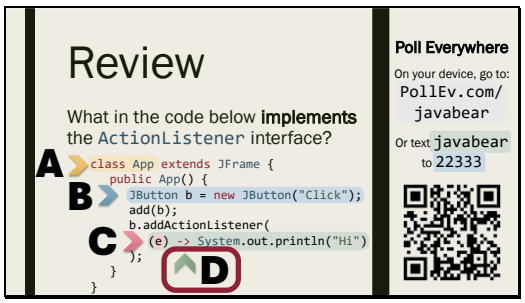
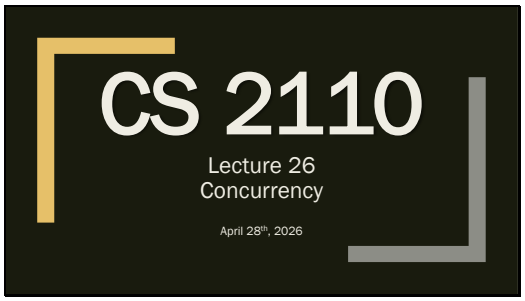
Slide 1

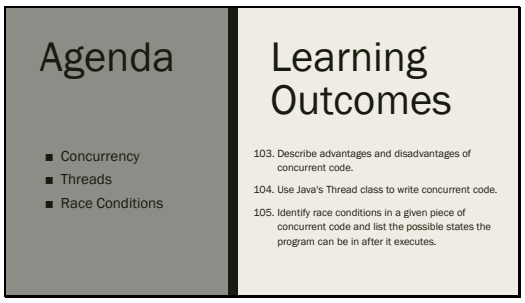



The `ActionListener` interface is a functional interface which is used as part of the observer pattern. The implementation of it is passed to `addActionListener()`. Recall that a lambda expression is itself an object that will implement the appropriate functional interface, so the entire lambda expression is the answer.

Slide 2

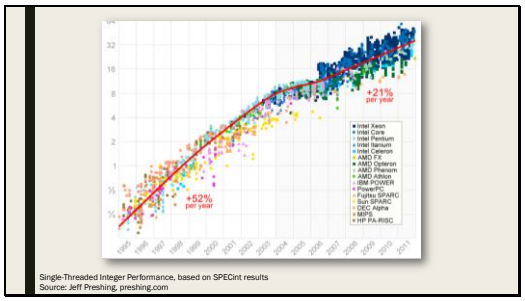



Slide 3



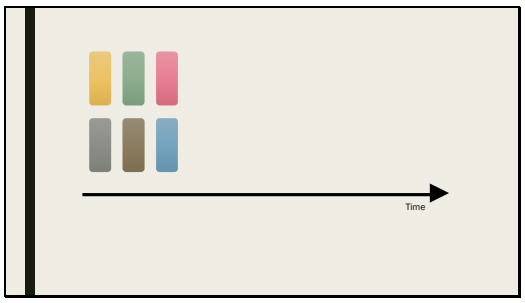
Slide 4


Slide 5




For the longest time, computers got twice as fast basically every two years. But unfortunately, around the 2000's, this stopped happening.

Slide 6



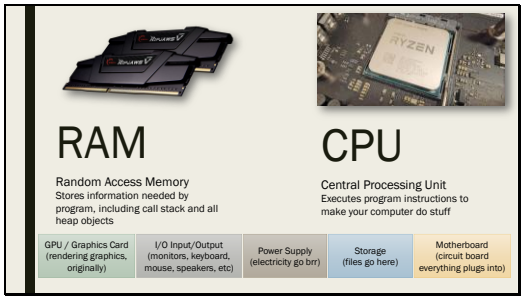

This is unfortunate. It used to be that if you had code that was slow, just wait two years and it would be twice as fast. But as single core performance started plateauing, it became increasingly difficult to squeeze more performance out of our computers. As such, another idea to improve the speed of computation is to run multiple parts of it at the same time.

Slide 7

# Parallelism

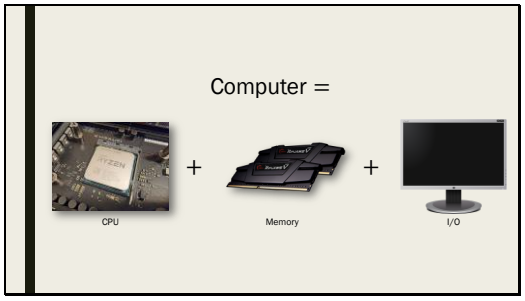
Carrying out multiple tasks
at the exact same time

Slide 8



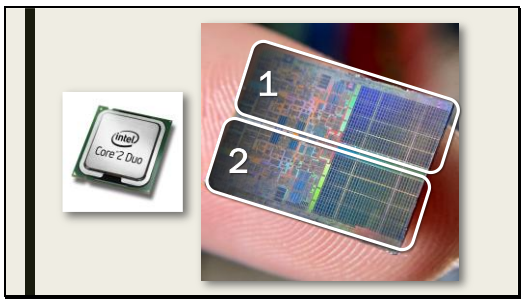

Quick primer about the parts of your computer: the CPU is the processor and executes instructions, while RAM is the memory where all your variables (call stack + heap) are stored.

Slide 9




There's some other stuff, but basically a computer is something that computes data, stores data, and interfaces with humans

Slide 10




To get parallelism, we need to look at the CPU. CPUs in the 2000s, like the Core 2 Duo, just straight up had two whole cores on the chip.

Slide 11

```
task1();
task2();
task3();
task4();
task5();
task6();
```

NOT real Java code

So if we imagine writing code that can take advantage of this fact, let's write some pseudocode.

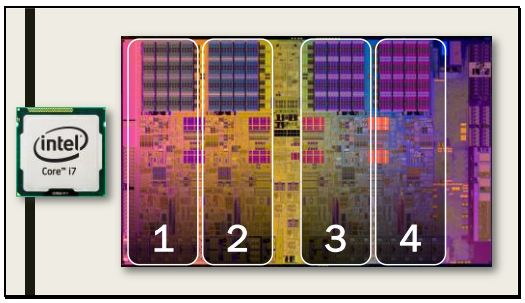
Slide 12

```
run(core1) {
    task1();
    task2();
    task3();
}
run(core2) {
    task4();
    task5();
    task6();
}
```

NOT real Java code

One proposal might be for the programmer to just write out what tasks go on which core.

Slide 13




But then along comes the 2010's and new chips now have 4 or more cores!

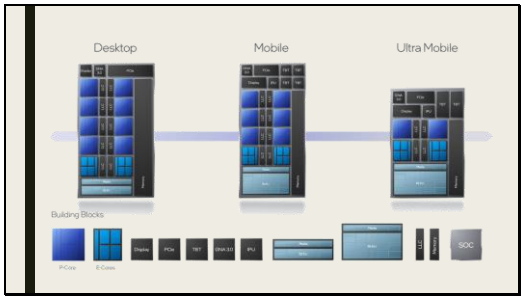
Slide 14

```
run(core1) {
    task1();
    task2();
}
run(core2) {
    task4();
    task5();
}
run(core3) {
    task3();
}
run(core4) {
    task6();
}
```

NOT real Java code

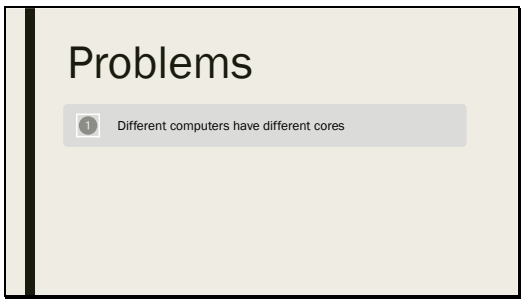
So now what? We could imagine having to write new code that now takes advantage of all four cores. But this isn't very sustainable as CPUs keep gaining more cores. What's worse, now our code no longer runs on an older CPU that doesn't have 4 cores.

Slide 15



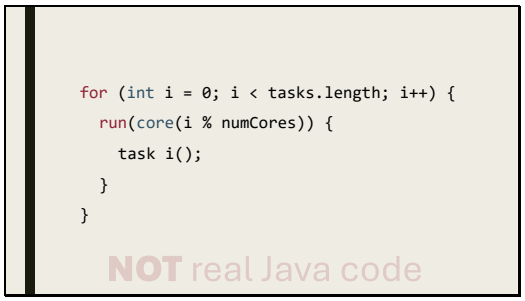

And this is assuming all the cores are the same speed, which isn't actually true on a lot of modern devices (eg: Apple M chips, Intel 12$^{th}$ gen and later, any phone chip, etc.)

Slide 16



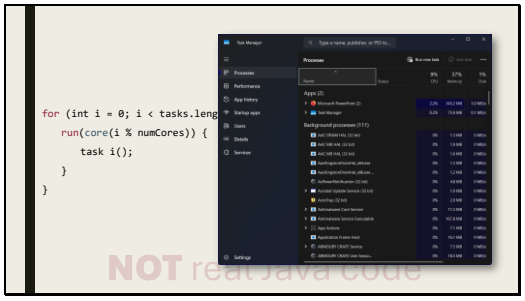

So the first problem, right off the bat, is that different computers have different cores, both in number and type

Slide 17



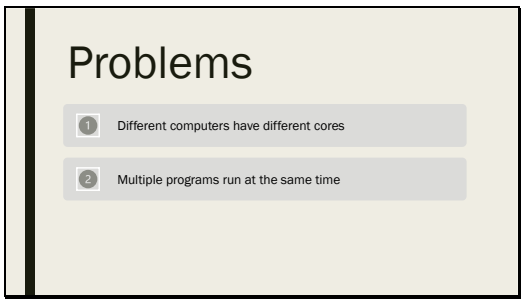

But let's pretend somehow that's not a problem. Maybe we write some code to load balance automatically.

Slide 18




Unfortunately, there are other programs also running on your computer. And they would all like CPU time too.

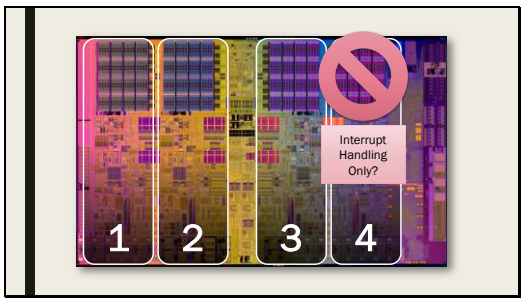
Slide 19




Slide 20

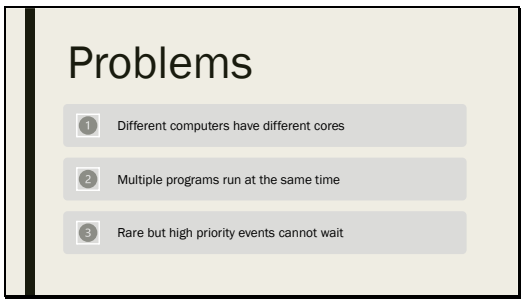

But even if we could somehow work around that problem, we have an even bigger issue. Sometimes the user likes to, you know, use their computer. And when they move the mouse or press something on the keyboard, they like it when the computer is able to respond. Unfortunately, if all of the programs are super efficiently using the entire CPU, nothing is available to process those inputs.

Slide 21

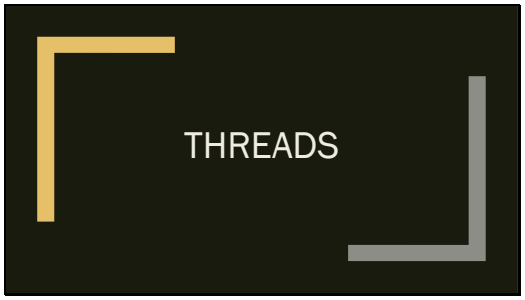



But at the same time, we don't want to reserve an entire CPU core to only handle these inputs, because they're relatively rare in the grand scheme of things. We'd be wasting a lot of CPU potential if the core just sat around idle whenever the user isn't actively moving their mouse.

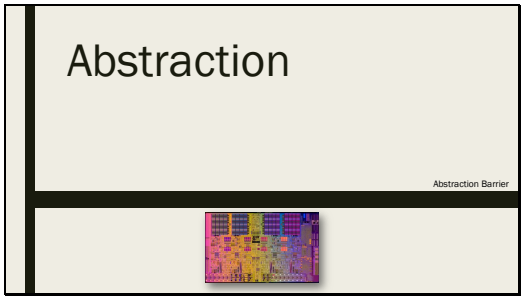
Slide 22




Slide 23

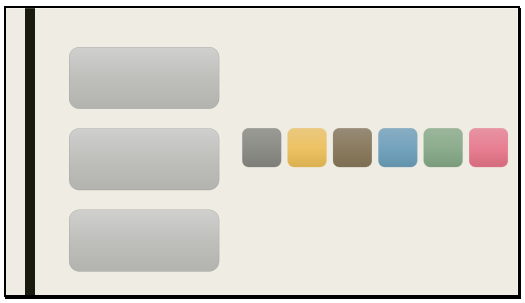



Slide 24



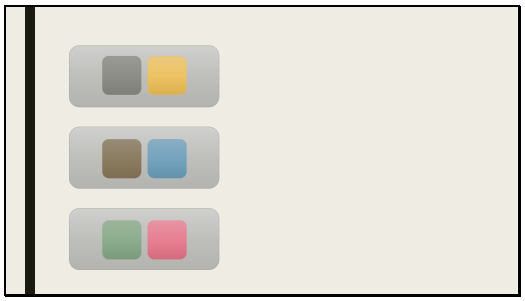

As always, the solution to every problem in CS is another abstraction, and that's what is going to save us here. Instead of directly writing code for each CPU core, we abstract the cores away, behind the abstraction barrier.

Slide 25



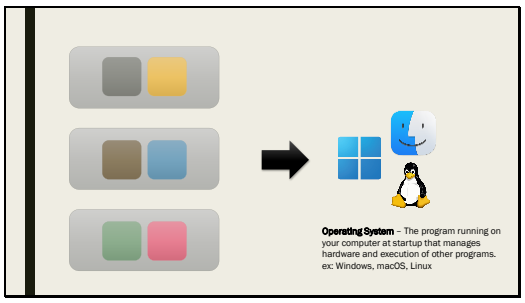
Instead, given some tasks, we can pretend we have as many cores as we naturally would need.

Slide 26



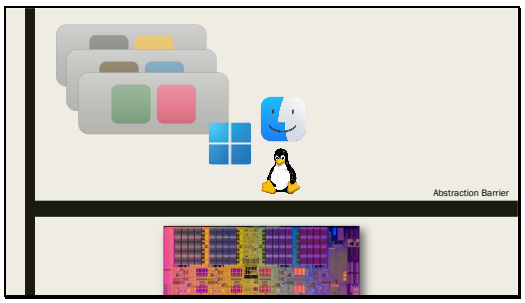
We can then assign our tasks to these imaginary cores based on what the natural split between tasks is, and not the actual hardware of the computer.

Slide 27



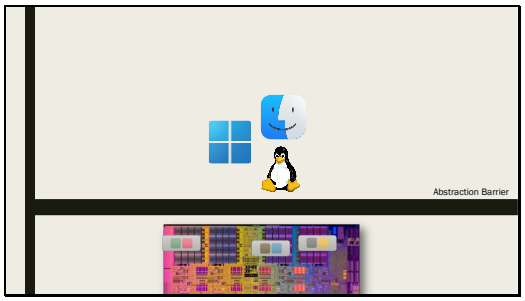

Then, we hand off these imaginary cores to the operating system.

Slide 28



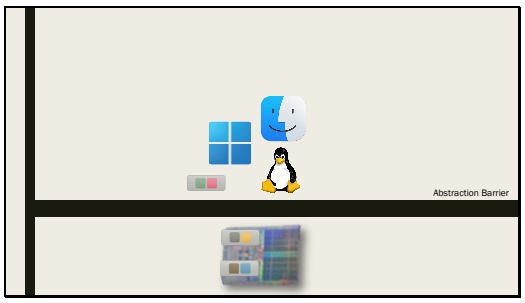

The OS can then look beneath the abstraction barrier and see how many actual cores exist.

Slide 29



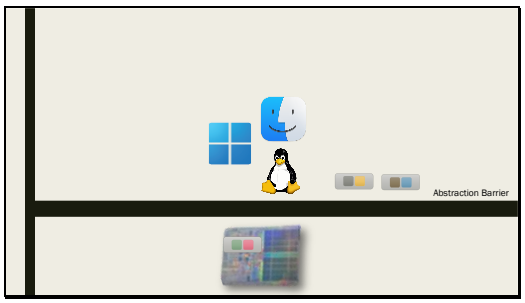

In this case, there's more cores then we need, so the CPU can just map each task to a core.

Slide 30



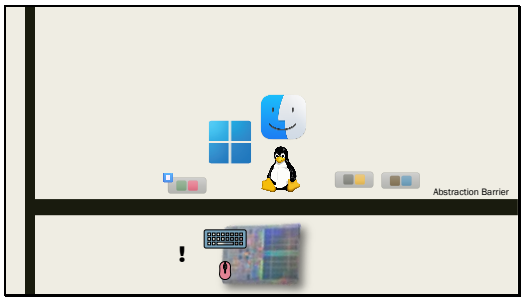

But if we have an older CPU with less cores, the OS may instead choose to first assign some of the tasks to the cores and have the other tasks wait…

Slide 31



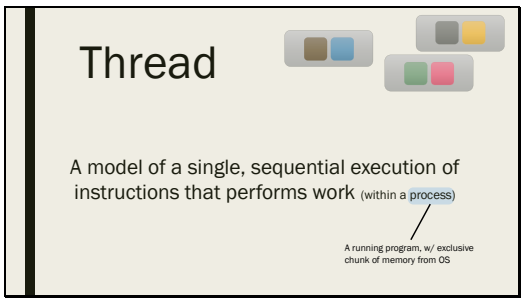

… and then run that other task later.

Slide 32



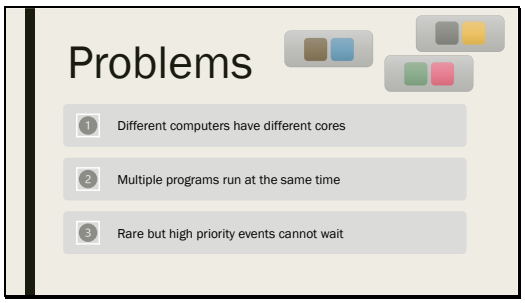

And if a high priority interruption occurs, the OS can pause (or preempt) our task and pull us off the CPU, letting the higher priority thing go first. We'll be able to resume later once the CPU is free.

Slide 33



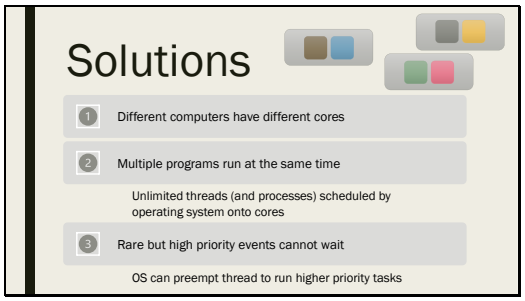

As you may have guessed, these imaginary cores are what we call threads. They are an abstraction we use to cover up the complexity of different hardware setups, and they each model a set of sequential steps. Note that all threads within a program (or "process") share the same memory. Different processes do not share memory.

Slide 34



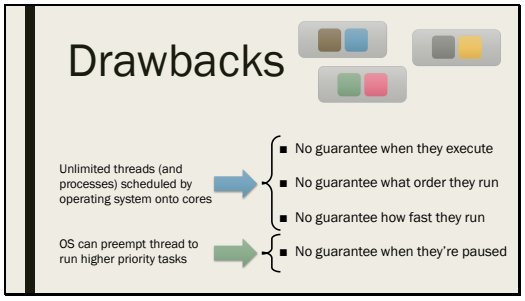

Returning to our original list of problems, notice that threading solves all of them.

Slide 35



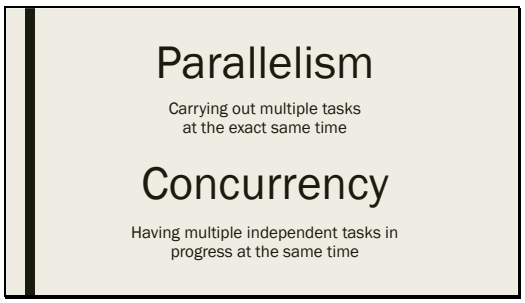

For problems 1 and 2, we no longer need to worry about how many cores or programs there are, because we can write code with as many threads as we need. For the high priority events, the OS can handle pausing one of our threads to run the higher priority one first, ensuring we resume later.

Slide 36



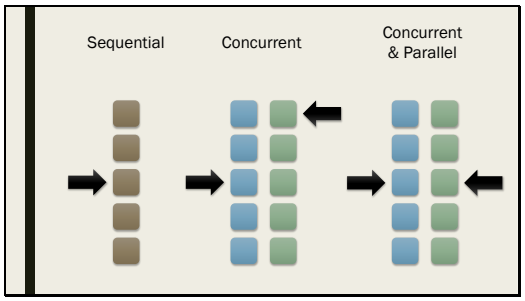

This abstraction solves all of our problems, but it also means we have given up a lot of control to the operating system. We no longer get to control when our threads run, what order or speed they run, or when they are paused. This tradeoff is worth it to be able to harness the power of parallelism, but we need to keep these in mind, as they'll come back later.

Slide 37




A note on terminology, concurrency is the more general term that describes having multiple threads in general. Parallelism is when they actually run at the same time (you could alternatively imagine, on a CPU with only one core, the OS only running one thread at a time but switching back and forth between them to make progress on both).

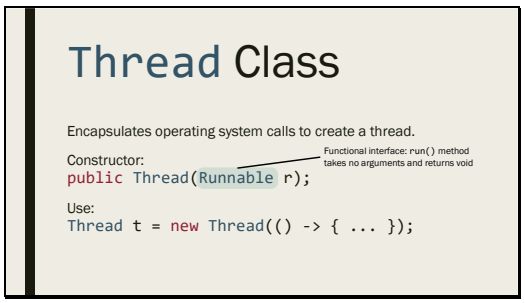
Slide 38




In other words, having multiple threads makes code concurrent; if the OS schedules multiple threads to run at the exact same time, then it is also parallel.

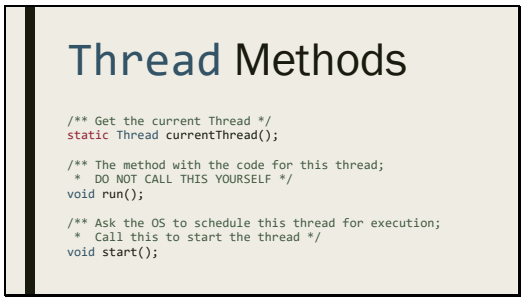
Slide 39

Slide 40



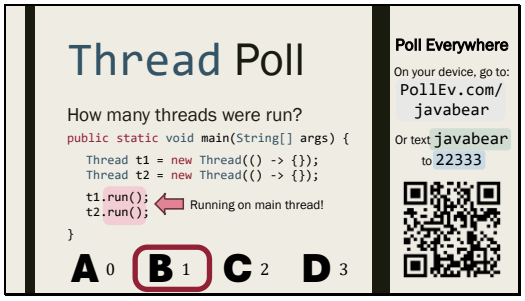

The `Thread` class encapsulates all the details of communicating with the operating system to request a new thread. As a client, just pass a lambda (or other `Runnable` implementation) to the constructor.

Slide 41




The most important method to call is `start()`, which requests the new thread from the operating system.

Slide 42

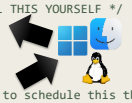



Warning: Calling `run()` does NOT create a new thread! The `run()` method literally only contains the code inside the `Runnable` lambda. Calling it directly is the same as just running the code normally. Only the `start()` method creates a new thread from the operating system.

Slide 43

## Thread Methods

```
/** The method with the code for this thread;
 *  DO NOT CALL THIS YOURSELF */
void run();
```



```
/** Ask the OS to schedule this thread for execution;
 *  Call this to start the thread */
void start();
```

You

In other words, you call `start()`, which talks to the operating system, which creates a new thread and calls `run()` there on your behalf.

Slide 44

## Thread Methods

```
/** Make the current thread sleep for the given time */
static void sleep(long milliseconds);

/** Wait for the thread to terminate */
void join();

/** Make the thread a daemon thread */
void setDaemon(boolean on);
```

Program will not wait for thread to finish before exiting

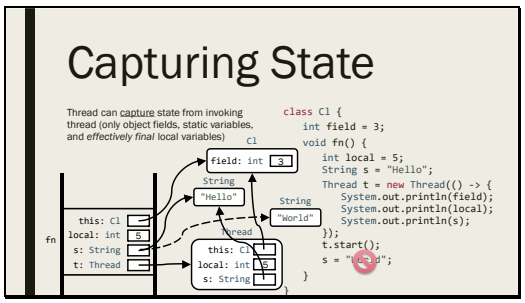
Other useful methods can be used to make the current thread go to sleep, make the current thread wait for another thread (the name join should be understood as to join up with another thread, as in you're out for a walk with your friends and you get separated so you join up), or to make a thread a daemon thread, which is a thread that the process will not wait for. By default, a process does not end until all of its threads finish, but daemon threads are ignored for these purposes.

Slide 45

## NERD CORNER

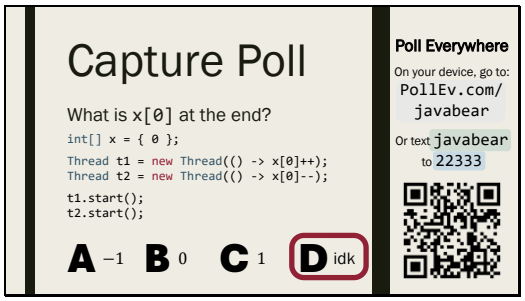
An in-depth discussion of lambda capture

Slide 46



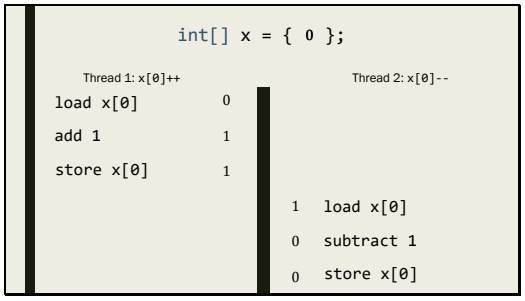

Multiple threads can share state by capturing the variable in the lambda. However, if a local variable is later edited, note that the copy in the thread will get out of sync with the original variable. For that reason, Java has chosen to disallow this. You may not capture a local variable that is not "effectively final."

Slide 47



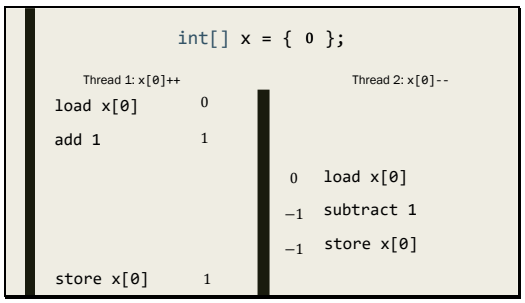

Note it's okay to capture x because it is a pointer to an array in the heap. The pointer itself does not change, even if the value inside the array changes, so it's okay to capture the pointer. Unfortunately, the final value inside the array depends on how the threads were run.

Slide 48



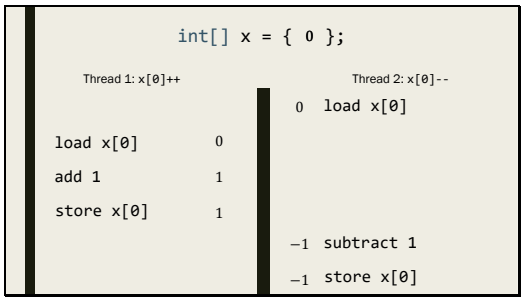

If one thread runs before another, the final value is 0, as expected.

Slide 49



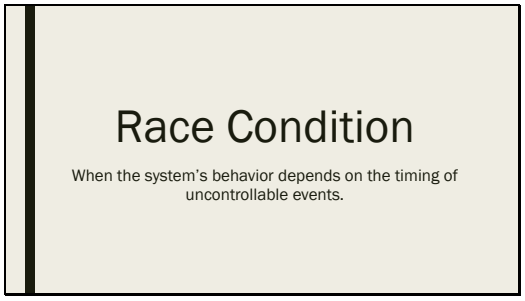

But if we get unlucky and thread 1 gets interrupted halfway through, suddenly we get a final value of 1

Slide 50



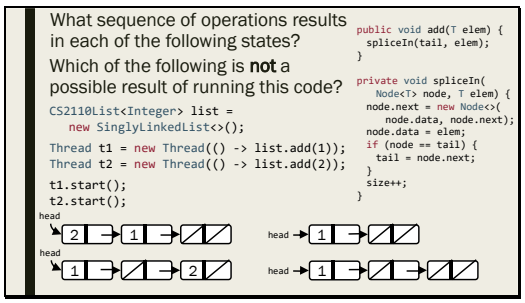

And the other way around is also possible. Remember, we do not control what order the operating system schedules and pauses our threads. So the final value here is something we do not know!

Slide 51



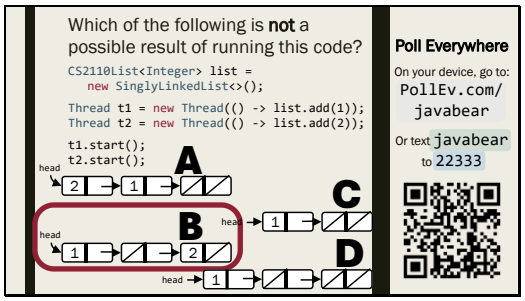

This is a classic race condition, when two threads are both racing to finish and depending on who wins, the result changes.

Slide 52

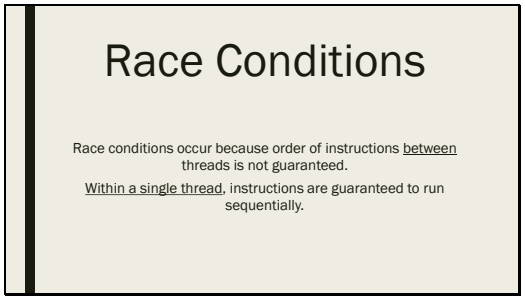



This is an exercise from the lecture notes online.

Slide 53




Slide 54



Slide 55

## Recap

- Parallelism lets us do multiple things at the same time
- Split code into concurrent threads (abstraction over cores)
- Give up control of execution order and speed to OS
- Race conditions if multiple threads write to same data

Slide 56