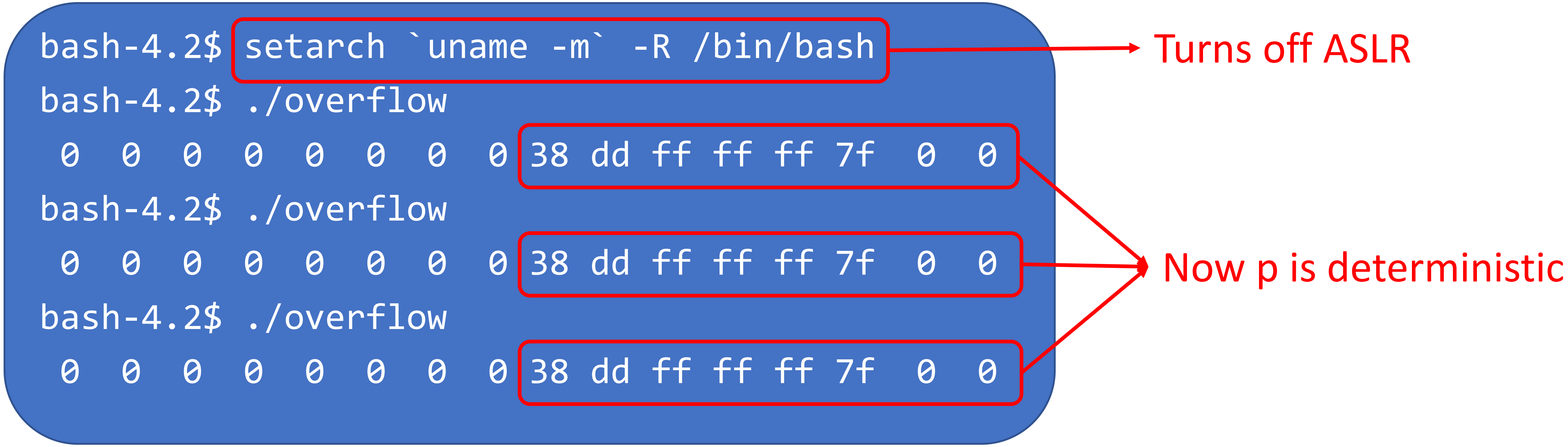
```
bash-4.2$ ./overflow
  0  0  0  0  0  0  0  0 f8 9b e3 3d fd 7f  0  0
bash-4.2$ ./overflow
  0  0  0  0  0  0  0  0 d8 ab e5 aa fd 7f  0  0
bash-4.2$ ./overflow
  0  0  0  0  0  0  0  0 c8 58 f3 4e fd 7f  0  0
```

data

p (in reverse),
randomized by ASLR

- Randomized addresses can leak out to output due to a memory error!

You could Turn Off ASLR …



```
bash-4.2$ setarch `uname -m` -R /bin/bash
bash-4.2$ ./overflow
  0  0  0  0  0  0  0  0 38 dd ff ff ff 7f  0  0
bash-4.2$ ./overflow
  0  0  0  0  0  0  0  0 38 dd ff ff ff 7f  0  0
bash-4.2$ ./overflow
  0  0  0  0  0  0  0  0 38 dd ff ff ff 7f  0  0
```

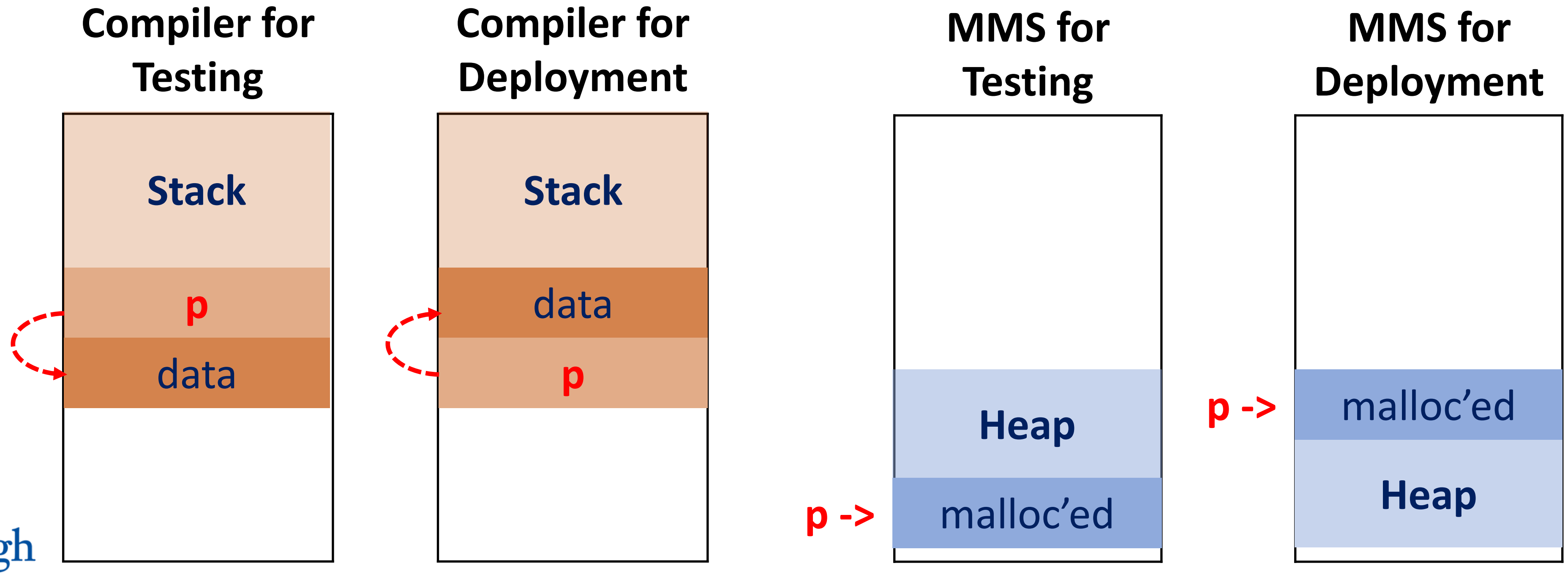
Turns off ASLR

Now p is deterministic

- Can help in reproducing bugs in a debug setting
- But clients will still want ASLR on for security → surprise defects can still happen

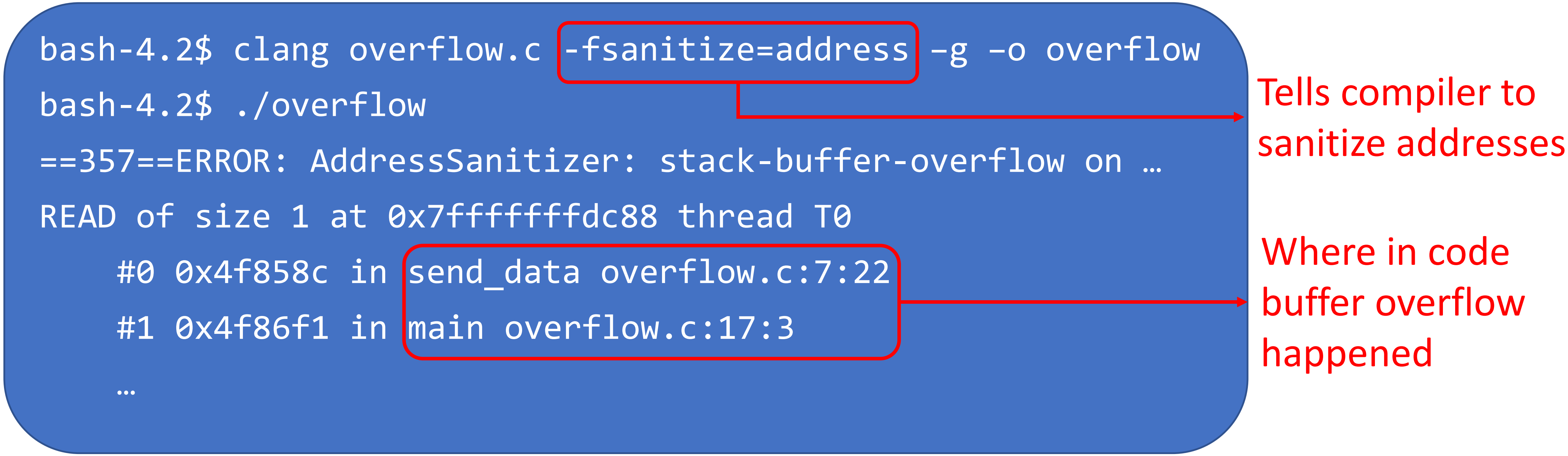
You could Turn Off ASLR …

- Even if client does not use ASLR, things can still go wrong
  - If binary deployed to client uses different *compiler* than test version
    (Or same compiler but different compile options)
  - If client uses a different runtime *memory management system* (MMS)
    (Or same MMS but MMS is nondeterministic on parallel mallocs)



What to do? Stamp Out the Error!

- Let’s use Google Address Sanitizer for this purpose



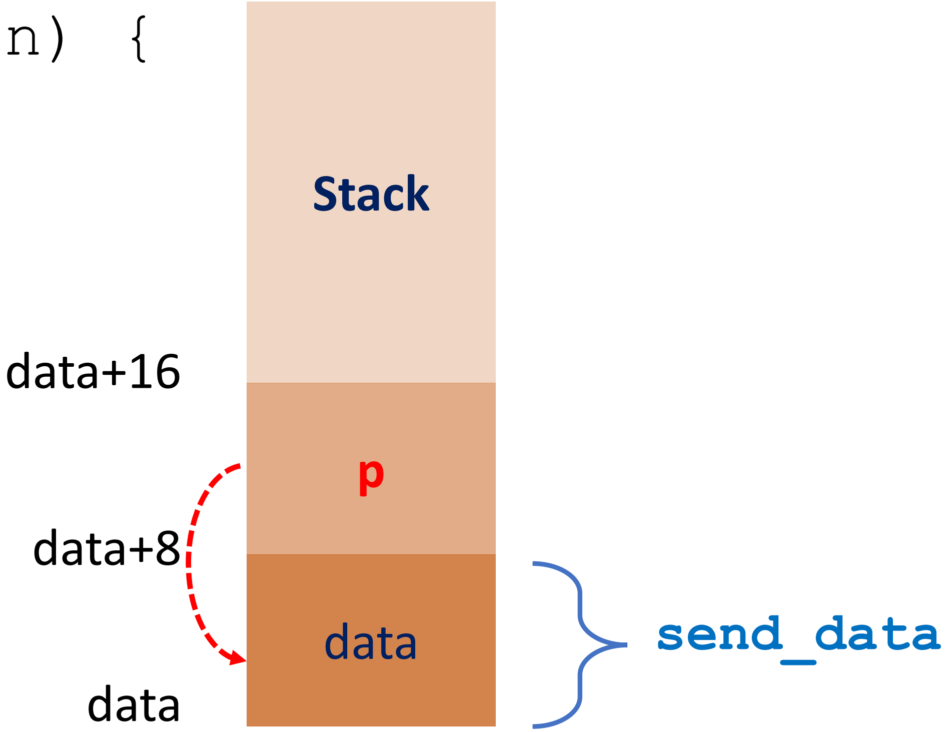
```
bash-4.2$ clang overflow.c -fsanitize=address -g -o overflow
bash-4.2$ ./overflow
==357==ERROR: AddressSanitizer: stack-buffer-overflow on …
READ of size 1 at 0x7ffffffdc88 thread T0
    #0 0x4f858c in send_data overflow.c:7:22
    #1 0x4f86f1 in main overflow.c:17:3
    …
```

Tells compiler to sanitize addresses

Where in code buffer overflow happened

Buffer Overflow Fixed

```
void send_data(char *data, int len) {
  for (int i=0; i < len; i++)
    printf("%2hhx ", data[i]);
  printf("\n");
}
int main() {
  char *p;
  char data[8] = {0};
  p = data;
  printf("p = %p\n", p);
  send_data(data, 8);
  return 0;
}
```



Now only `data` is sent!

Google Address Sanitizer (ASAN)

- Part of Google sanitizer suite
    - Address Sanitizer (ASAN)
    - Thread Sanitizer (TSAN)
    - Memory Sanitizer (MSAN)
    - And more.
- Works through *instrumentation*
    - Inserting extra instructions into program for the purpose of monitoring
    - Instrumentation code reports back issues during/after execution of code
- Compilers where available (with the `-fsanitize=address` switch)
    - LLVM (clang) starting with version 3.1
    - GCC starting with version 4.8

Outline

- Nondeterminism by mistake
  - Memory errors (examples / solutions)
  - Datarace errors (examples / solutions)

- Nondeterminism by design
  - Random number generation (examples / solutions)
  - Thread interleaving (examples / solutions)

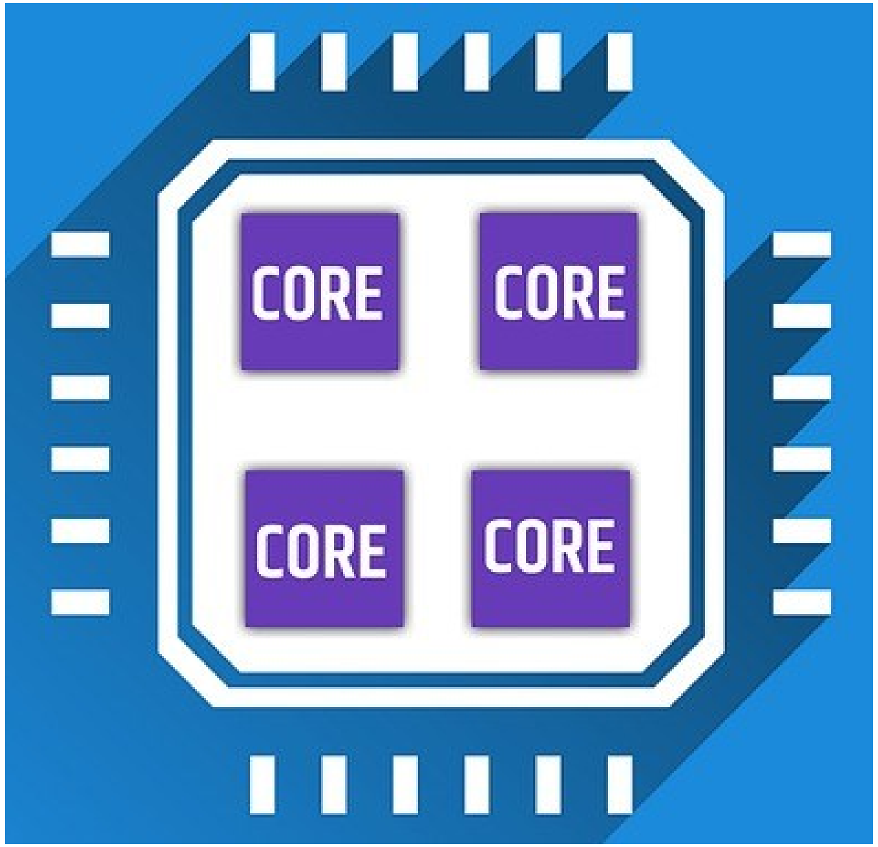
- Summary

First, an Intro to Parallel Programming

- Your laptop or cellphone has multiple CPUs (Seen on the right: quad-core with 4 CPUs)

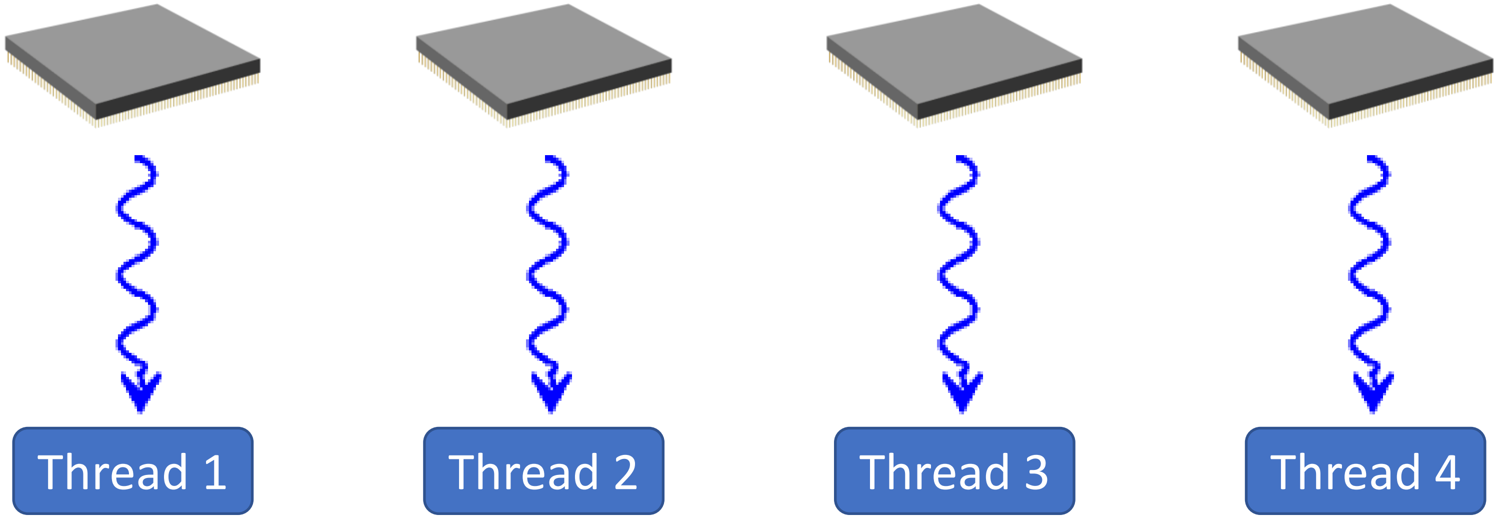
- A program runs on just 1 CPU by default (Using just 25% of the computing power!)

- A parallel program can use all 4 CPUs (Utilizing your computer 100%)



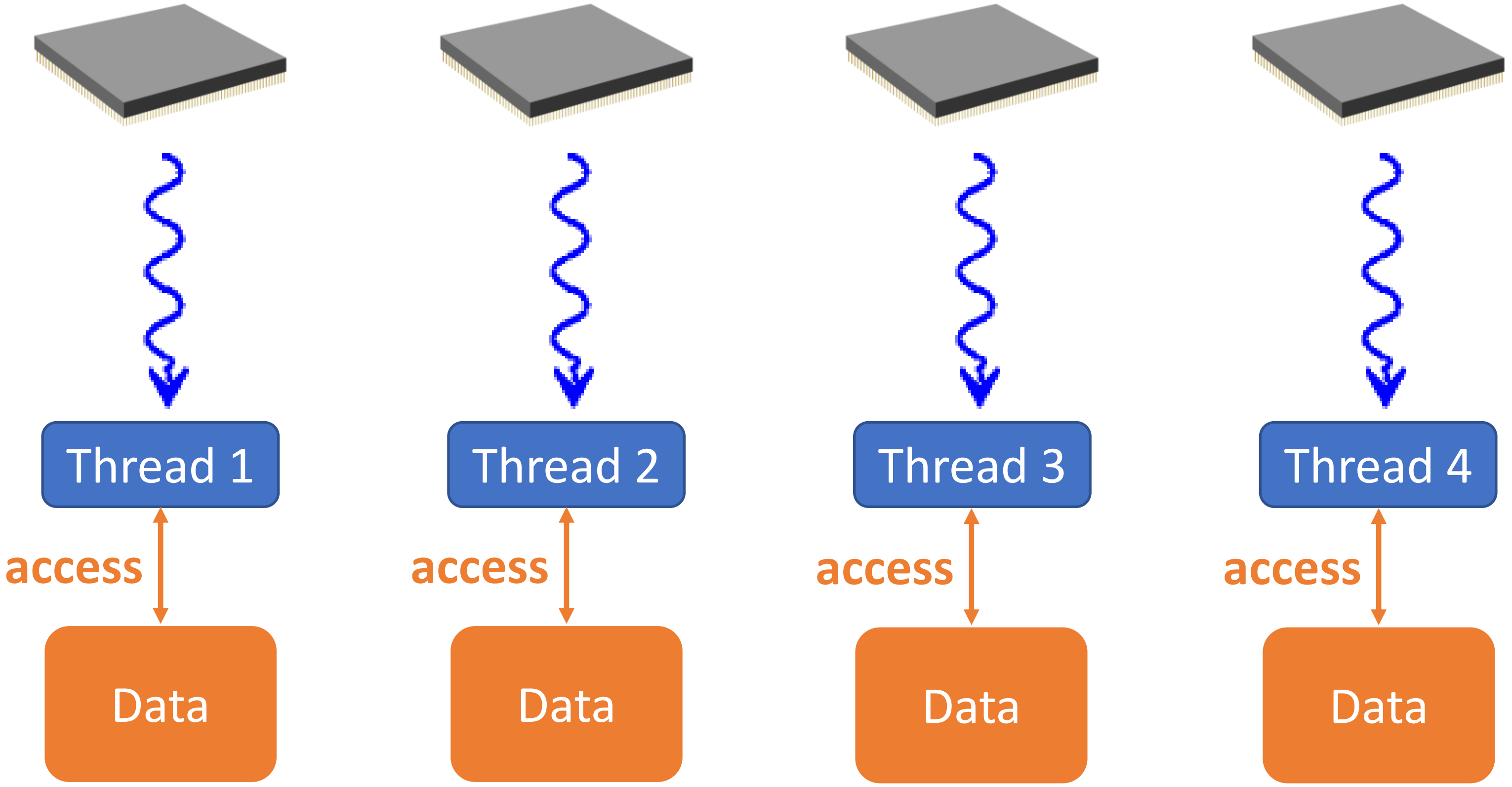
A Parallel Program Runs 1 Thread per CPU

- Thread: a unit of code that runs on 1 CPU



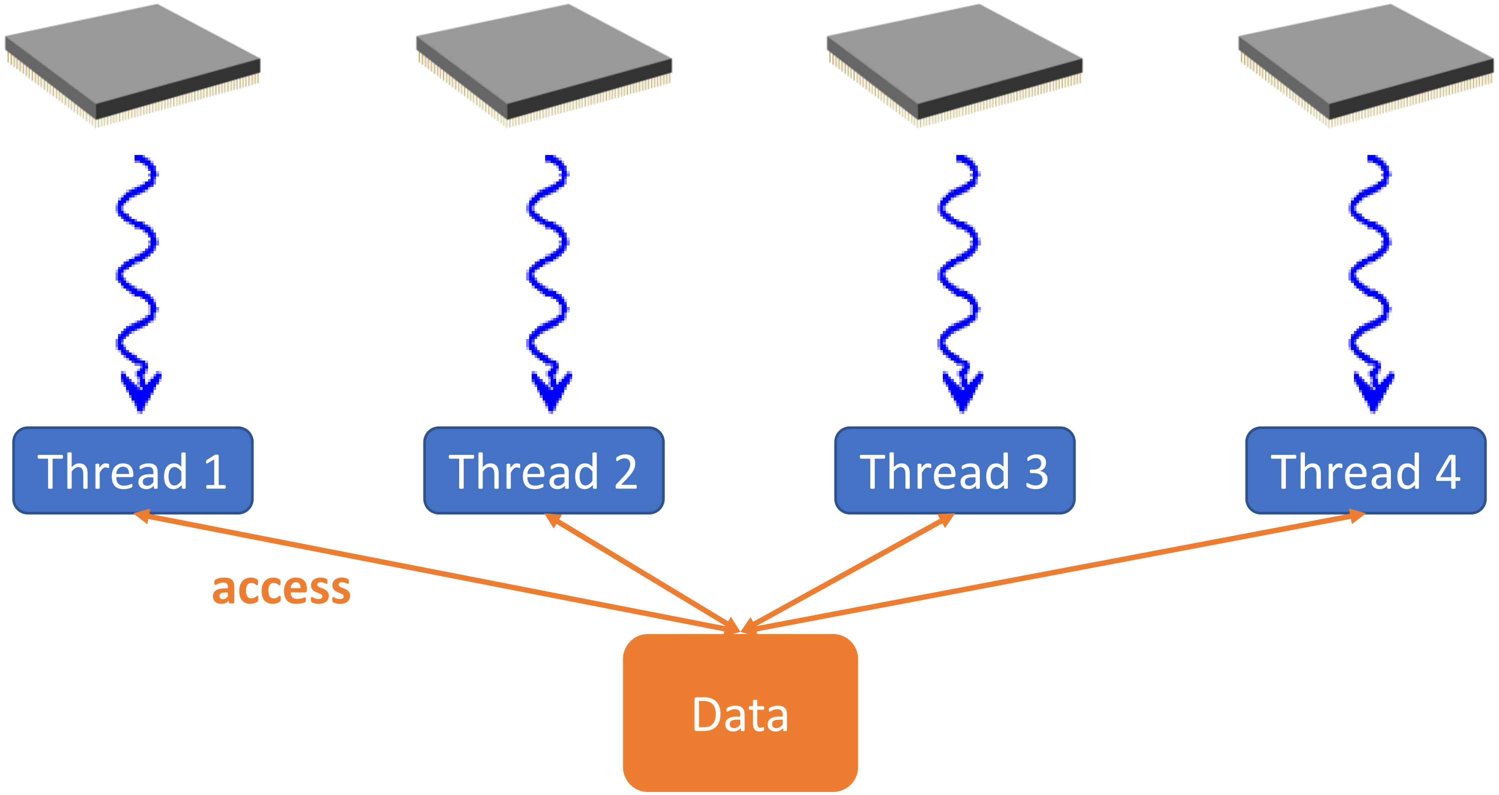
Mostly, a Thread Works on its Own Data

- Thread: a unit of code that runs on 1 CPU



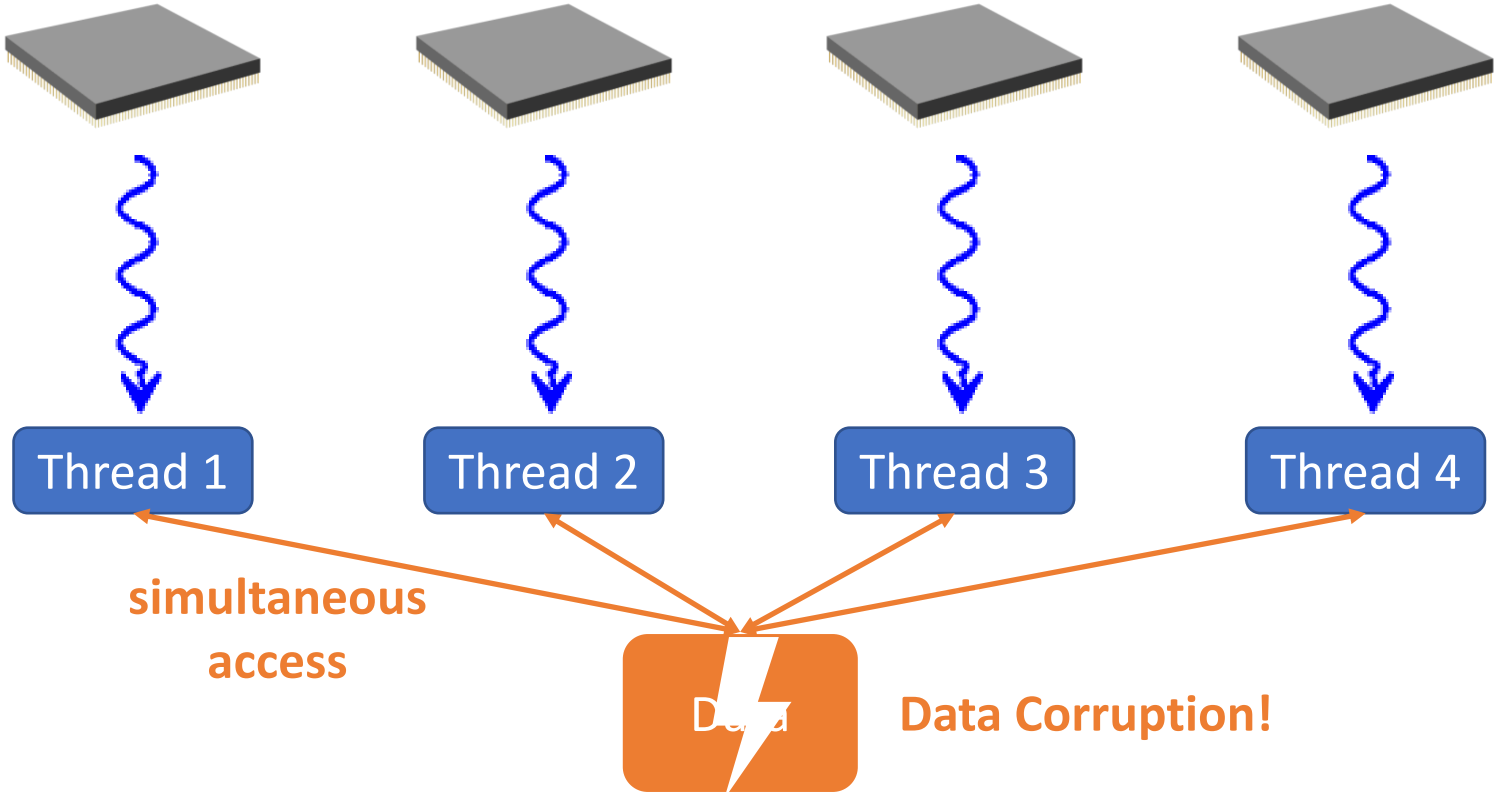
Sometimes, Threads Work on Shared Data

- Thread: a unit of code that runs on 1 CPU



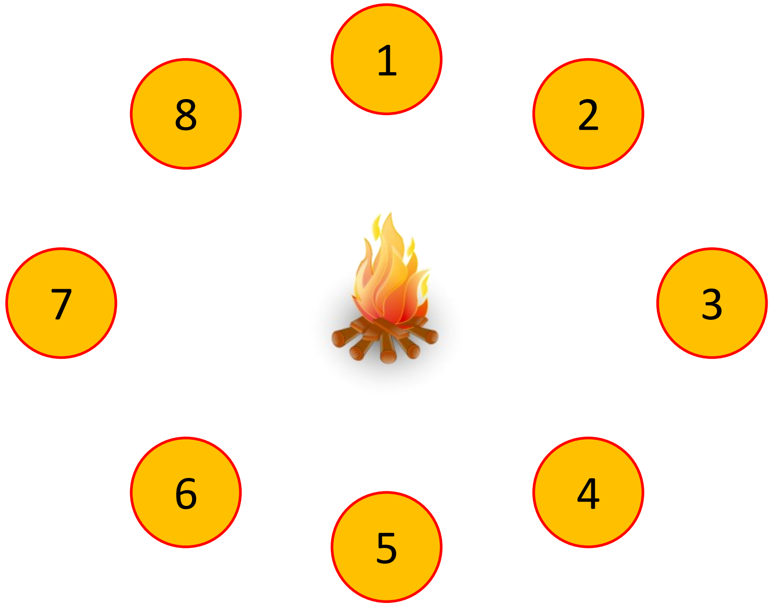
If not Careful, Data can be Corrupted!

- If threads don't take turns, and access data simultaneously



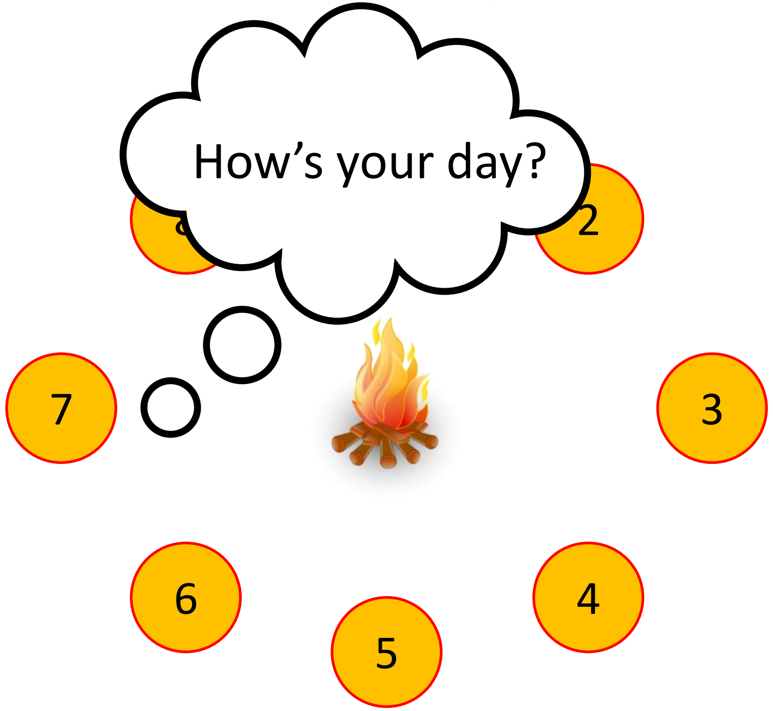
Why Data Corruption on Simultaneous Access?

- Imagine a group of village elders having a discussion around a campfire



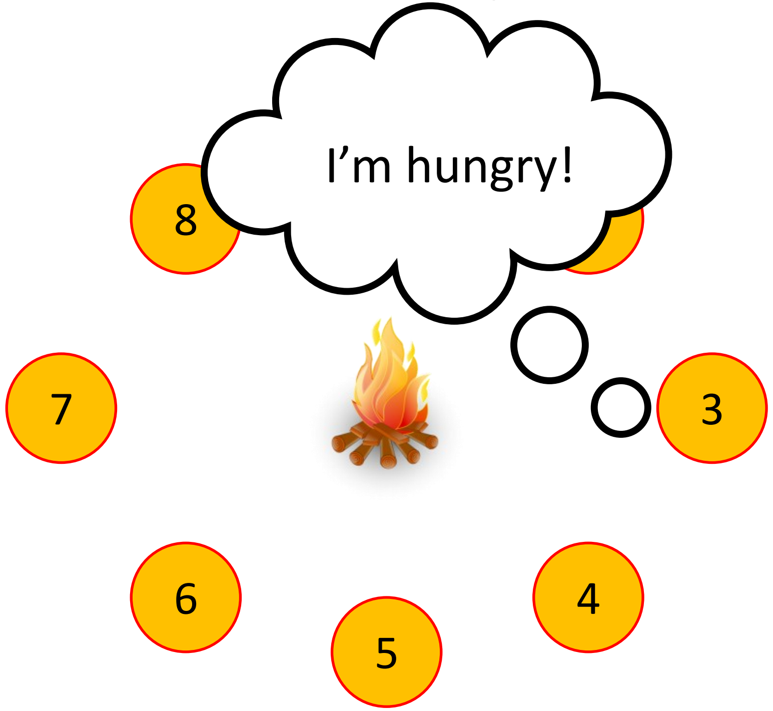
Why Data Corruption on Simultaneous Access?

- If the elders speak one at a time, they will understand each other



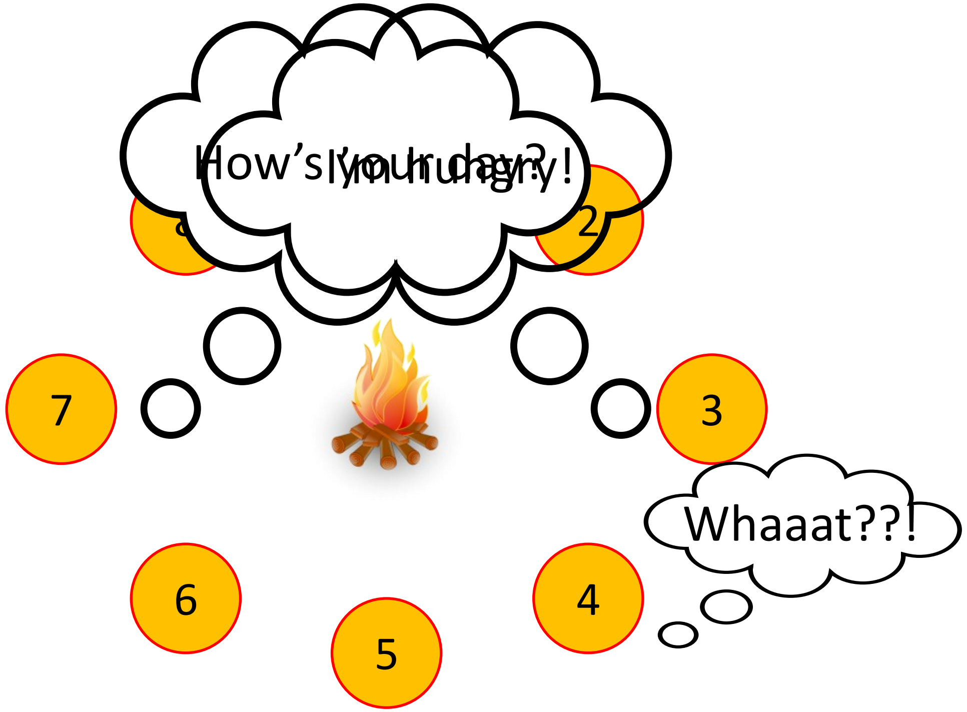
Why Data Corruption on Simultaneous Access?

- If the elders speak one at a time, they will understand each other



Why Data Corruption on Simultaneous Access?

- But if they talk at the same time, the speech becomes garbled



- Same thing happens with data. This is called a *data race*.

Worst Part: Data Races are Nondeterministic

- If lucky, meeting may adjourn with everyone taking turns speaking

How's your day?
2
7
3
6
5
4

I'm hungry!
8
7
3
6
5
4

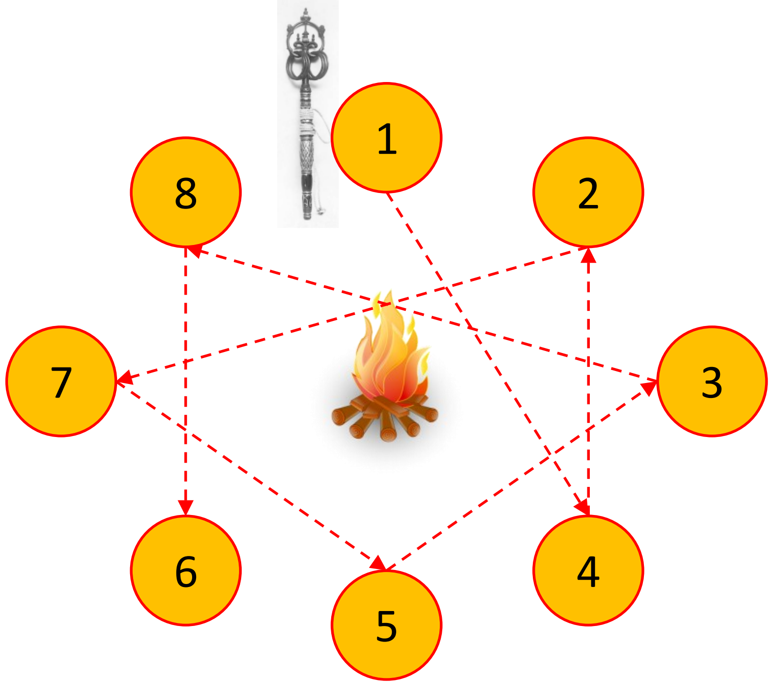
- If unlucky, two elders may speak at the same time causing the problem

How's your day?
I'm hungry!
2
7
3
6
5
4

- It all depends on timing …

Solution: Speaking Staff

- Rule: only the elder with the speaking staff shall speak
- Forces elders to take turns speaking by passing around the staff



University of Pittsburgh

Speaking Staff in Software is the *Lock*

- *Lock*: a software object that only one thread can hold at a time
- Threads take turn accessing shared data by passing around a lock
- Data races can be removed with proper use of locks

Datarace Example

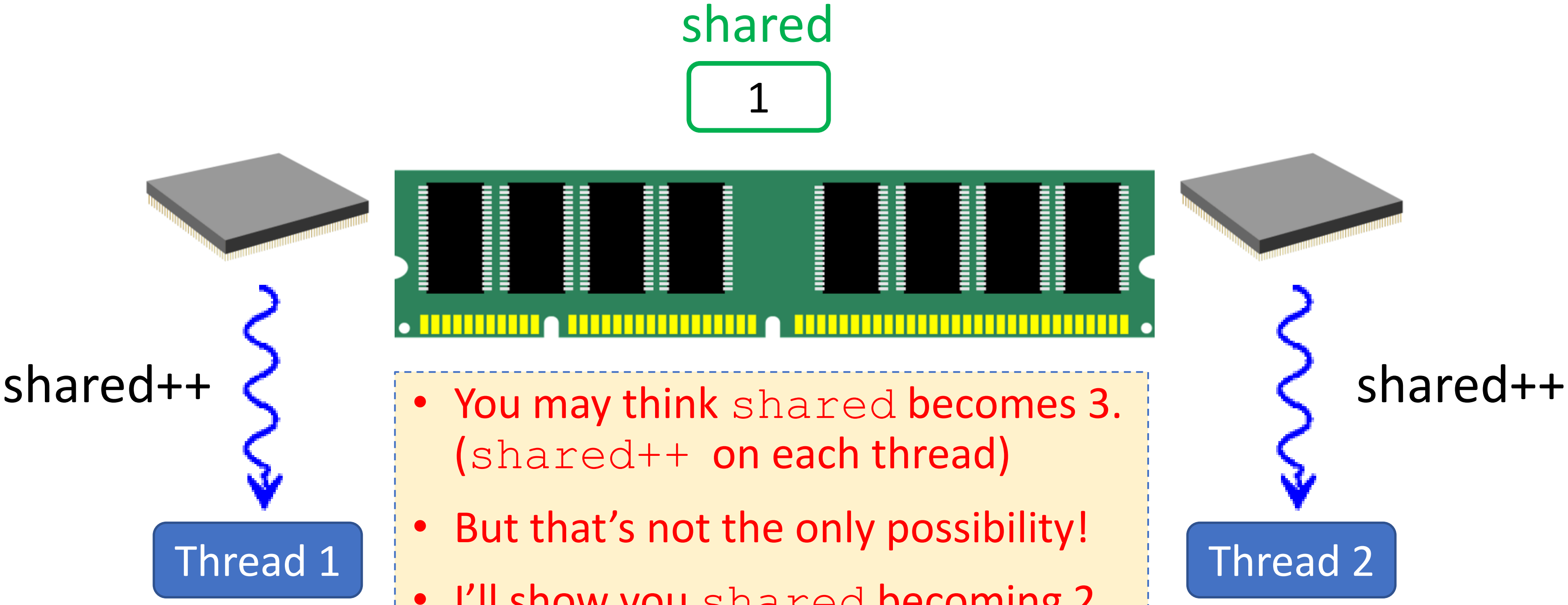
```
int shared = 0;
void *add(void *unused) {
  for(int i=0; i < 1000000; i++) { shared++; }
  return NULL;
}
int main() {
  pthread_t t;
  // Child thread starts running add
  pthread_create(&t, NULL, add, NULL);
  // Main thread starts running add
  add(NULL);
  pthread_join(t, NULL);
  printf("shared=%d\n", shared);
  return 0;
}
```

```
bash-4.2$ ./datarace
shared=1085894
bash-4.2$ ./datarace
shared=1101173
bash-4.2$ ./datarace
shared=1065494
```

- What do you expect from running this?
- Maybe shared=2000000 ?
- Due to datarace on `shared`.

Datarace: Sequence of Events

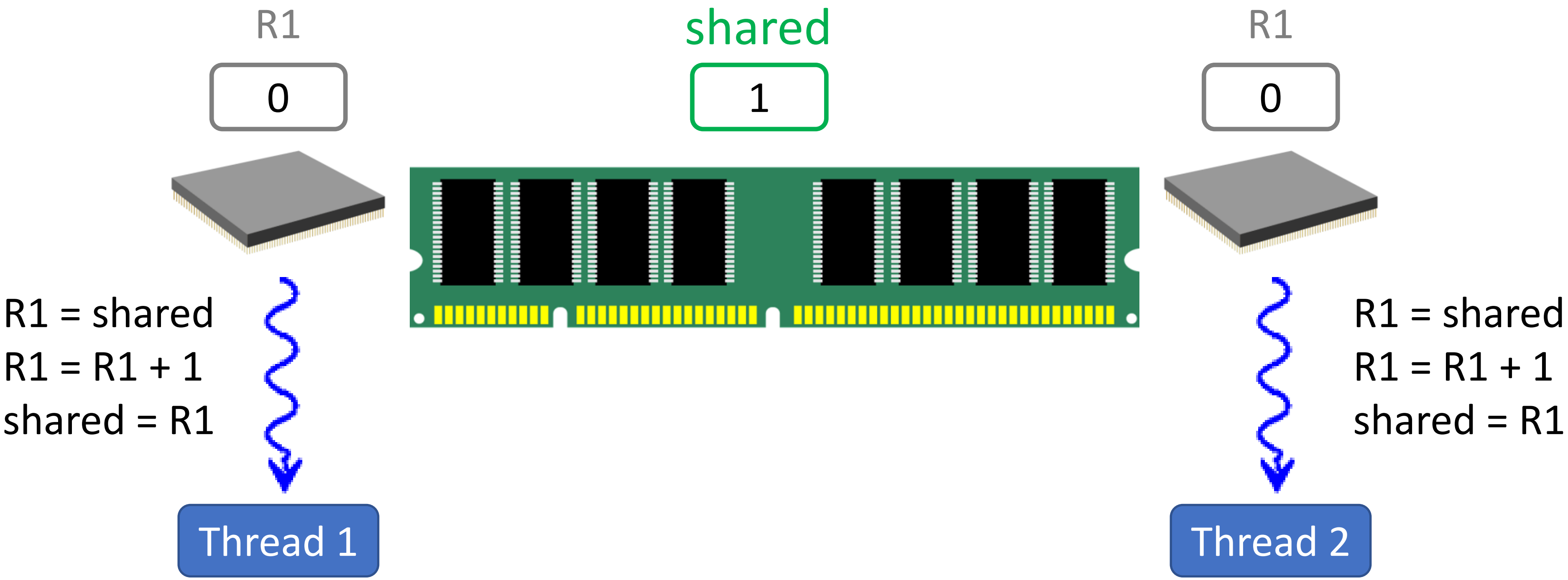
- When two threads do `shared++;` initially `shared` = 1



- You may think `shared` becomes 3. (`shared++` on each thread)
- But that’s not the only possibility!
- I’ll show you `shared` becoming 2.

Datarace: Sequence of Events

- When two threads do `shared++;` initially `shared` = 1



R1

0

shared

1

R1

0

R1 = shared
R1 = R1 + 1
shared = R1

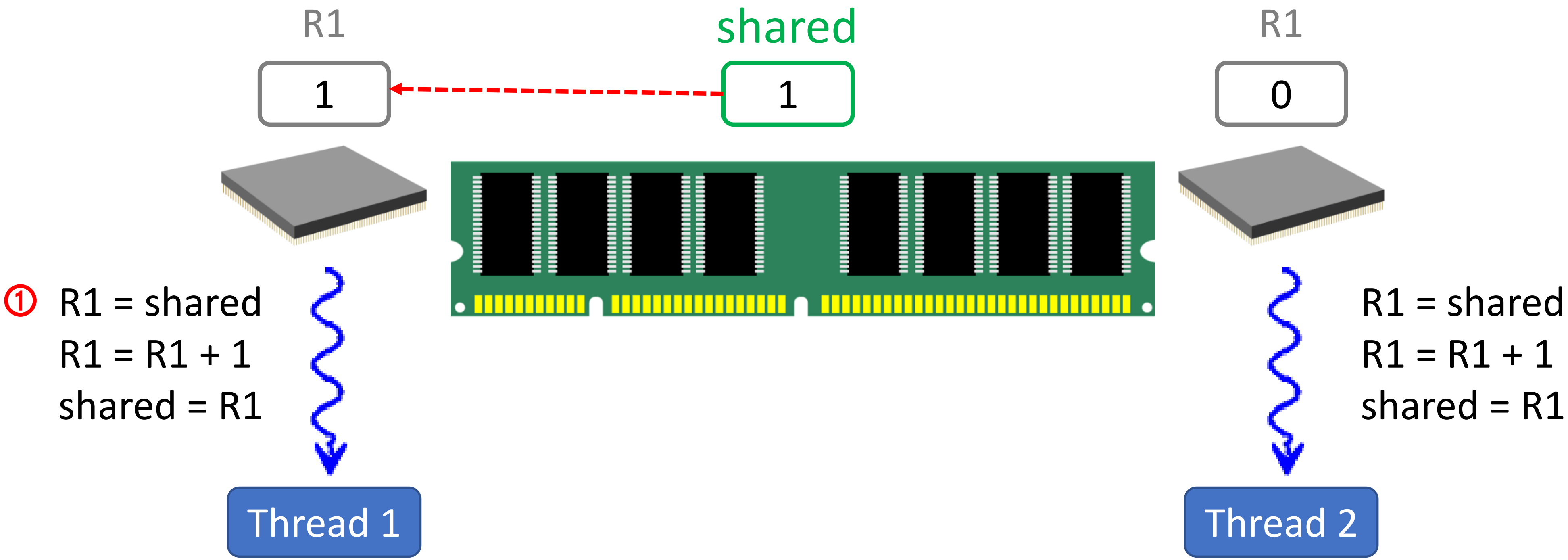
Thread 1

R1 = shared
R1 = R1 + 1
shared = R1

Thread 2

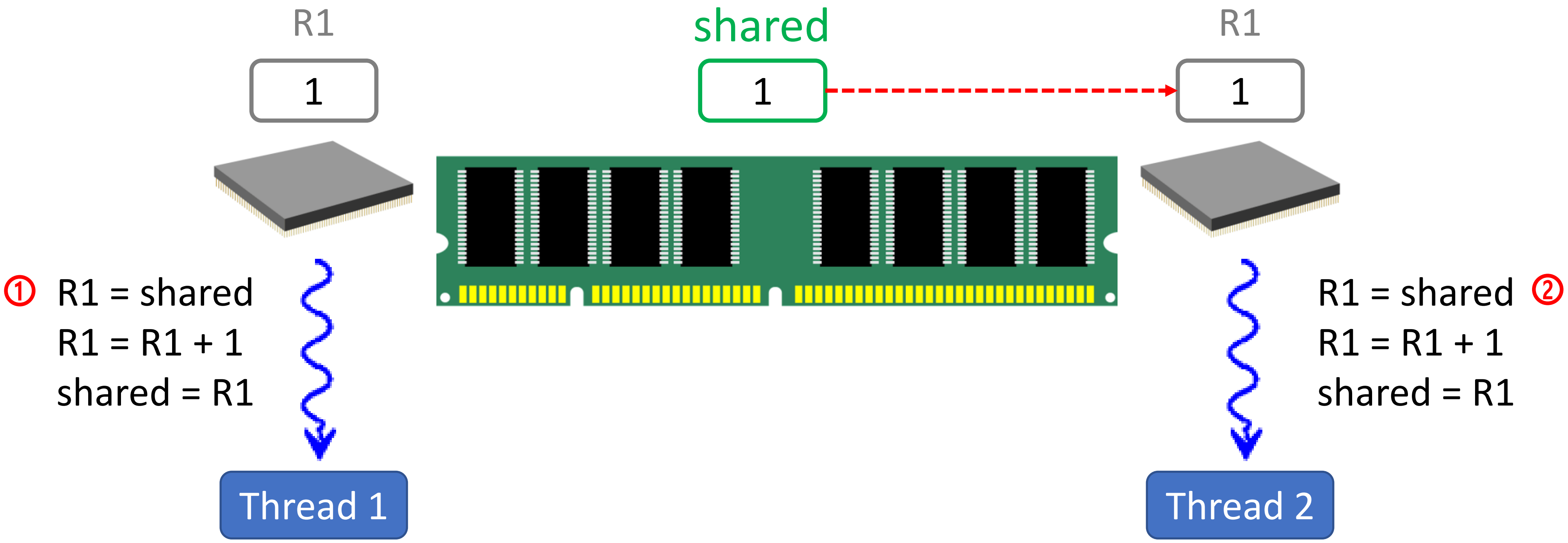
Datarace: Sequence of Events

- When two threads do `shared++;` initially `shared` = 1



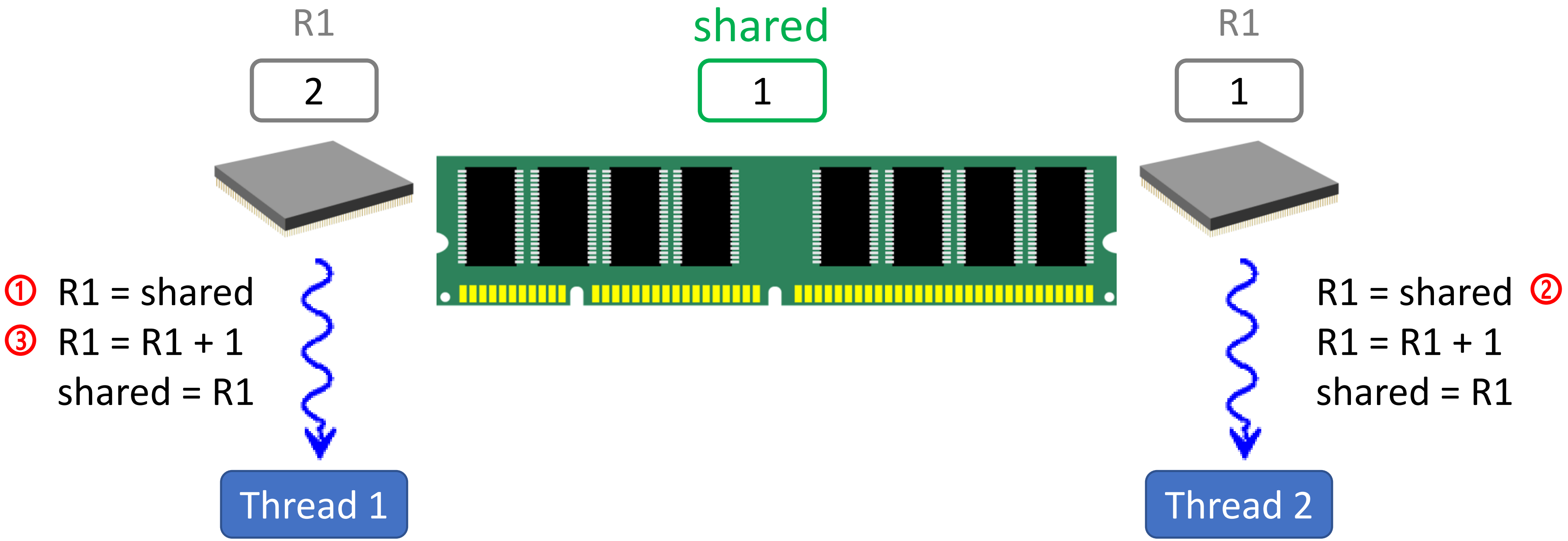
Datarace: Sequence of Events

- When two threads do `shared++;` initially `shared` = 1



Datarace: Sequence of Events

- When two threads do `shared++;` initially `shared` = 1

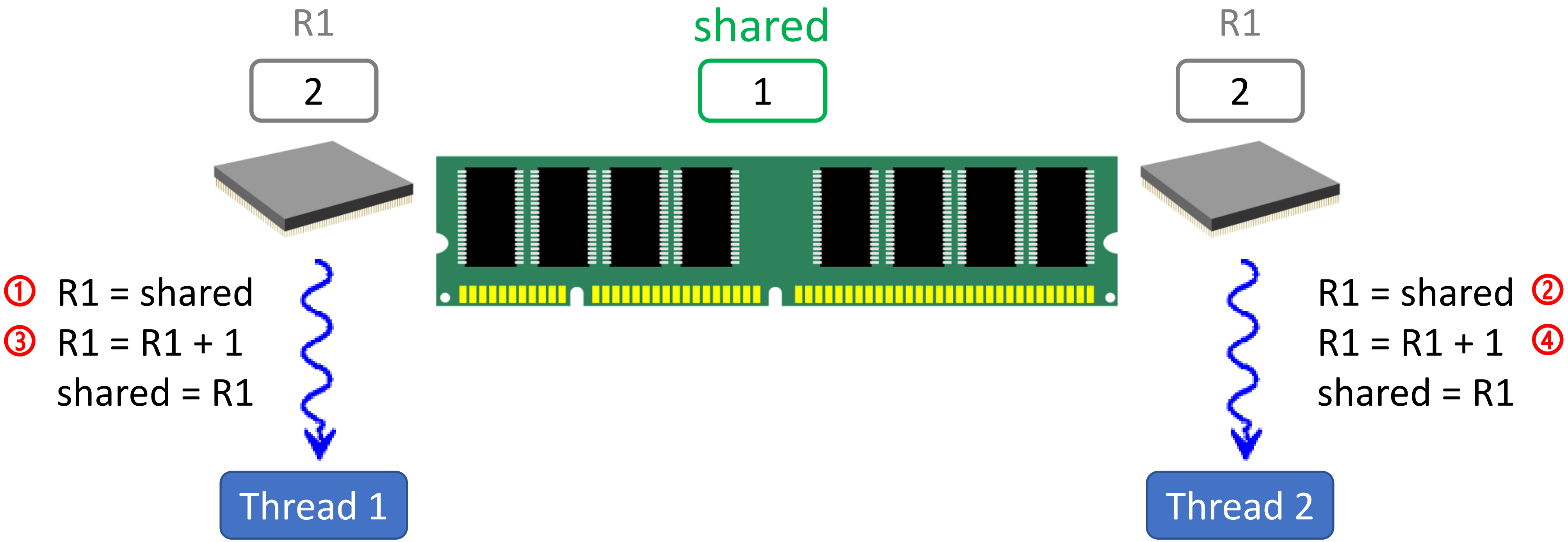


```
① R1 = shared
③ R1 = R1 + 1
  shared = R1
```

```
R1 = shared ②
R1 = R1 + 1
shared = R1
```

Datarace: Sequence of Events

- When two threads do `shared++;` initially `shared` = 1



R1

2

shared

1

R1

2

① R1 = shared
③ R1 = R1 + 1
shared = R1

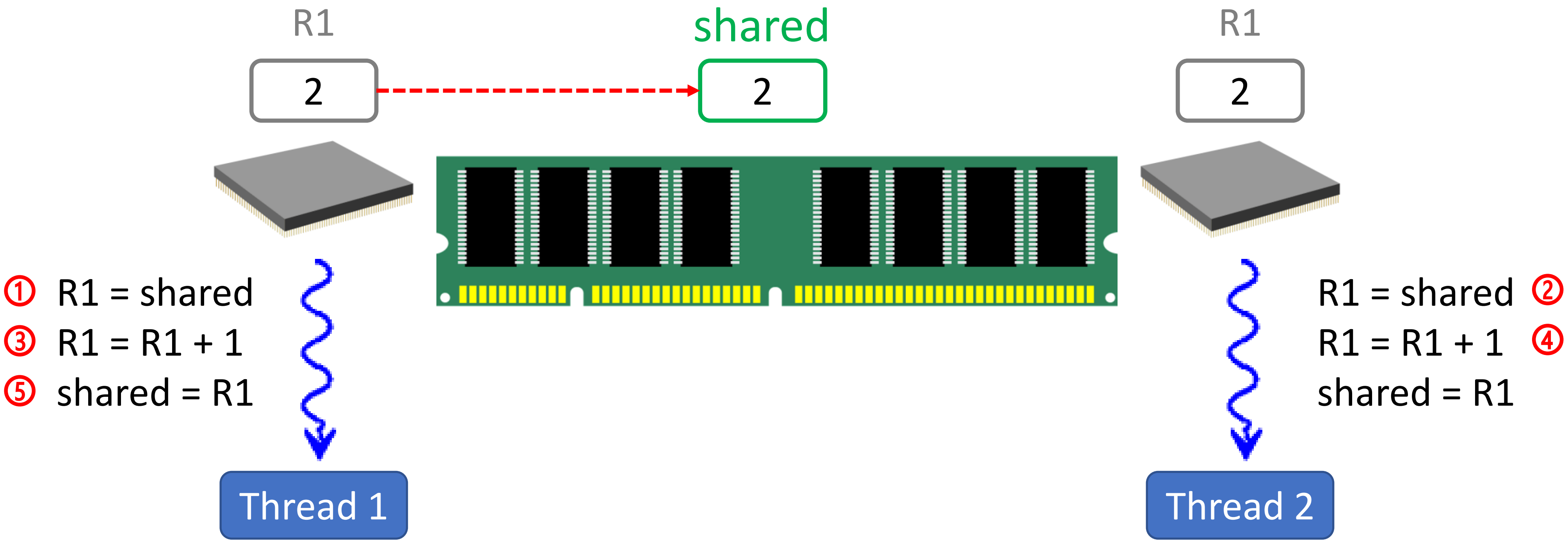
R1 = shared ②
R1 = R1 + 1 ④
shared = R1

Thread 1

Thread 2

Datarace: Sequence of Events

- When two threads do `shared++;` initially `shared` = 1

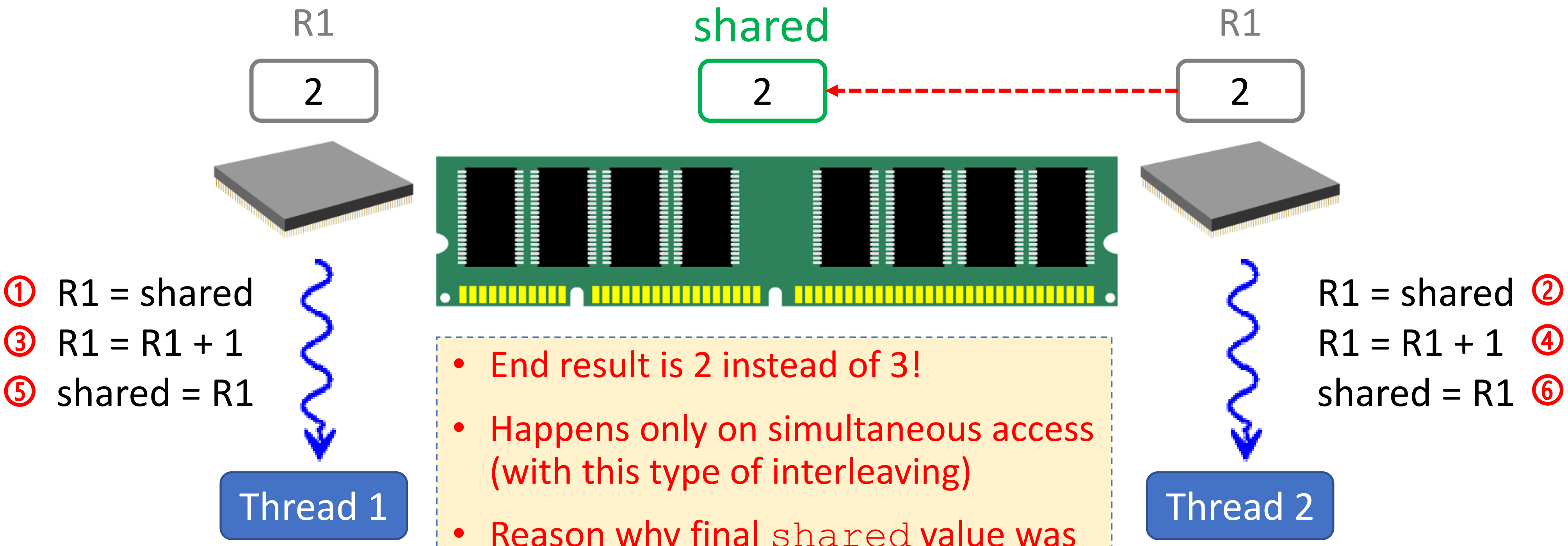


① R1 = shared
③ R1 = R1 + 1
⑤ shared = R1

R1 = shared ②
R1 = R1 + 1 ④
shared = R1

Datarace: Sequence of Events

- When two threads do `shared++;` initially `shared` = 1



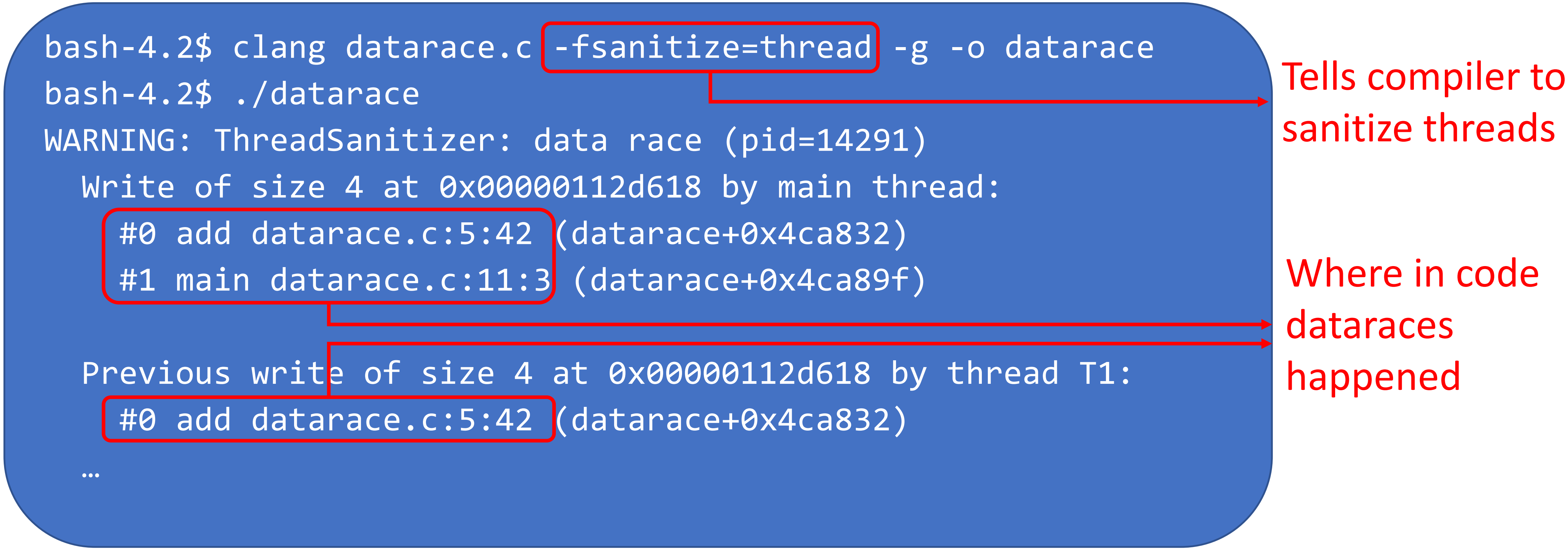
① R1 = shared
③ R1 = R1 + 1
⑤ shared = R1

R1 = shared ②
R1 = R1 + 1 ④
shared = R1 ⑥

- End result is 2 instead of 3!
- Happens only on simultaneous access (with this type of interleaving)
- Reason why final `shared` value was nondeterministic for the 3 runs

What to do? Stamp Out the Error!

- Let’s use Google Thread Sanitizer this time!



```
bash-4.2$ clang datarace.c -fsanitize=thread -g -o datarace
bash-4.2$ ./datarace
WARNING: ThreadSanitizer: data race (pid=14291)
  Write of size 4 at 0x00000112d618 by main thread:
    #0 add datarace.c:5:42 (datarace+0x4ca832)
    #1 main datarace.c:11:3 (datarace+0x4ca89f)

  Previous write of size 4 at 0x00000112d618 by thread T1:
    #0 add datarace.c:5:42 (datarace+0x4ca832)
  …
```

Tells compiler to sanitize threads

Where in code dataraces happened

Datarace Fixed

```
pthread_mutex_t lock;
int shared = 0;
void *add(void *unused) {
  for(int i=0; i < 1000000; i++) {
      pthread_mutex_lock(&lock);
      shared++;
      pthread_mutex_unlock(&lock);
  }
  return NULL;
}
int main() {
  …
}
```

```
bash-4.2$ ./datarace
shared=2000000
bash-4.2$ ./datarace
shared=2000000
bash-4.2$ ./datarace
shared=2000000
```

- Now result is deterministic
- Threads take turns accessing `shared`

Nondeterminism by Design

It’s by Design – Deal with it!

- Due to randomness by design
  - Random number generation
  - Thread interleaving

- You can’t just stamp out the nondeterminism. It’s by design.

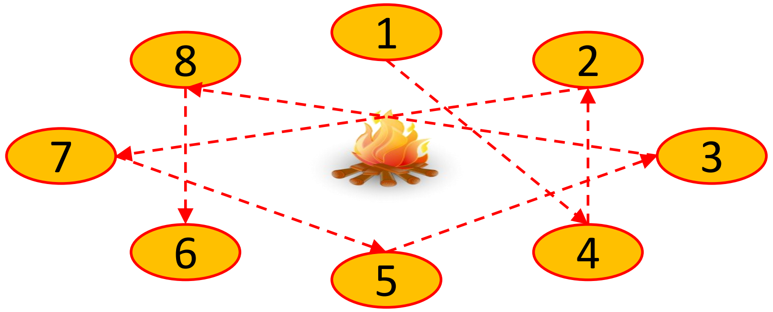
- Somehow deal with the nondeterminism such that
  - You do not get any surprise defects at the client site
  - Defects are reproducible while debugging

Outline

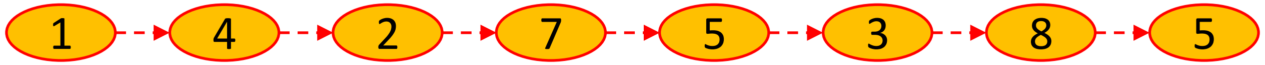
- Nondeterminism by mistake
    - Memory errors (examples / solutions)
    - Datarace errors (examples / solutions)
- Nondeterminism by design
    - Thread interleaving (examples / solutions)
    - Random number generation (examples / solutions)
- Summary

Speaking Staff doesn’t Remove All Nondeterminism

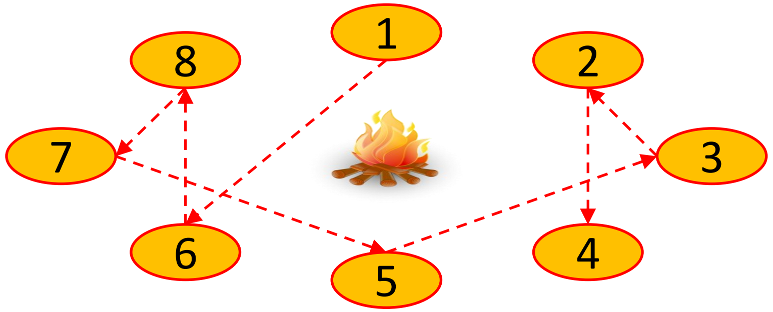
- Depending on the order the staff is passed, meeting script can change
- Order 1:



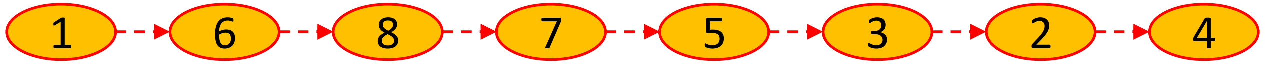
Meeting Script 1:



- Order 2:

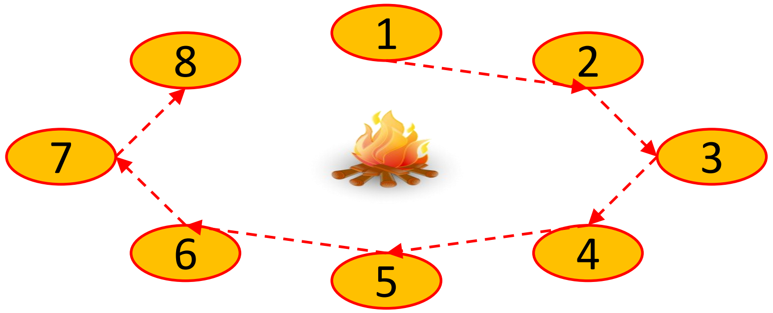


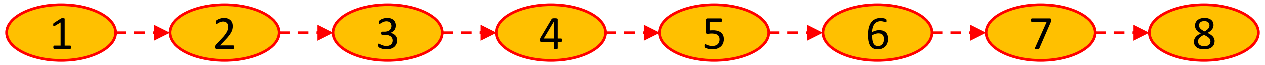
Meeting Script 2:



For Full Determinism, Must Fix Passing Order

- For example, fix staff passing order to clockwise direction
- Fixed clockwise order: Fixed meeting script

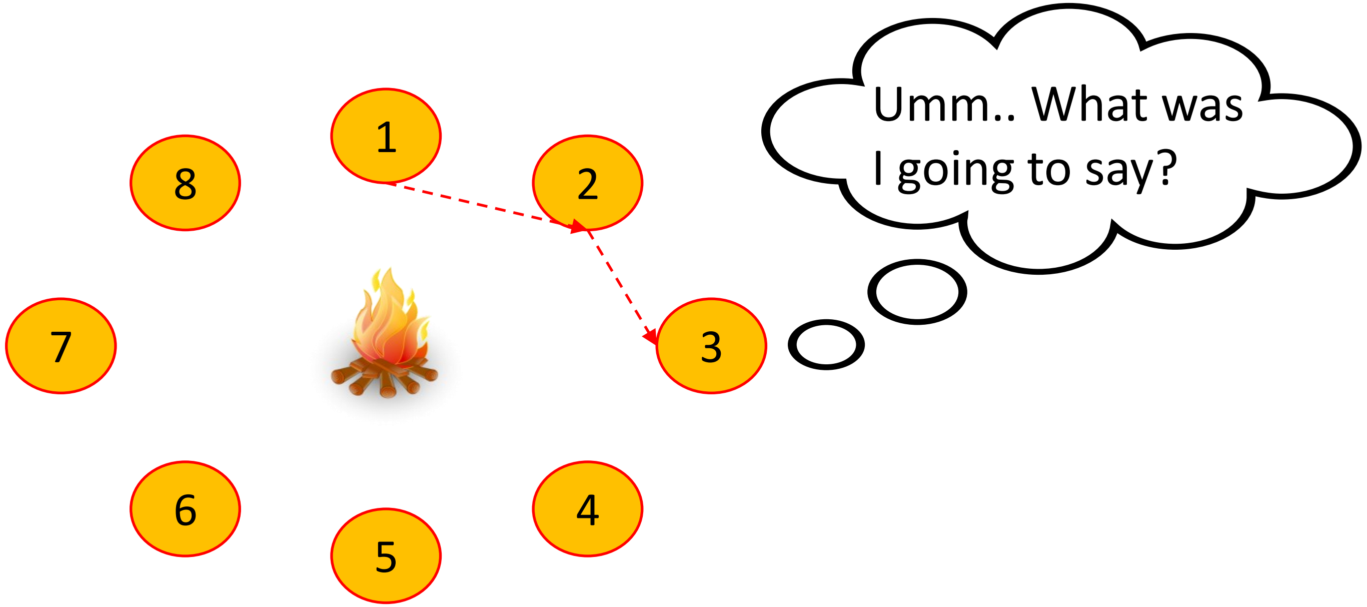




- But programmers don’t like doing this because it hurts performance

Performance Slow Down due to Fixed Order

- If an elder is not ready to speak, it can slow down the entire meeting



- Doesn't happen if staff is passed nondeterministically on demand

→ Many programs don't constrain thread interleaving for this reason

Nondeterministic Interleaving Example

```
class Interleaving implements Runnable {
  public static String script = "";
  public void run() {
    synchronized(this) {
      script += Thread.currentThread().getName() + " ";
    }
    synchronized(this) {
      script += Thread.currentThread().getName() + " ";
    }
  }
  public static void main(String[] args) throws InterruptedException {
    Interleaving m = new Interleaving();
    Thread t = new Thread(m);
    t.start();  // Child thread does run()
    m.run();    // Main thread does run()
    t.join();
    System.out.println(script);
  }
}
```

Java version of a lock, so no datarace.

But still nondeterministic due to interleaving.

- Main thread appends “main” twice
- Child thread appends “Thread-1” twice
- What are all the possible outputs?

Nondeterministic Interleaving Output

- 6 different possible outcomes!

```
bash-4.2$ java Interleaving       bash-4.2$ java Interleaving
main main Thread-1 Thread-1       Thread-1 main main Thread-1

bash-4.2$ java Interleaving       bash-4.2$ java Interleaving
main Thread-1 main Thread-1       Thread-1 main Thread-1 main

bash-4.2$ java Interleaving       bash-4.2$ java Interleaving
main Thread-1 Thread-1 main       Thread-1 Thread-1 main main
```

- All could be correct – if you don’t care about the ordering. Or not.

Nondeterministic Interleaving is Problematic

- A defect may show up only on a particular interleaving
- No guarantee which interleaving will be chosen at runtime
- So testing becomes unreliable and unreproducible
- What to do? We'll get to it soon, but let's first talk about …

Outline

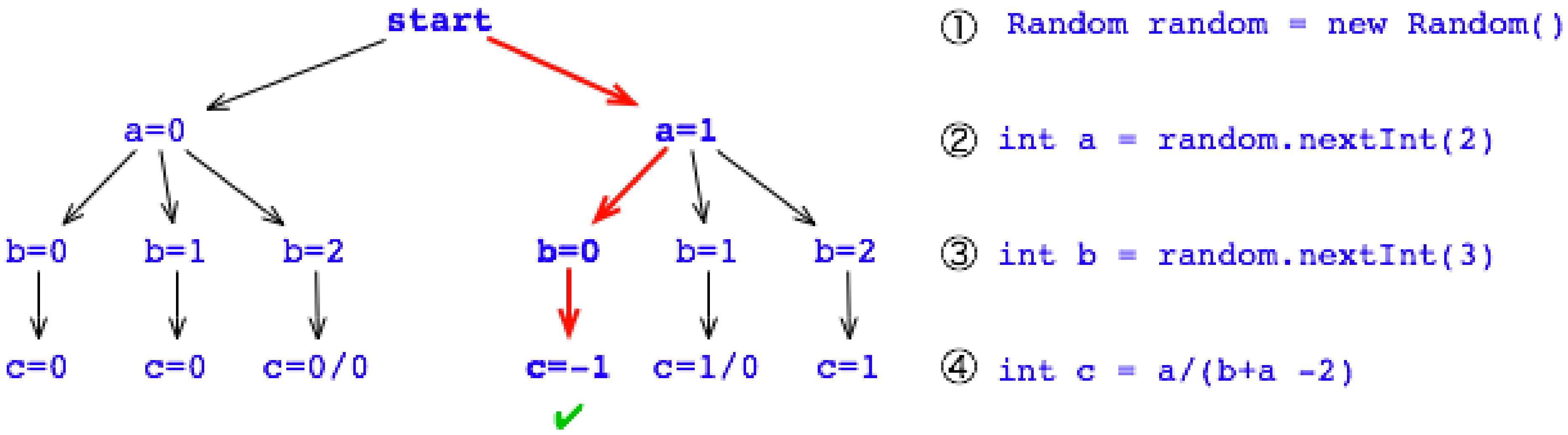
- Nondeterminism by mistake
  - Memory errors (examples / solutions)
  - Datarace errors (examples / solutions)
- Nondeterminism by design
  - Thread interleaving (examples / solutions)
  - Random number generation (examples / solutions)
- Summary

Random Number Generation Example

Given this code:

```
int a = random.nextInt(2);
int b = random.nextInt(3);
int c = a/(b+a -2);
```

If unlucky, paths with defects will not be covered during testing and bug may never be found!



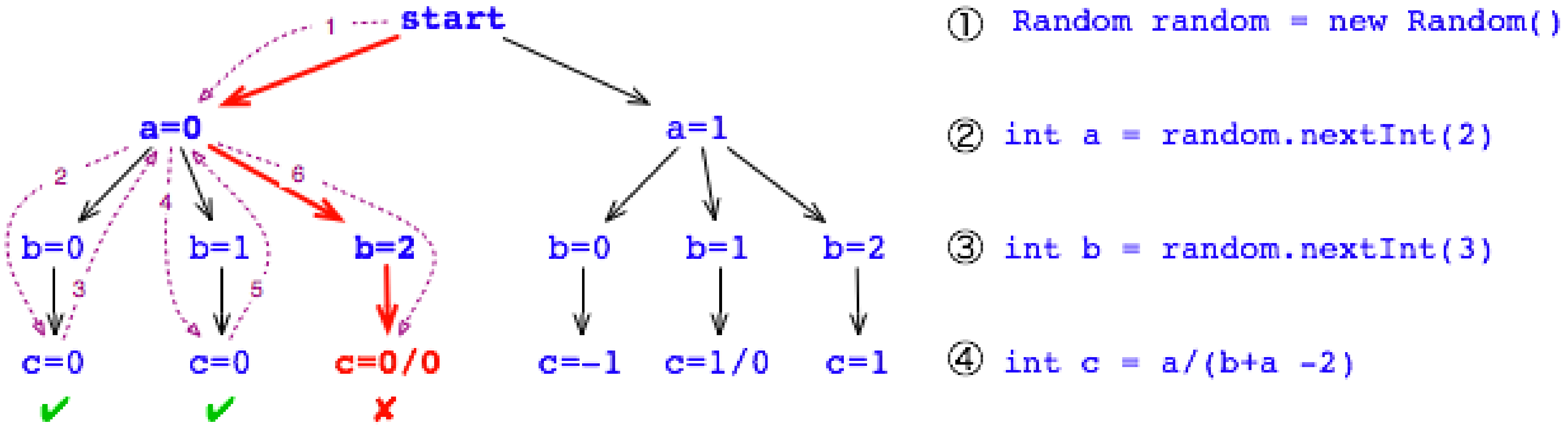
```
Random random = new Random()
int a = random.nextInt(2)
int b = random.nextInt(3)
int c = a/(b+a -2)
```

What to do? Deal with it!

Given this code:

```
int a = random.nextInt(2);
int b = random.nextInt(3);
int c = a/(b+a -2);
```

Exhaustively search through all possible paths to find the defect!



```
① Random random = new Random()
② int a = random.nextInt(2)
③ int b = random.nextInt(3)
④ int c = a/(b+a -2)
```

Java Path Finder (JPF)

- Model checker developed by NASA to verify mission critical code
- Exhaustively searches through all possible states of a program
  - Enumerates all possible values from random number generators
  - Enumerates all possible interleavings between threads
- We will learn to use this towards the end of the semester

JPF on Random Number Generation

```
int a = random.nextInt(2);
int b = random.nextInt(3);
int c = a/(b+a -2);
```

Not shown, but also generates a "trace" of random values chosen

```
-bash-4.2$ ./runJPF.sh Random.jpf
JavaPathfinder core system v8.0 (C) 2005-2014 United States Government.
…
====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: division by zero
        at Rand.main(Rand.java:34)
…
```

Where in code exception happened

JPF on Thread Interleaving

```
-bash-4.2$ ./runJPF.sh Interleaving.jpf
JavaPathfinder core system v8.0 (C) 2005-2014 United States
Government.
…
====================================================== search started
main main Thread-1 Thread-1
main Thread-1 main Thread-1
main Thread-1 Thread-1 main
Thread-1 main main Thread-1
Thread-1 main Thread-1 main
Thread-1 Thread-1 main main
```

Able to explore all interleavings and generate all possible outputs!

Summary

Summary

- We learned there are two types of nondeterminism
  - Nondeterminism by mistake – stamp it out!
    - Memory errors
    - Datarace errors
  - Nondeterminism by design – deal with it!
    - Random number generation
    - Thread interleaving
- We also learned three tools that can help you
  - Google Address Sanitizer
  - Google Thread Sanitizer
  - NASA Java Path Finder

Open Source Resources

- Google Address Sanitizer: https://github.com/google/sanitizers/wiki/AddressSanitizer
- Google Thread Sanitizer: https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual
- NASA Java Path Finder: https://github.com/javapathfinder/jpf-core/wiki https://github.com/javapathfinder/jpf-core/wiki/GSoC-2020-Project-Ideas

References

- Konstantin Serebryany et al. “AddressSanitizer: A Fast Address Sanity Checker”. USENIX, 2012: https://research.google/pubs/pub37752/

- Konstantin Serebryany et al. “ThreadSanitizer – data race detection in practice”. Workshop on Binary Instrumentation and Applications (WBIA), 2009: https://research.google/pubs/pub35604/

- Ranjit Jhala and Rupak Majumdar. “Software model checking”. ACM Computing Surveys, 2009: https://people.mpi-sws.org/~rupak/Papers/SoftwareModelChecking.pdf

- 10th Competition on Software Verification (SV-COMP), 2021: https://sv-comp.sosy-lab.org/2021/results/results-verified/

Questions?