CSE 374
Programming Concepts & Tools

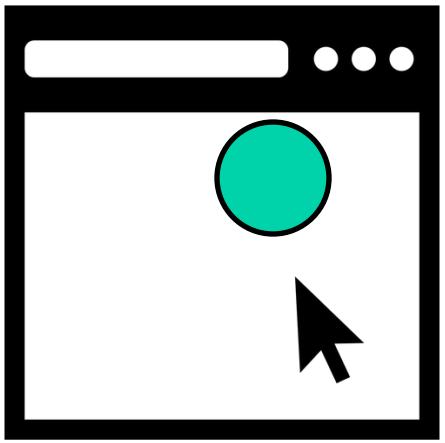
Brandon Myers

Winter 2015

Lecture 26/27 – Intro to Concurrency & Parallelism

Scenario 1/3

- Graphical user interface: The main loop alternates between processing user input (mouse movements) and updating and rendering shapes (suppose that takes 100ms).

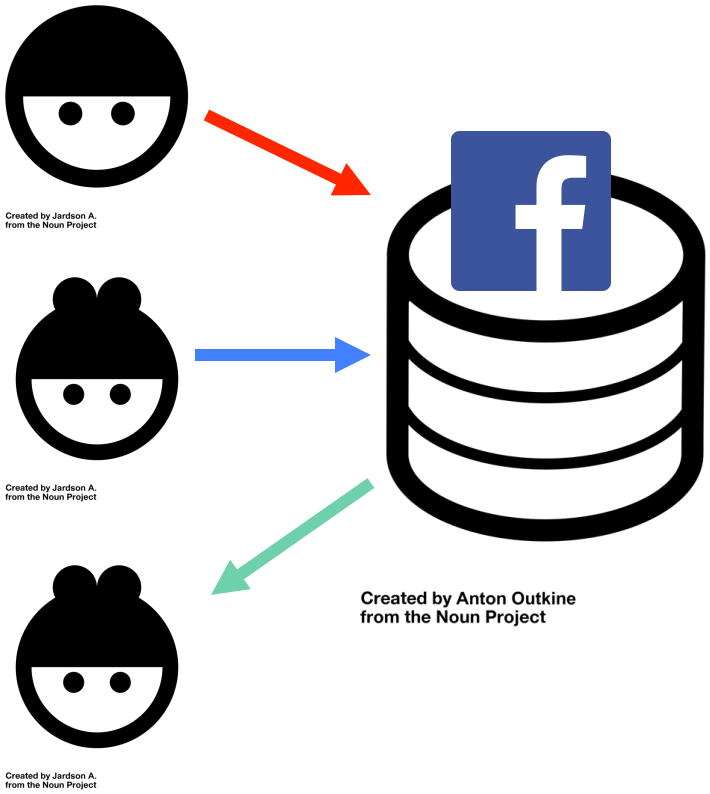


Created by Adriano Emerick
from the Noun Project

- How do we keep the mouse from pausing every time the processor is busy drawing the shapes?

Scenario 2/3

- Social network is accessed by multiple users

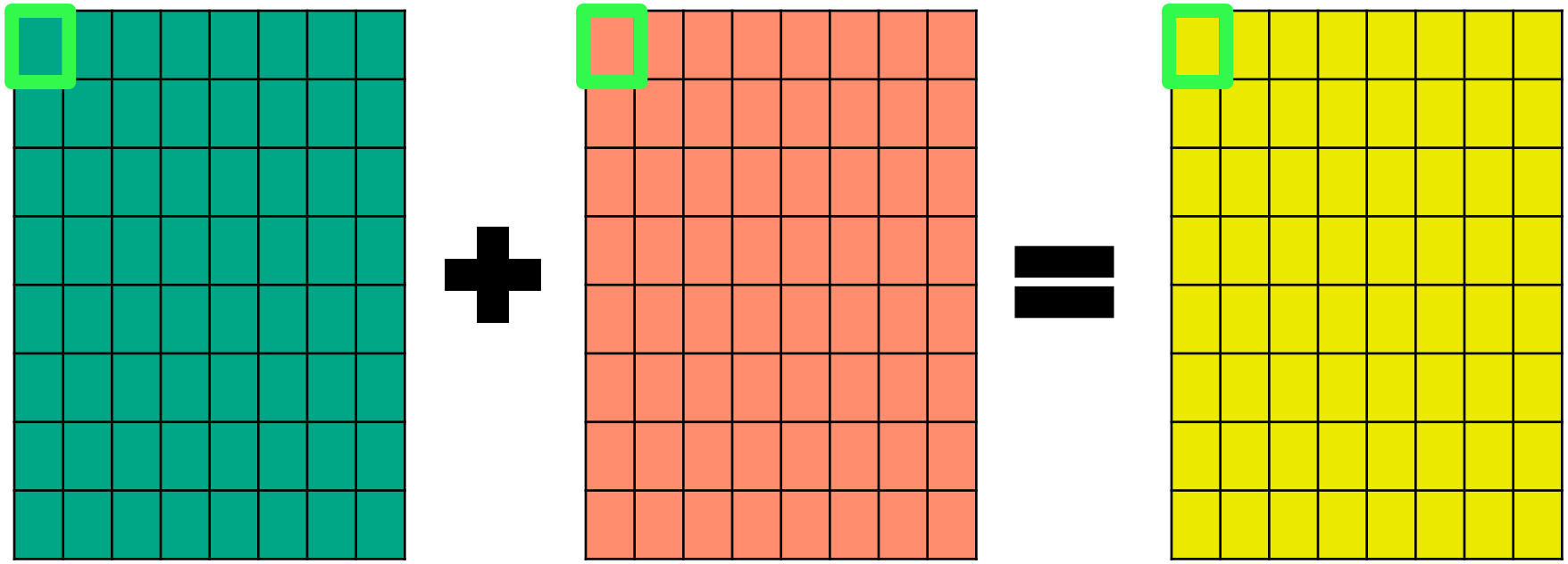



```
post.write_comment(bob,
                   “knock knock”)

post.write_comment(alice,
                   “who’s there?”)

post.read_comments()
   => [“who’s there?”]
```

- Can users see comments out of order?

Scenario 3/3

- Adding two large matrices.



- How can I do it K times faster?

Parallel vs. Concurrent

- Precise definitions of these two terms vary depending on who you talk to, but experts agree at least that they are different
- Here are my preferred definitions:
- Parallel:
  - multiple computations running simultaneously using *independent resources*
  - for energy-efficiency or performance; *never* required for correctness
- Concurrent:
  - multiple computations running simultaneously
  - sometimes required for correctness (i.e. avoiding deadlock)
- With these definitions, the set of parallel programs is a subset of the set of concurrent programs

Milk analogy

- grocer and customer, 1 shelf
- customer gets there first; waits at shelf for milk to appear
- grocer comes over and waits for customer to move away from shelf, so she can put some milk there
- deadlock! We need concurrent access to shelf so that the grocer may proceed even when the customer is waiting
- Make it a concurrent program: give the shelf two sides, so that even if customer is waiting on one side of the shelf, the grocer can still put milk on it

Parallelism for milk

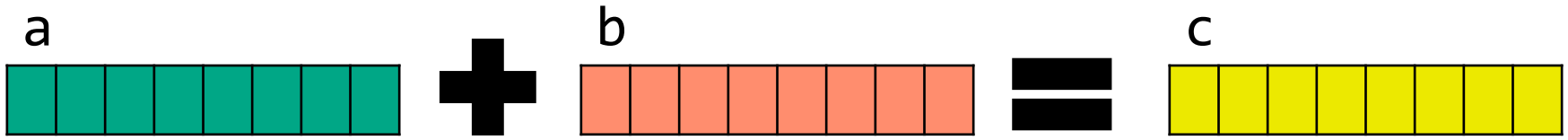
- Let's complicate the story.
- Suppose now to take milk or put milk on the shelf, you need a shopping cart
- The store only has one cart
- So for the grocer and customer to proceed, they must alternate their use of the cart
- E.g., grocer uses cart to bring milk and put it on the shelf
  - customer then takes the cart and uses it to take milk off the shelf and bring it to the checkout
- If the store has *two carts*, then the customer and grocer can work **in parallel** without sharing one cart
- the cart is like a processor

more terminology

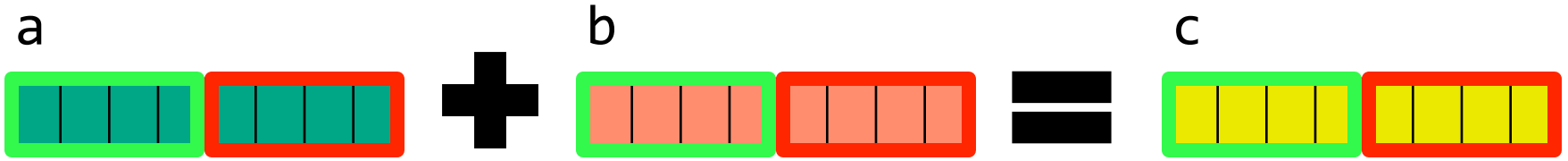
- task: a unit of work that may (or must) be run concurrently with other tasks
- thread: a software execution resource that can run one task at a time
- processor: a hardware execution resource that can run one thread at a time

- # of tasks determines the amount of concurrency
- # of processors determines the amount of parallelism

Adding two arrays (in parallel)

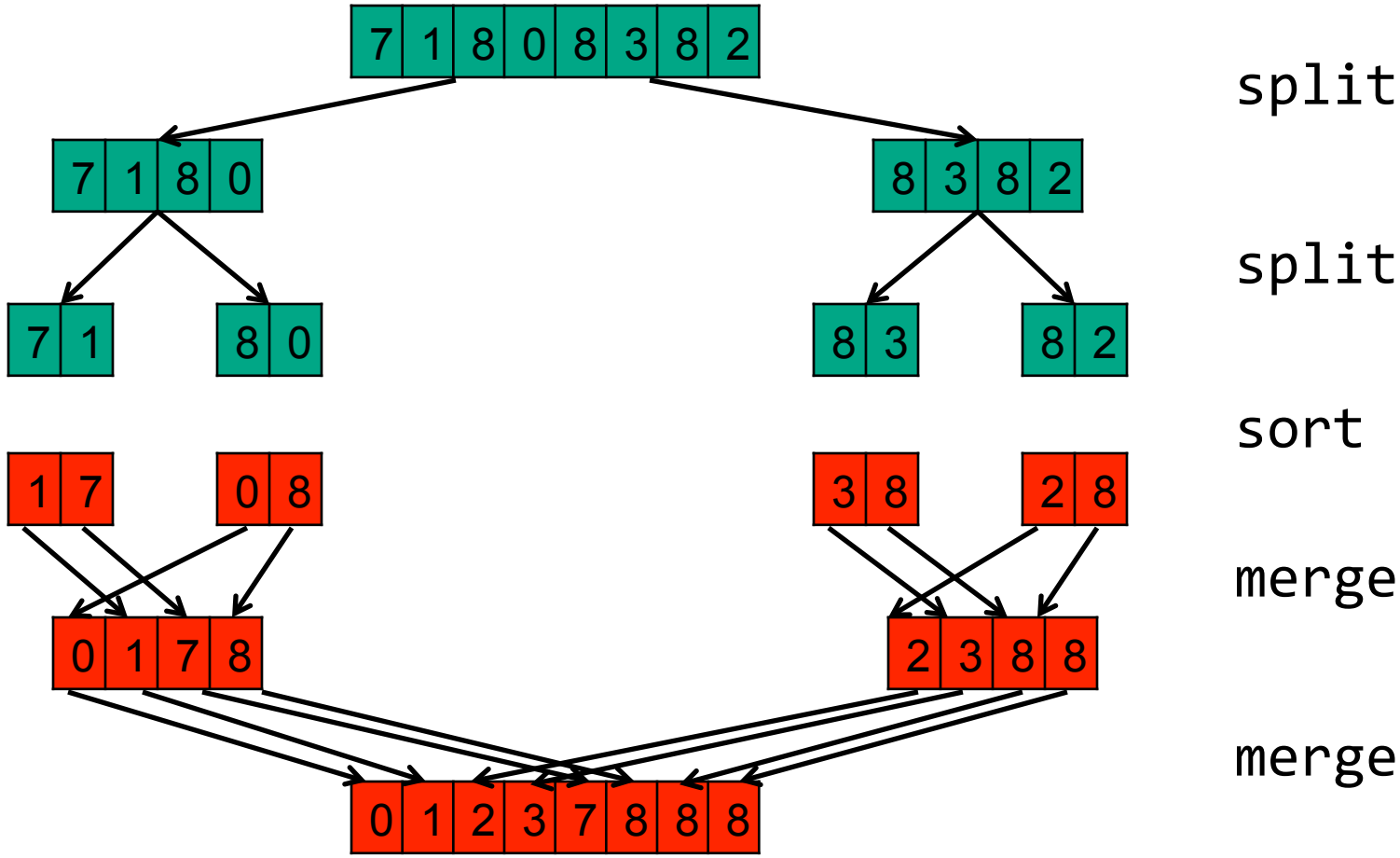



```
c[i] = a[i] + b[i]
```



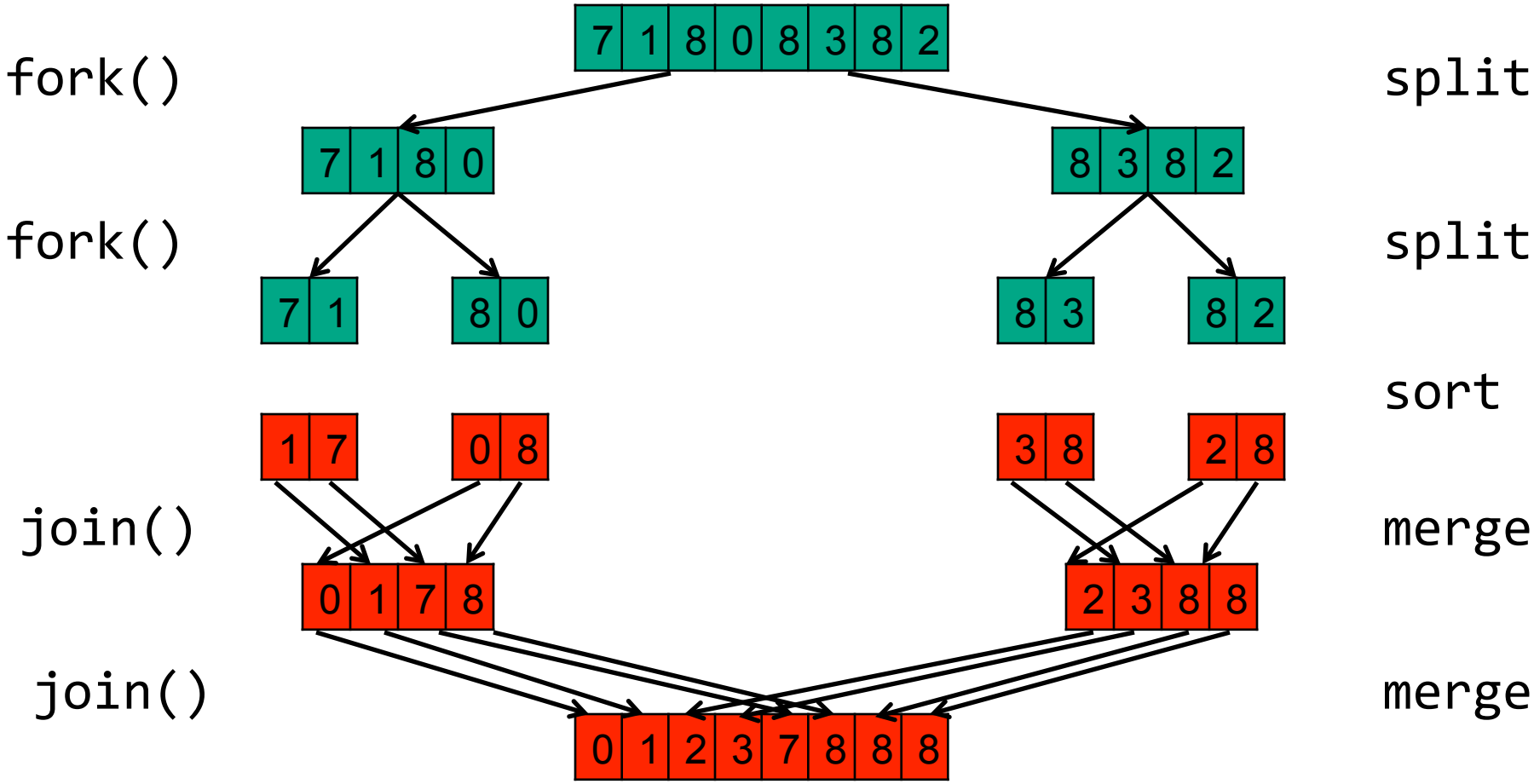
- Each element c[i] of the result is determined only by a[i] and b[i]
- So two tasks can compute in parallel without touching the same data
- see parallel_array_add.cc

Sorting an array (in parallel)



- recursively create a new task to sort the left and right child
- task and parent task must coordinate before merge!

Fork-join parallelism



- create a new task with fork(); wait for a task (and its result) with join()

Sharing data

- In our fork-join mergesort, we coordinated two tasks with join(). Think of fork() and join() as passing **ownership of data** between tasks.
  1. parent task forks two children, effectively granting ownership of a subarray to each child
  2. each child sorts its subarray (reads and writes)
  3. parent task joins both children; now it owns the array again, and may see the results of the childrens' actions
  4. parent task does the merge (reads and writes)
- This is a nice model! But are there programs that can't be expressed with fork-join?

Shared counter example

- Suppose we have a website that returns to the user just the next count
  - alice: GET → 144
  - bob: GET → 145
  - bob: GET → 146
  - alice: GET → 147
- No number may be skipped and no number may be returned twice
- first try: `shared_counter1.cc`

Data race!!

- *data race*: when two tasks access the same data (without synchronization) and at least one of them does a write

`counter++`

This operation really involves reading the current counter from memory, adding one, and writing the new value to memory.

So we might get this execution:

Alice READS 144

Bob READS 144

Bob WRITES 145

Alice WRITES 145

Mutual exclusion

- We want `counter++` behave like one uninterrupted operation.
- This is possible by maintaining *mutual exclusion* of threads touching `counter`.
  - This means only one thread may read or write counter at any given time
- second try:
  - we'll require a thread to lock a "mutex" before it is allowed to read and write `counter`
  - if a thread tries to lock a mutex that is currently locked, it must wait until it gets unlocked
  - `shared_counter2.cc`

Summary

- **Concurrency** and **parallelism** are different ideas (regardless of your precise definition of them)
- parallel programs are a subset of concurrent programs where tasks do not need to be run on with independent resources for correctness (parallelism is for performance and energy)
- two concurrent *tasks* can only safely communicate through synchronization constructs provided by the programming language, e.g.
  - fork and join
  - locking and unlocking the same mutex
  - transactions (we didn't get to talk about it)
  - message passing (we didn't get to talk about it)
  - version control (git) with merging on conflicts…

Final review session

- 6:45-8:45pm
- which one?
  - Monday 3/16
  - Wednesday 3/18

- If you plan to attend, make your voice heard. Take the poll on the homepage about which past exam questions to go over.

Course wrap-up

A slide from lecture #1

- We have 10 weeks to move to a level well above novice programmer:
  - Command-line tools/scripts to automate tasks
  - C programming (lower level than Java; higher than assembly)
  - Tools for programming
  - Basic software-engineering concepts
  - Basics of concurrency
- That's a lot!
- Get used to exposure, not exhaustive investigation
  - This is not intro programming anymore

just some of the things you learned

- how to get around Linux and the command line
- how to automate tasks with scripts
- how to do powerful text search and processing with regular expressions
- what's going on under the hood
  - how programs are stored
  - how programs are run
- how multiple source files are turned into an executable
- how to use an interactive debugger effectively
- why you should be thankful when you get a NullPointerException in Java
- how to find memory errors and memory
- how to work on a multi-file, multi-person code project

Where from here?

- Advanced non-major CSE courses
  - CSE 373: if you liked 143/HW5 and want more data structures and analysis of complexity. Also a pre-req for some of the 400's
  - CSE 417: computation theory beyond 373
  - CSE 410: if you liked learning about how programs are stored in memory and run, this will take you much deeper!
  - CSE 413: if you liked learning about function pointers, const, and object-oriented programming; or curious how programming languages work
  - CSE 414: data management is useful for any programmer or computer user; also learn more about parallelism and concurrency