Parallel Systems

CSCI 320

We have parallel hardware…
We need learn to use them!

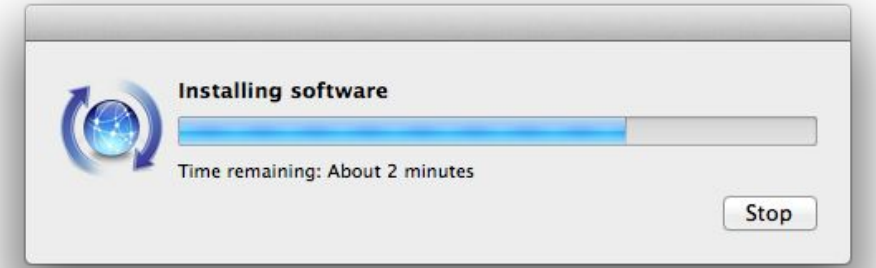
Objectives

1. Understand a variety of parallel hardware
    a. How it *branches* from sequential hardware
    b. How people refer to them
2. Understand some problems caused by parallel hardware
3. Have some idea about parallel software

Concurrency vs Parallelism

- Concurrency
  - Performing multiple tasks at once
  - The tasks themselves could be sequential
  - Goal: *To handle multiple things*
- Parallelism
  - May divide a task into subtasks
  - Performing multiple actions simultaneously
  - Goal: *To improve performance*

Concurrency Example



- Consider this GUI Element
  - Separate tasks
    - Perform the actual install operation
    - Draw and update the graphic

Concurrency Example (continued)

- Create multiple ***threads*** to handle each individual task
- A thread:
  - A sequence or block of instructions (machine code)
  - Given to a **scheduler** of the operating system
- The **scheduler** decides **how** and **when** to run the code on the computer

Aside: Scheduling

- How does the OS **schedule** threads?
- Consider:
  - Many threads exist - potentially from different programs
  - Don't know how long a thread will take
- Typical strategies:
  - FIFO, round-robin, priority-based RR, etc

Both concurrency and parallelism share many similar challenges!

Consider the following code.

```
long long sum = 0;
for (x = 0; x < 100 ; x++) {
    sum = sum + arr[x];
}
```

Can you do this faster with parallel processing?

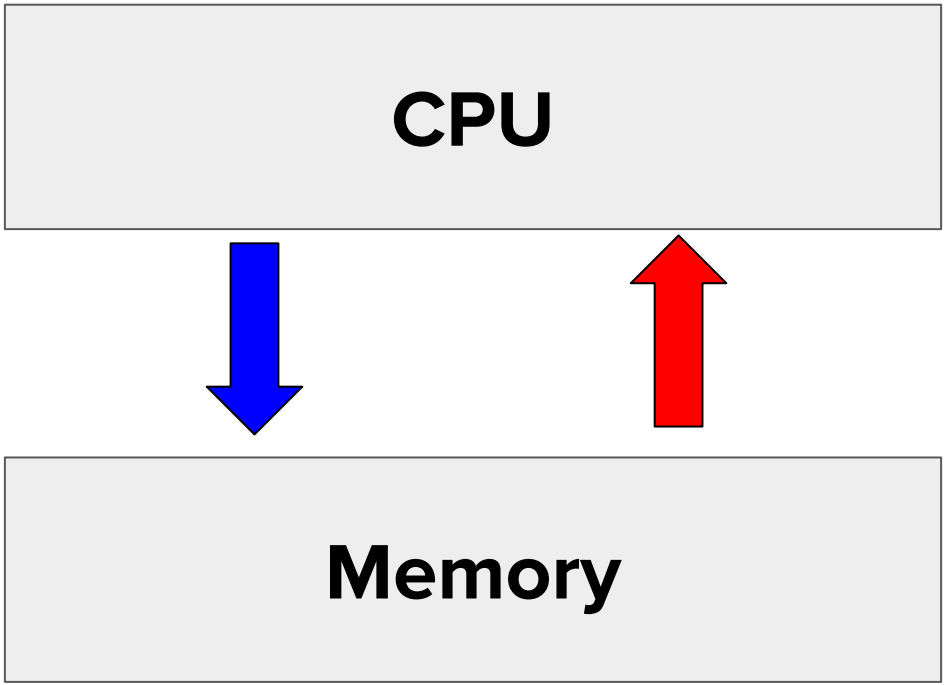
Memory, Instructions

- Where exactly is the array **arr** stored in memory?
- How does multiple processors interact?
- How can multiple processors interact with data?

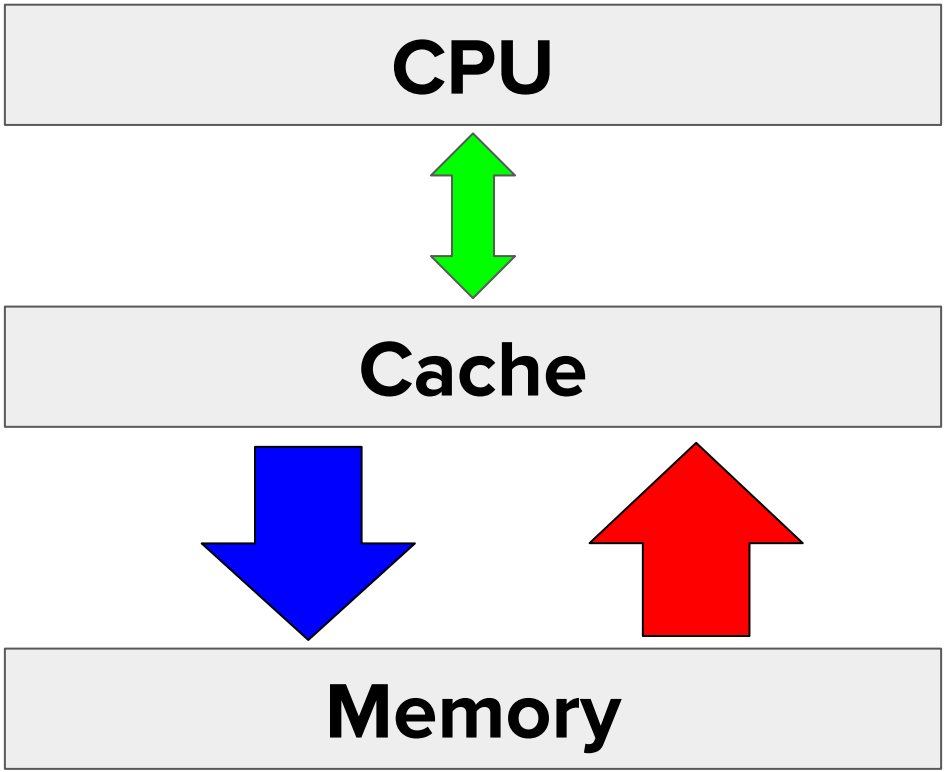
The von Neumann model

- Classical model of *computers*
- 1945
- **Computers have:**
  - A *CPU* or *core* or *processor*
    - ALU (arithmetic logic unit)
    - Control unit
  - Main memory
  - Input and output

The von Neumann Bottleneck



Modification - caching



Modifications to von Neumann

- Cache
- Virtual memory
- **ILP**

Modifications to von Neumann

- Cache
- Virtual memory
- **ILP - *Instruction Level Parallelism***
  - Pipelining
  - Multiple Issue
  - (You do not need to know the details for the course, just know that they exist!)

Parallel Systems

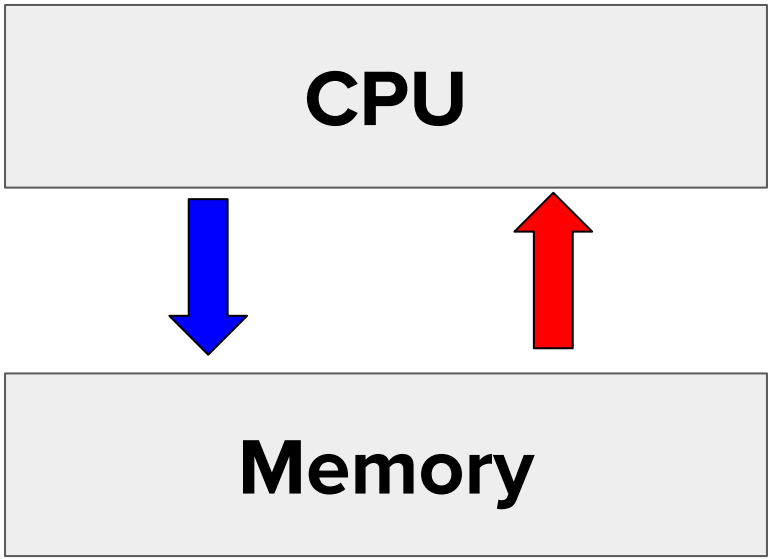
- *Technically* all systems are parallel because of ILP
- However:
  - Invisible to programmer
  - Can't really code for it on purpose
  - Fun branch prediction, etc etc etc
  - (Computer architecture would focus a ton on these)
- *We will talk about parallel systems we can interact with*

Parallel Systems Classification

- **Flynn's Taxonomy**
  - Classification of systems based on the number of instruction streams and data streams it can handle
  - Instructions, Memory.

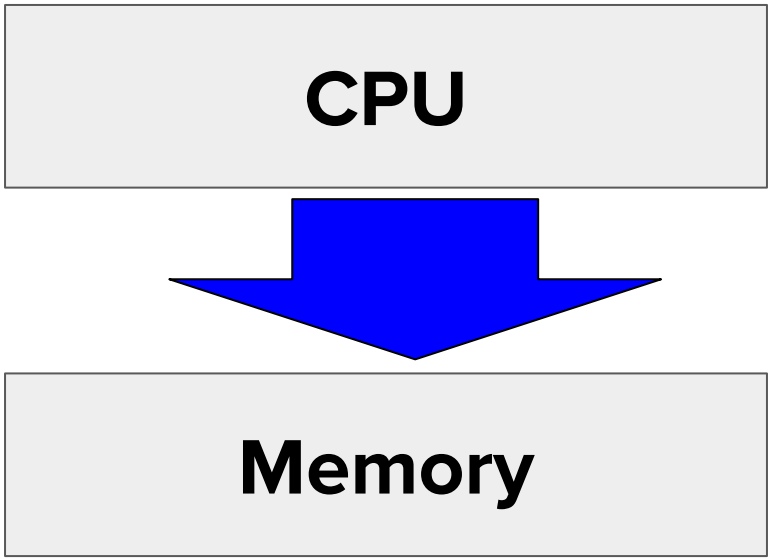
SISD

- *Single Instruction, Single Data*
- This is the standard von Neumann computer



SIMD

- *Single Instruction, Multiple Data*
- Do the same thing but on a *batch* of stuff at a time



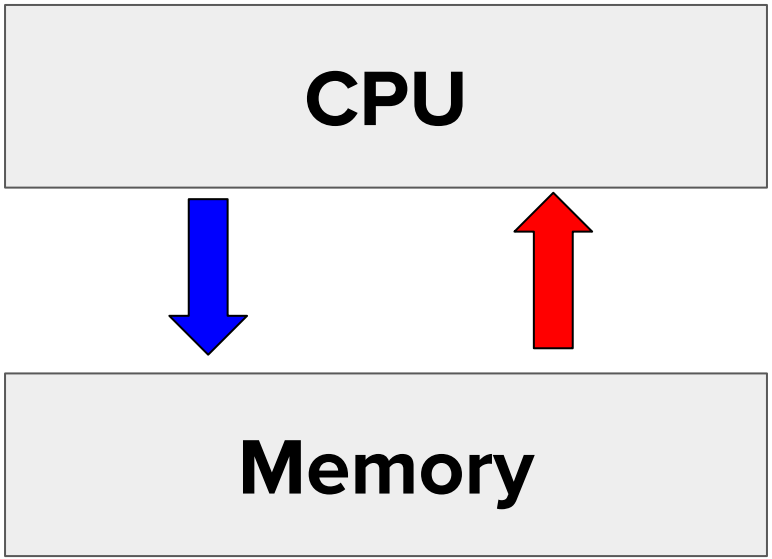
Good for SIMD!

```
for (x = 0; x < 1000 ; x++) {
    arr[x] += 15;
}
```

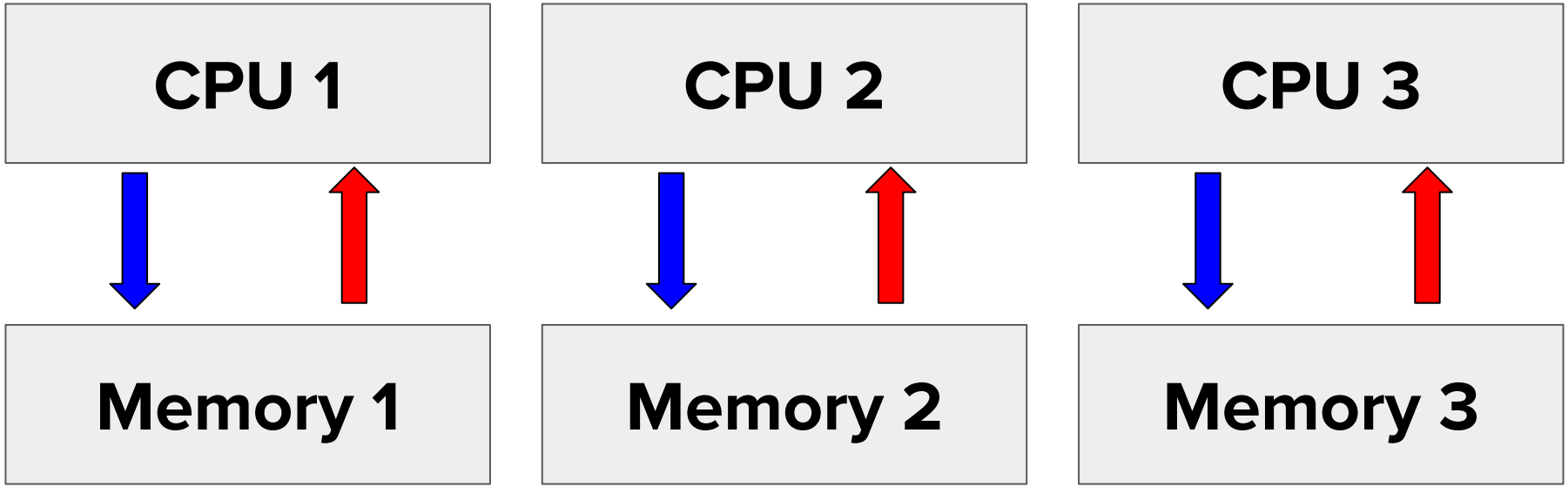
- *vector processors*
- Related to the concept of "data-parallel"
- Related to the other BIG technology - GPUs

SISD

- *Single Instruction, Single Data*
- This is the standard von Neumann computer

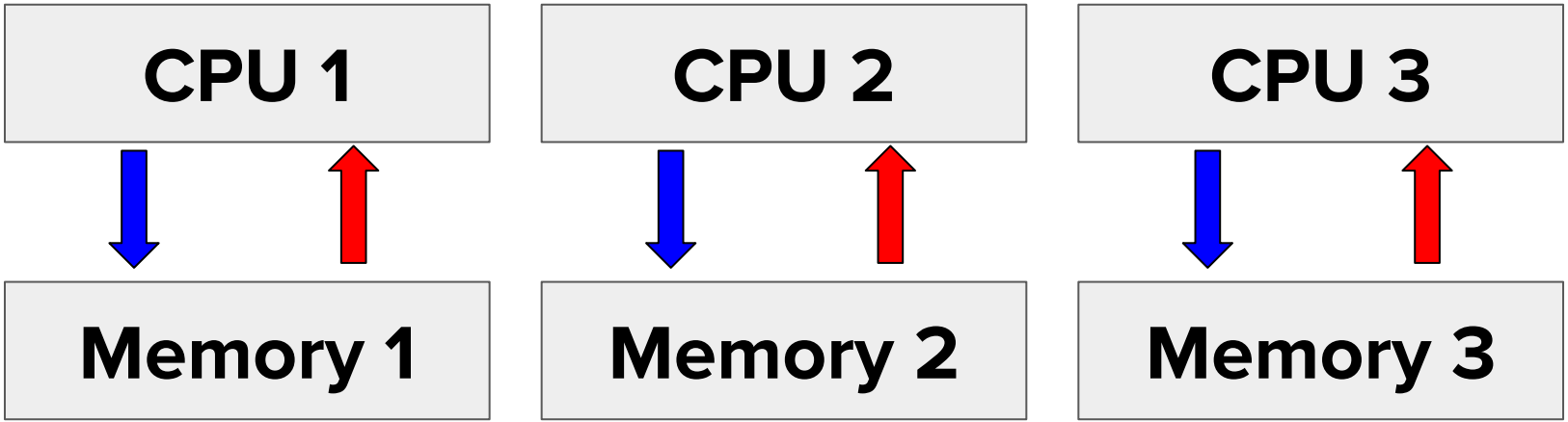


MIMD



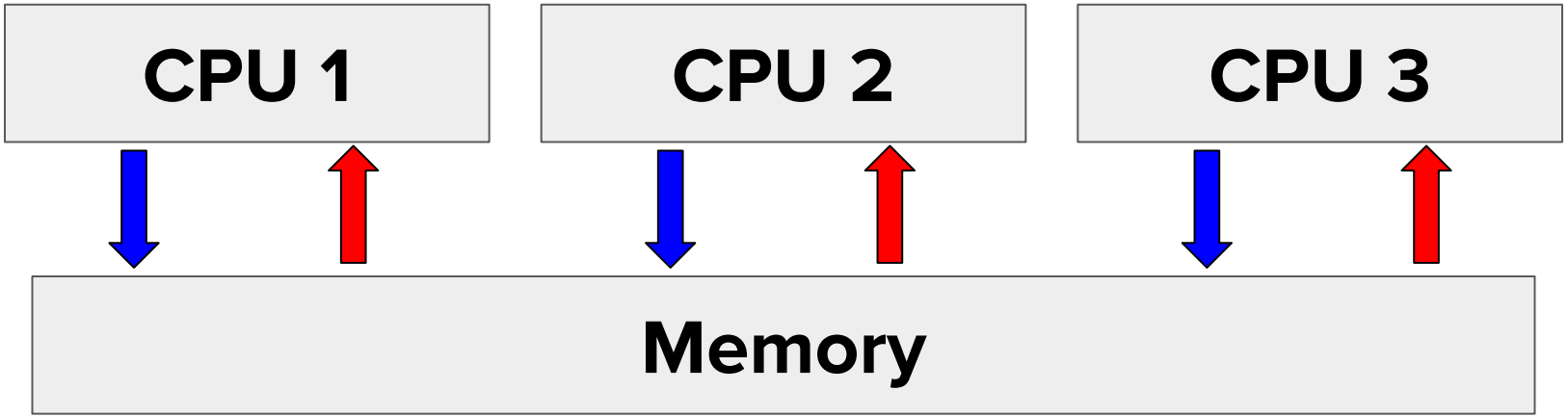
MIMD

- *Multiple Instruction, Multiple Data*
- Obviously this can do a lot!
- Challenges?



Shared-Memory

- Challenges?



Summary

- Flynn's Taxonomy
  - SISD - standard
  - SIMD
  - MIMD
  - Shared-Memory

Addendum

- SISD - standard
  - You already know how to program these!
- SIMD
  - Generally *straightforward** to learn
  - GPUs are similar (though not completely SIMD)
- MIMD
  - Powerful and intricate to program
- *Shared-Memory*
  - *We will focus on this!*

Issues and Software

CSCI 320

Basic Classification

- SISD - standard
- SIMD
- MIMD
- Shared-Memory

Other Classifications

- Hardware/Scale based
  - How many processors? How connected are they?

Scale

- Small Scale
  - Multi-Core Systems
  - SMP (Symmetric Multiprocessors)
    - Many identical processors
    - Connect by a bus
- Large Scale
  - Clusters
  - Grid computers

Parallel? Distributed?

- "Parallel Computing" and "Distributed Computing" should, in theory be *distinct*
  - In practice the difference blurs - "PDC"
  - Usually:
    - Parallel computing deals with smaller scale
      - Memory is shared or at least ***close***
    - Distributed computing is usually about clusters of machines
      - **Communication is the bottleneck**

Can't hope to cover everything!

We'll do ***Parallel Computing*** with a focus on programming in the *shared-memory* setting in additional to the theory of efficient and **correct** parallel programs.

A Question!

Consider this code!

```
ulong incr = 21;
for (x = 0; x < 100 ; x++) {
    arr[x] = arr[x] + incr;
}
```

Which type of parallel system is this easy in? Difficult in?

Everything

```
ulong incr = 21;
for (x = 0; x < 100 ; x++) {
    arr[x] = arr[x] + incr;
}
```

This is easy in pretty much every type of parallel system.

(Though it's faster in some)

Embarrassingly parallel.

Task-Parallel and Data-Parallel

- Task-Parallel
  - Divide the various tasks (or steps) among processors
  - Processors may work on the same data
- Data-Parallel
  - Divide the data up between the processors
  - Do the same operations on each processor

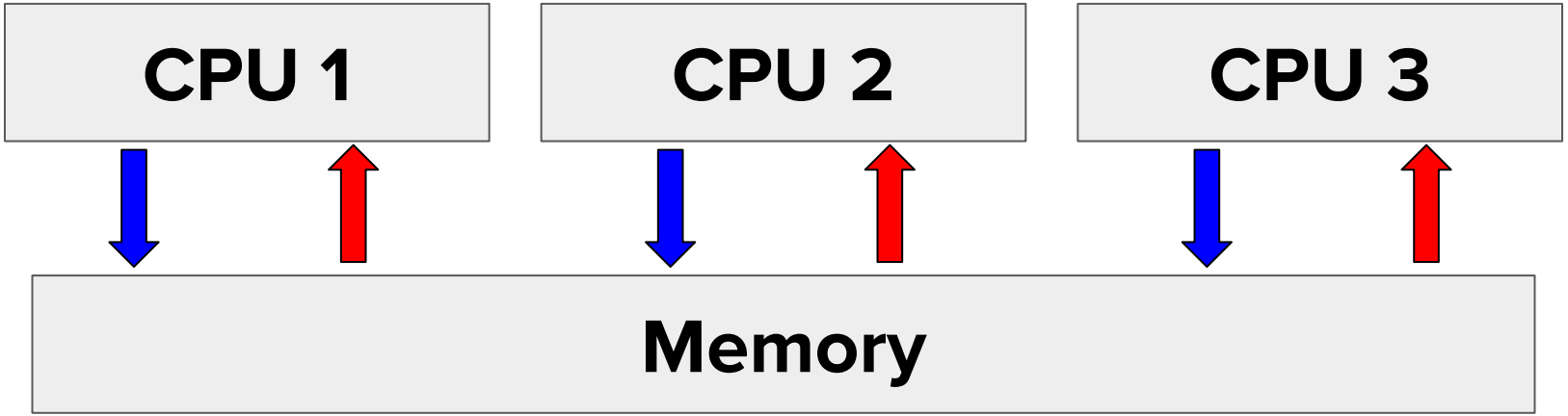
Good for Task-Parallel? Data-Parallel?

- SIMD
- MIMD
- Shared-Memory

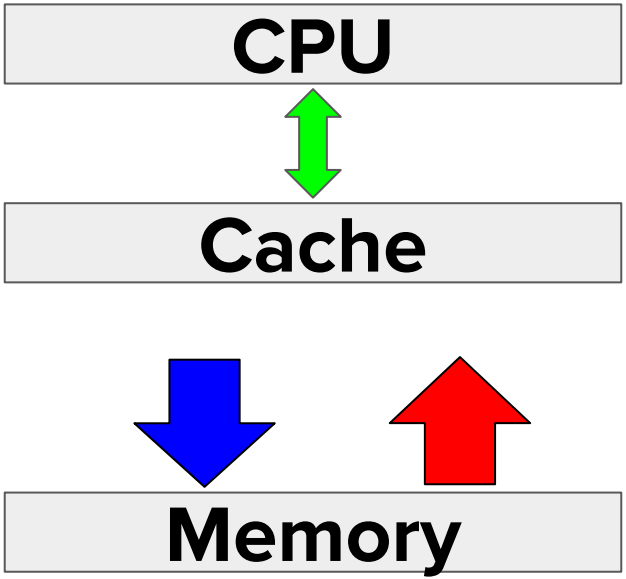
Shared-Memory

- We will focus on Shared-Memory Systems
  - Many "normal" computers are this type
- Advantages:
  - Easier to think about
  - Parallelizing "old" code
- Disadvantages:
  - MIMD is more "powerful" but
  - MIMD programs must be designed ground-up

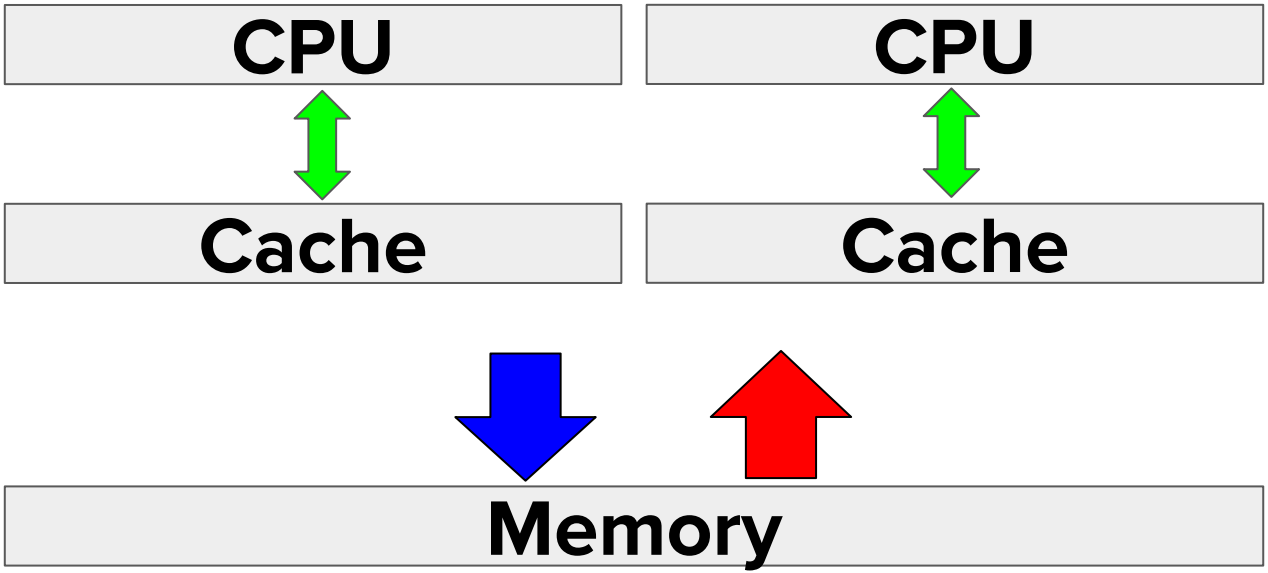
Shared-Memory



Modifications



Many Caches



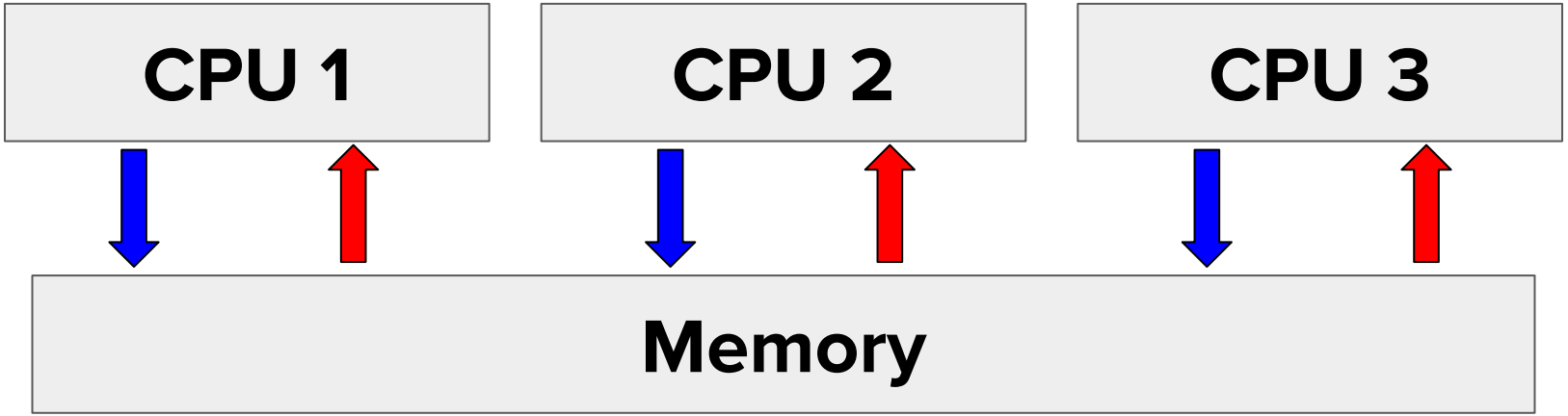
What are the values?

Time	CPU 1
0	`a=1;`
1	`b=a;`
2	`c=2*a;`
3	`a=5;`
4	`d=3*a;`

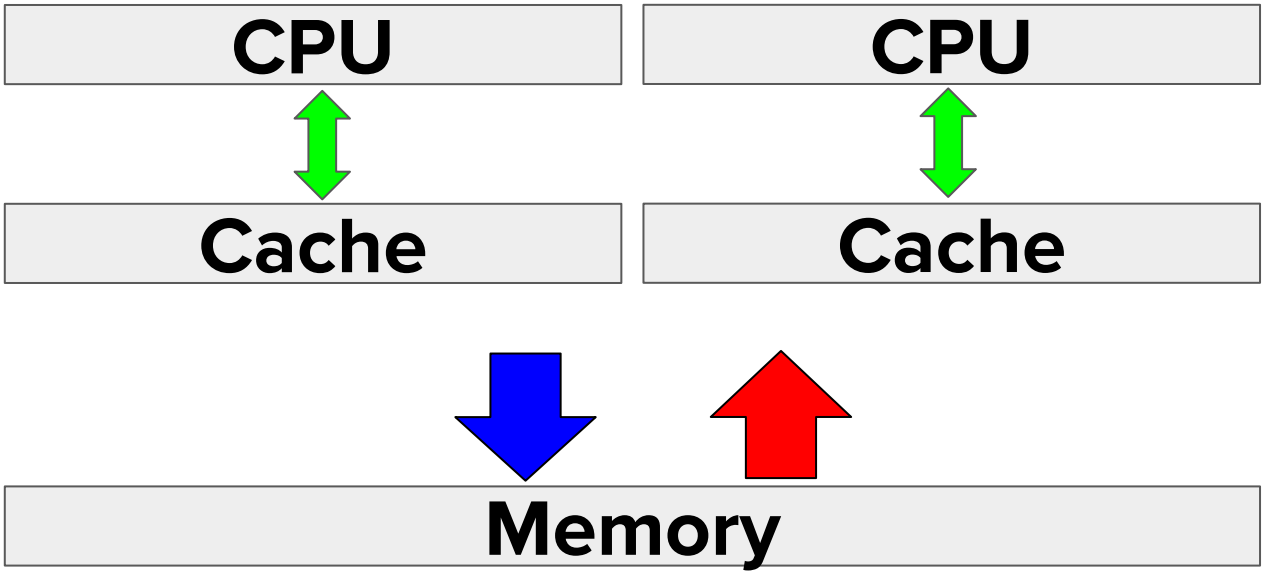
What are the values?

Time	CPU 1	CPU 2
0	`a=1;`	`/* Idle */`
1	`b=a;`	`c=2*a;`
2	`a=5;`	`/* Idle */`
3	`/* Idle */`	`d=3*a;`

Shared-Memory



What is the problem?



Cache Coherence

- The Cache Coherence problem deals with the possible inconsistencies in the caches of different CPUs
- Many "solutions"
  - Snooping
  - Tracking (directory-based)
  - *All of these impact performance*

Parallelization

Parallelization

- The process of converting a sequential program to a parallel program
- "Embarrassingly parallel" programs
  - Can be parallelized in a **straightforward** way
  - Usually just divide the work to different processors
- In general, programs are a bit more difficult to parallelize

Sketch of Parallelization

1. Identify the task to parallelize
2. Divide the work among the processes (or threads)
3. Collect the result together (somehow)
    a. Communication (MIMD)
    b. Synchronization (Shared-Memory)

Threads? Processes?

- Processes:
  - Usually a full program
  - Has its own *memory*
  - **More Independent**
- Threads:
  - Usually *part* of a program
  - Shares memory with threads of the same program
  - **Less independent**

Threads and Processes (2)

- Processes:
  - MIMD or distributed settings
  - Explicit communication
    - Message Passing
    - Broadcast

Threads and Processes (3)

- Threads:
  - Shared-Memory
  - Communication through variables:
    - **Shared variables - visible by all threads**
    - **Private variables - *local* to a specific thread**

It's mostly about how *independent* they are...

Similarities

- Both process and threads can run in parallel
- Both can cause **nondeterminism**
  - Different executions of the same program can result in different outputs, even if the inputs are the same
  - ***There is no guaranteed ordering on which process or thread "goes first"***

Thread-safety

- Does your code behave in a good way when threads are involved?
- Because **nondeterminism**
- Because shared access to the same resources

Conclusion

- Shared-Memory!
- We will start doing parallel programming next time!
- I will be demonstrating code through SSH!

Pthreads (1)

CSCI 320

Threads vs Processes

Goals (these two weeks)

- Understand the basics multithreading
  - Programming in Pthreads
  - Learning the general process
- Understand the *problems* multithreading can create
  - Implementing programs
  - Identifying bad behaviors
  - Debugging them
- Write *safe and correct* programs!

Non-Objectives

- **We don't care too much about:**
  - Performance (Big-O, etc)
  - Scalability
  - Parallelizing Complex Tasks
  - **Even *embarrassingly parallel* programs take work to parallelize!**
- We will save the complicated algorithms for later!

POSIX Threads

Pthreads

- A standard threading library for POSIX systems
  - Linux
  - OS X
- Commonly called "pthreads"
- Most C compilers support it
  - (We are using GCC, which supports it)
  - On Windows, unfortunately MSVC has its own Windows Threads

Threading!

- There are *many* other threading frameworks for languages
  - Java threads
  - C++14 threads
  - Etc.
- **They all share similar basic ideas!**
- C usually allows you to get at many more aspects of a thread than the other languages

That's it! Time to get into code!

The Hello World

```c
#include <stdio.h>

int main(int argc, char** argv){
   printf("Hello world");
}
```

Compiling C code (gcc)

```
gcc <filename.c> -Wall -o <filename>
```

- Useful Flags:
  - `-Wall` Give lots of warnings
  - `-g` Allows debugger usage (gdb)
  - `-o <filename>` specifies output binary
  - `-O2` compiler optimization (optional)

The Hello World - Goal

- We will now throw some threads in:
  - Create a bunch of threads
  - Have them all say "Hello World"
  - End the program

General Flow

- In a program, there's usually a *main* or *starting* thread
- This thread:
  - *Spawn* more threads
  - Wait until other threads *join* up with it
- Other threads:
  - Each thread has its own **local variables**
    - [They have their own stack]
  - Each thread has access to **shared variables**

The Hello World

1. Create some threads
2. Make threads display output
3. Wait until the threads are done
    - **How do we know they are done?**
    - **How do we wait enough?**

Thread Handler

- A variable which keeps track of threads
- Its *type* will differ depending on the API
- Usually has functions for:
  - Stopping the thread
  - Waiting on the thread
  - Etc.

Making Threads Act

- Threads must execute some code
- Usually:
  - Thread's code is written in a separate function
  - The thread is told to "go run that function"
  - The *main* code then continues onwards

Compiling C with Pthreads (gcc)

```
gcc <filename.c> -lpthread … -o <filename>
```

- Additional Flag:
  - `-lpthread` Tells the compiler to link the pthreads library
  - On some machines, you can get away without it!
  - (On others you will just have a compile error)

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

int main(int argc, char** argv){
   printf("Hello world");
}
```

```
...

void* SayStuff(void* blah){
   printf("Hello world, I am a thread!\n");
   return NULL;
}
//Threads always run some function.
//All Pthread functions must have this form.
```

```
...

int main(int argc, char** argv){
   int threads = 4; //We will make 4 threads
   pthread_t* handlers; //Array of handlers
   handlers=malloc(threads*sizeof(pthread_t));
```

Thread Handlers

- We want to be able to refer to threads after they are created
  - ***pthread_t*** is an **opaque** object
    - System specific
    - Doesn't have accessible members
  - Pthread functions examine/compare/operate on them
  - It basically just *refers* to a specific thread

```
...

int count;  //counter variable for loops
for (count=0; count < threads; count++){
   pthread_create(&handlers[count], NULL, \
      SayStuff, NULL);
}
```

pthread_create

```
pthread_create(pthread_t* ptr, pthread_attr_t* \
   attr, void* func, void* arg_ptr)
```

- What does it do? ***Spawns*** a thread.
- 4 Arguments:
  a. Pointer to a pthread_t (thread handler)
  b. Pointer to an attribute handler (advanced, use NULL)
  c. Pointer to the *function* the thread will run
  d. Pointer to the *arguments* given to the *function* called

pthread_create

- Have a return value (mostly to indicate errors)
- Advice:
  a. Make space (allocate) for the pthread_t object before creating the thread!
  b. Mostly use NULL for the attribute pointer
  c. Be careful with the *argument pointer!*
     - (Many tricky stuff! We will see next time)

Time to try it!

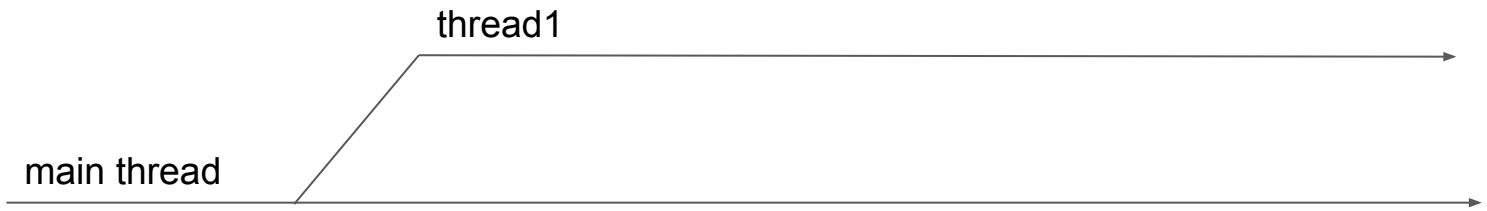
```
...

for (count=0; count < threads; count++){
   pthread_join(handlers[count], NULL);
}
```

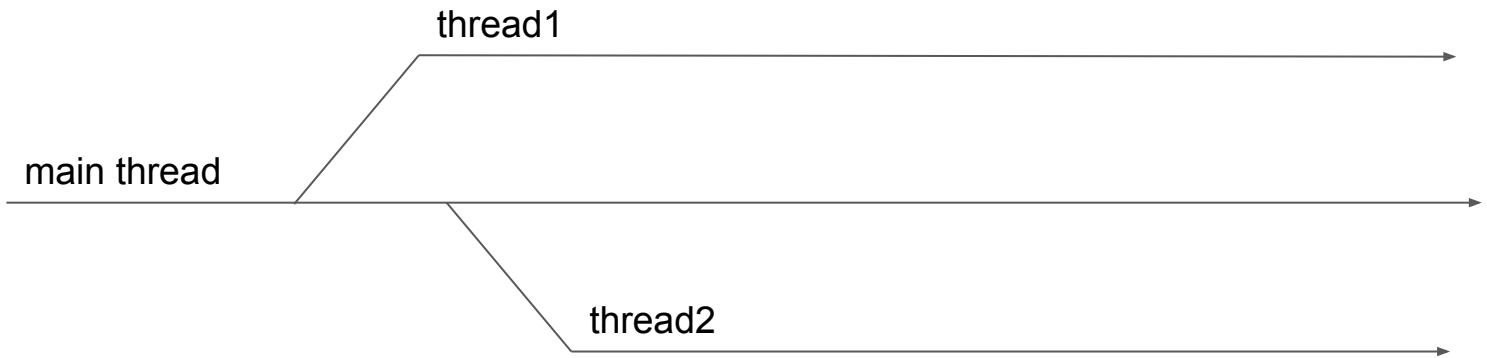
pthread_join

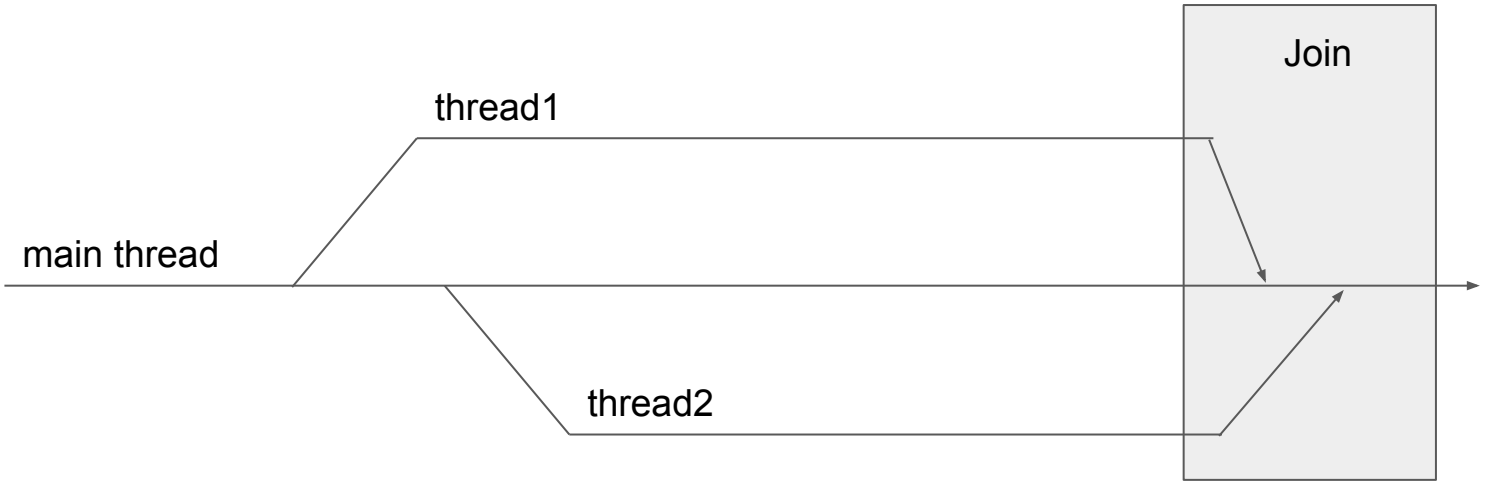
```
pthread_join(pthread_t thread, void** return_ptr)
```

- This is called a ***join***.
- 2 Arguments:
  a. Pointer to a pthread_t (thread handler)
  b. Pointer to *receive* the return value of the thread function


main thread


thread1
main thread


thread1
main thread
thread2


Join
thread1
main thread
thread2

Pthreads (2)

CSCI 320

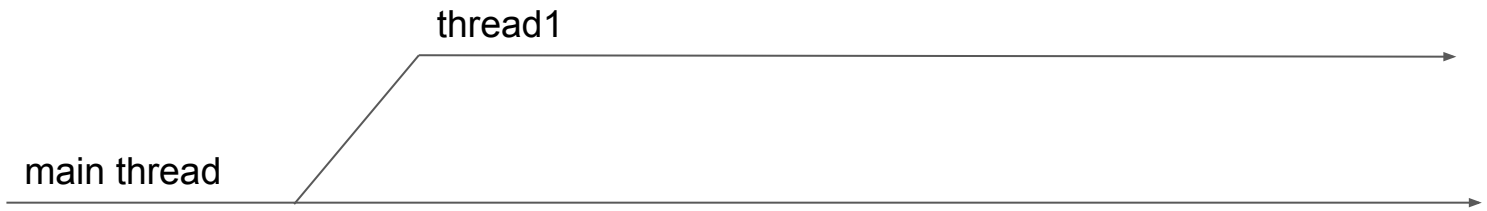
Threads vs Processes

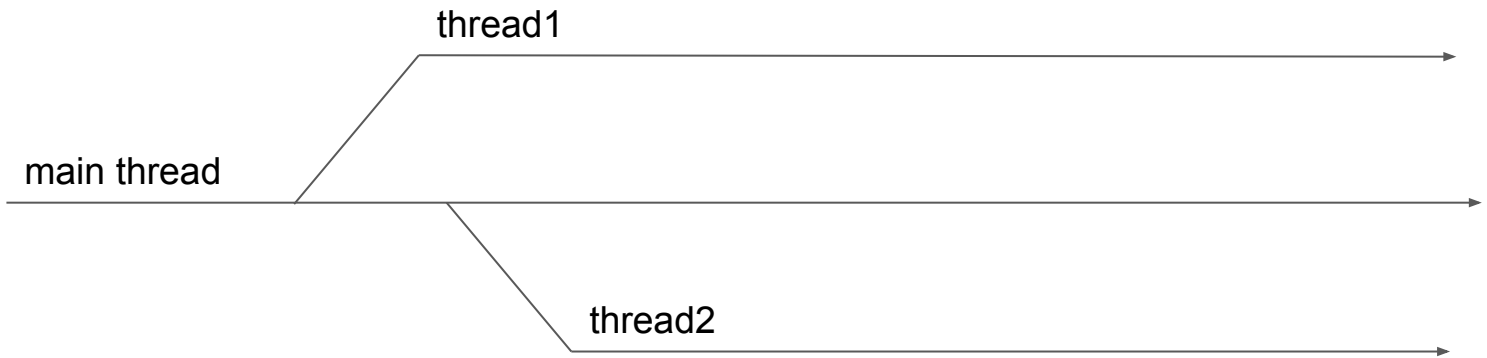
- Process
  - A "complete" program
  - **Private memory**
- *Threads*
  - A sequence of instructions in a program
  - A single *program* can have multiple *threads*

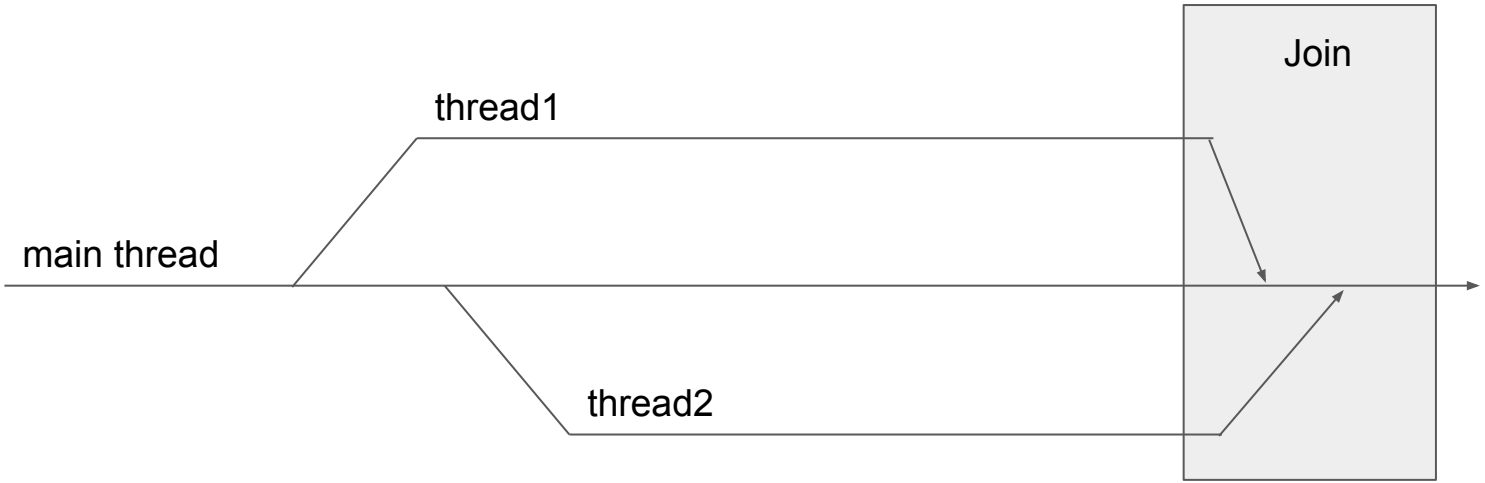
Threading in General

- Basic ideas:
  - *Spawn* or create
  - *Join* or synchronize


main thread


thread1
main thread


thread1
main thread
thread2


Join
thread1
main thread
thread2

Quick Review

Steps for Using Pthreads

1. Allocate memory for the thread handler
2. Create the threads
    a. Pass in the thread handler
    b. Pass in a function and arguments
3. Join the threads
4. Free the thread handlers

```
//Step 1
pthread_t* handlers = malloc(threads*sizeof(pthread_t));

//Step 2
for (int count=0; count < threads; count++){
    pthread_create(&handlers[count], NULL, Func, NULL);
}

//Step 3
for (int count=0; count < threads; count++){
    pthread_join(handlers[count], NULL);
}

//Step 4
free(handlers)
```

pthread_create

```
pthread_create(pthread_t* ptr, pthread_attr_t* \
   attr, void* func, void* arg_ptr)
```

- 4 Arguments:
  a. Pointer to a pthread_t (thread handler)
  b. Pointer to a attribute handler (use NULL)
  c. Pointer to the *function* the thread will run
  d. Pointer to the *arguments* given to the *function* called

pthread_create

- Does have a return value (mostly to indicate errors)
- Tips:
  a. Make space (allocate) for the pthread_t object before creating the thread!
  b. Mostly use NULL for the attribute pointer
  c. Be careful with the *argument pointer!*

pthread_join

```
pthread_join(pthread_t thread, void** return_ptr)
```

- 2 Arguments:
  a. Pointer to a pthread_t (thread handler)
  b. Pointer to *receive* the return value of the thread function

Sending/Receiving Information

- **All global variables are *shared!***
- Other methods:
  - Passing arguments to the thread's function
  - Receiving arguments from the thread

Passing Arguments

pthread_create

```
pthread_create(pthread_t* ptr, pthread_attr_t* \
   attr, void* func, void* arg_ptr)
```

- 4 Arguments:
  a. Pointer to a pthread_t (thread handler)
  b. Pointer to a attribute handler (advanced, use NULL)
  c. Pointer to the *function* the thread will run
  d. Pointer to the *arguments* given to the *function*

```
void* Hi(void* number){
    ??? \\Need a line here
    printf("Hi, I am thread %d \n", number);
}

int main(int argc, char** argv){

…
    for (int count=0; count < threads; count++){
        pthread_create(&handlers[count], NULL, Func, NULL);
    }
}
```

```
void* SayHi(void* number){
    int num = *(int*) number;
    printf("Hi, I am thread %d \n", num);
}
```

```
…
int j = 3;
for (int count=0; count < threads; count++){
    pthread_create(&handlers[count], NULL, Hi, (void*) &j);
}
```

More about (c)

- Many programmers will do hacks if they only need to pass a simple data type such as an **integer**
- Back in the "old" days
  - Common practice to cast integers or longs to pointers
  - **Why is this bad?**
- Nowadays, all compilers will give warnings for doing this!

```
void* SayHi(void* number){
    int num = (int) number;
    printf("Hi, I am thread %d \n", num);
}
```


```
…
for (int count=0; count < threads; count++){
    pthread_create(&handlers[count], NULL, Hi, (void*) count);
}
```

```
void* SayHi(void* number){
    uintptr_t num = (uintptr_t) number;
    printf("Hi, I am thread %d \n", num);
}
```

```
…
for (uintptr_t count=0; count < threads; count++){
    pthread_create(&handlers[count], NULL, Hi, (void*) count);
}
```

Sharing Values

- Pass the values in
- Pass in an actual pointer to a variable
- Global Variable
  - All threads have access
  - Useful for giving the same info to all threads

Return Values

Return Values

```
pthread_join(pthread_t thread, void** return_ptr)
```

- Tricky!
- The void* that the **return_ptr** points to will be filled by the return value of the function!
- This is to allow it to return pointers!
- But hacks! (They always happen!)

```
void* SayHi(void* number){
    int* ans = malloc(sizeof(int));
    *ans = 255;
    return (void*) ans;
}

…

int* ret_val;
for (int count=0; count < threads; count++){
    pthread_join(&handlers[count], (void**) &ret_val);
    printf("Thread returns %d \n", *ret_val);
    free(ret_val);
}
```

```
void* SayHi(void* number){
    uintptr_t ans = 235;
    return (void*) ans;
}
```

…

```
void* ret_val;
for (int count=0; count < threads; count++){
    pthread_join(&handlers[count], &ret_val);
    printf("Thread returns %d \n", (uintptr_t) ret_val);
}
```

Alternatives

- Recall in C you can also pass pointers!
  a. Could pass in a pointer to the place where the final answer should be put
  b. Have threads just dereference and change that
- Can also declare a global array for threads to put results!

Summary

- Pass in a thread rank for each thread
  - Either cast some *int* to a pointer
  - Or make an array of *ints* and pass their address
- Return Values
  - Allocate them in the thread function and pass back a pointer
  - Pass in a pointer so threads can directly modify
  - Or have threads put their results into a global array

Sum

CSCI 320

Steps for Using Pthreads

1. Allocate memory for the thread handler
2. Create the threads
    a. Pass in the thread handler
    b. Pass in a function and arguments
3. Join the threads
4. Free the thread handlers

Return Values

Sharing Values

- Return the value
- **Global Variables**

Return Values

```
pthread_join(pthread_t thread, void** return_ptr)
```

- Tricky!
- The void* that the **return_ptr** points to will be filled by the return value of the function!
- This is to allow it to return pointers!
- There are some hacks! (They always happen!)

```
void* SayHi(void* number){
    int* ans = malloc(sizeof(int));
    *ans = 255;
    return (void*) ans;
}

…

int* ret_val;
for (int count=0; count < threads; count++){
    pthread_join(&handlers[count], (void**) &ret_val);
    printf("Thread returns %d \n", *ret_val);
    free(ret_val);
}
```

```
void* SayHi(void* number){
    uintptr_t ans = 235;
    return (void*) ans;
}
```

…

```
void* ret_val;
for (int count=0; count < threads; count++){
    pthread_join(&handlers[count], &ret_val);
    printf("Thread returns %d \n", (uintptr_t) ret_val);
}
```

Review

- To pass in a thread rank for each thread
  - Either cast some *int* to a pointer
  - Or make an array of *ints* and pass their address
- Return Values
  - Allocate them in the thread function and pass back a pointer
  - **Or have threads put their results into a global array**

General Tips

- For passing in arguments:
  - Simple arguments or pointers - I suggest casting
  - Data every thread needs to use - Use global variables
- For getting results out:
  - **Use a global "results array"**
  - *IF you know what you are doing, use return values.*

Goals for Today

- Write a Parallel Sum
  - Show off some behaviors
  - Try out some more of the pthreads

Let's Start!

Parallel Sum

- Goal:
  - Find the sum of some numbers
  - Display the sum
- Simplifications (for demonstration):
  - We'll just sum the numbers 0 through *n*
  - We'll use a fixed number of threads (how many?)

```
uint64_t sum = 0;



//Function with void*

void* Sum(void* ptr){

    uintptr_t rank = (uintptr_t) ptr;

    int64_t block = NUMBERS/THREADS;

    for (int64_t i = block*rank; i < block*(rank+1); i++){

        sum += i;

    }

    return NULL;

}
```

```
uint64_t sum = 0;



//Function with void*

void* Sum(void* ptr){

    uintptr_t rank = (uintptr_t) ptr;

    int64_t block = NUMBERS/THREADS;

    for (int64_t i = block*rank; i < block*(rank+1); i++){

        sum += i;

    }

    return NULL;

}
```

Think

```
total = total + 5;
```

- Simplified line
- What must the CPU do?

Steps

```
total = total + 5;
```

1. Evaluate the RHS
    a. Read the value of *total*
    b. (Read the value of 5)
    c. Compute the result
2. Store the sum back to *total*

Here's some assembly...

```
int total=0;

int main() {

    total = total + 5;

    return total;

}
```

```
total:
      .zero   4

main:
      push    rbp
      mov     rbp, rsp
      mov      eax, DWORD PTR total[rip]
      add      eax, 5
      mov      DWORD PTR total[rip], eax
      mov     eax, DWORD PTR total[rip]
      pop     rbp
      ret
```

Steps

1. **Evaluate the RHS**
2. **Store the sum back to *total***

Crucially, these are TWO steps!

What are the values?

```
total = total + 5;
```

Time	Thread 1
0	Read *total*
1	Compute *total+5*
2	Store *total*
3	/* Idle */

What are the values?

```
total = total + 5;
```

Time	Thread 1	Thread 2
0	Read *total*	/* Other */
1	Compute *total+5*	Read *total*
2	**Store *total***	Compute *total+5*
3	/* Other */	**Store *total***

```
int64_t sum = 0;



//Function with void*

void* Sum(void* ptr){

    uintptr_t rank = (uintptr_t) ptr;

    int64_t block = NUMBERS/THREADS;

    for (int64_t i = block*rank; i < block*(rank+1); i++){

        sum += i; //Actually is sum = sum + i;

    }

    return NULL;

}
```

Race

- The *race* is one of the most important concepts in parallel and concurrent programs
  - Races are not *always wrong* but:
    - Most common cause of errors
    - Difficult to detect (in real examples)
- Terminology: *Race, Race Conditions, Data Race*

Race

- Set up by *concurrent access* to memory
  - (Shared-memory model)
- Caused by the *interweaving* of code
  - The ordering of threads is nondeterministic
  - Some steps may be "weaved" in between

Atomic Instructions

- **Atomic instructions**
  - Instructions which cannot be interrupted
    - The an increment on the ALU
    - Storing a value
  - Takes no "time"
- *However, MOST CODE ARE NOT ATOMIC!*
  - A line of code is almost certainly not atomic
  - Even *assembly instructions* might correspond to several "micro-operations" on the CPU

Locking

CSCI 320

```
uint64_t sum = 0;



//Function with void*

void* Sum(void* ptr){

    uintptr_t rank = (uintptr_t) ptr;

    int64_t block = NUMBERS/THREADS;

    for (int64_t i = block*rank; i < block*(rank+1); i++){

          sum += i;

    }

    return NULL;

}
```

Race

- The *race* is one of the most important concepts in concurrency
  - Most common cause of errors
  - Difficult to detect (in real examples)
- *Race, Race Conditions, Data Race*
- A program/function/library is not "thread-safe"
  - If erroneous behaviour occurs when multiple threads run/use them at the same time

Race

- Set up by *concurrent access* to memory
- Caused by the *interweaving* of code
  - Threads may execute the code in any order
  - Some steps may be "weaved" in between

Atomic Instructions

- Atomic instructions
  - Instructions which cannot be interrupted
    - An increment on the ALU
    - Storing a value
  - "Uninterruptible" - nothing can weave between
- ***However, MOST CODE ARE NOT ATOMIC!***
  - A line of code is almost certainly not atomic
  - Even *assembly instructions* might correspond to several "micro-operations" on the CPU

*EVEN CODE WHICH LOOK **ATOMIC** MIGHT NOT!*

```
total = total + 5; //No
total += 5; //???, likely no
total = 10; //Maybe?
//What if "total" must access memory?
```

- To really figure it out, have to look at instructions
  - It's never safe to rely on compilers!
  - Also depends on the system architecture!

The Bank Example

```
struct account_t{
    int balance;
    int account_number;
};

struct deposit_t {
    struct account_t* account;
    int amount;
};

void* deposit(void* dep){
    struct deposit_t* depos = (struct deposit_t*) dep;
    struct account_t* acc = depos -> account;

    acc -> balance = acc->balance + depos -> amount;
}

void* withdraw(void* dep){
    ...
}
```

Funny Business

- If two deposits happen at the same time:
  - Maybe both goes through
  - Maybe only 1 goes through
- Other exploits...

Locks

Locks

- Similar terms: locking, synchronization
- Idea:
  - Restrict threads from running some code at the same time
  - Do not allow weaving!
- Many types of "locks":
  - Mutex
  - Etc...

Mutex

- Mutually Exclusive
- Mutual Exclusion
- *Only allow one thread to execute a certain region of code at a time*
  - **Critical region**

```
uint64_t sum = 0;



//Function with void*

void* Sum(void* ptr){

    uintptr_t rank = (uintptr_t) ptr;

    int64_t block = NUMBERS/THREADS;

    for (int64_t i = block*rank; i < block*(rank+1); i++){

        sum += i; //Only 1 thread at a time

    }

    return NULL;

}
```

Locking

- "Locks" are an abstract resource threads can "grab"
  - Only one thread may grab a ***lock*** at a time
  - While a thread is *holding* the lock
    - Other threads will wait if they try to grab the lock
    - Once the thread *releases* the lock, other threads can try to grab the lock

Critical Region with Lock

- One thread
  - Seizes the *lock*
  - Runs its critical region
  - ***Releases*** the lock
- Other threads
  - Failed to grab the lock
  - Repeatedly* try until they grab the lock

*In reality, they often *wait* a little bit before retrying, otherwise a lot of performance cost!

```
uint64_t sum = 0;
//<DECLARE A TOTAL_LOCK resource for threads to compete for!



//Function with void*

void* Sum(void* ptr){

    uintptr_t rank = (uintptr_t) ptr;

    int64_t block = NUMBERS/THREADS;

    for (int64_t i = block*rank; i < block*(rank+1); i++){
          // <CODE TO GRAB A LOCK TOTAL_LOCK>
          sum += i; //Critical region!
          // <CODE TO RELEASE A LOCK TOTAL_LOCK>

    }

    return NULL;

}
```

Notes on locks

- There can be multiple different locks
  - Maybe we're computing both a sum and a product
  - Make two locks:
    - *product_lock*
    - *sum_lock*
- Locks are a kind of "object" that must needs initialization (created) before code can grab or release them
  - (In most programming languages)

Pthread Mutex

```
pthread_mutex_init(pthread_mutex_t* lock_var,...)
pthread_mutex_lock(pthread_mutex_t* lock_var)
pthread_mutex_unlock(pthread_mutex_t* lock_var)
```

- 3 most important functions
- The lock/unlock functions take a pointer to a specific lock
- The init function initializes a lock and takes in two args

pthread_mutex_init

```
pthread_mutex_init(pthread_mutex_t* lock_var,
      const pthread_mutexattr* attr)
```

- 2 arguments
  a. A pointer to a **pthread_mutex_t**
  b. An attribute pointer (use NULL most of the time)
- This *initializes* the lock

pthread_mutex_lock

```
pthread_mutex_lock(pthread_mutex_t* lock_var)
```

- Locks the mutex, acquires the lock, locks the region, etc etc…
- The first thread to successfully grab lock will continue onwards
- Every other thread will ***block*** at this statement (until the first thread releases the lock)

pthread_mutex_unlock

```
pthread_mutex_unlock(pthread_mutex_t* lock_var)
```

- Unlock the mutex, release the lock, unlock the region, etc etc…
- Every other thread will no longer be blocked
- **HOWEVER**, the order in which the other threads proceed is arbitrary - depends on the implementation of the locks

Usage (for us):

1. Declare a (global) variable for the lock
2. Initialize it with the *init* before any threads are spawned
3. For critical regions:
    a. Thread uses the *lock* function
    b. Thread performs critical code
    c. Thread uses the *unlock* function

Mutex Facts

- Promises
  a. Only one thread can lock a mutex at a time
  b. Every blocked thread *eventually* locks the mutex
- Non-Promises
  a. Threads will access the mutex in order*
  b. Threads will *immediately* lock a mutex once they become unblocked

*Some implementations guarantee *some* ordering, but this is very system dependent!

This means...

- Suppose we have a bank account with $500
- Each thread does its own action twice:
  a. Thread 1 adds $20
  b. Thread 2 adds $15
  c. Thread 3 adds $5

Possible Intermediate Values

Time	Value	Thread with Lock
0	500	
1	520	1
2	535	2
3	555	1
4	560	3
5	575	2
6	580	3

Possible Intermediate Values (2)

Time	Value	Thread with Lock
0	500	
1	520	1
2	540	1
3	555	2
4	570	2
5	575	3
6	580	3

To summarize...

- Guaranteed to get $580 at the end
- **Intermediate values are not guaranteed**
- Without locks, what are the possible values?

Possible Intermediate Values (Race)

Time	Value	Comment
0	500	Start
1	520	Thread 1 adds
2	505	Thread 3 adds (read at time 0)
3	520	Thread 2 adds (read at time 2)
4	540	Thread 1 adds (read at time 1)
5	560	Thread 2 adds (read at time 3)
6	510	Thread 3 adds (read at time 2)

pthread_mutex_destroy

```
pthread_mutex_destroy(pthread_mutex_t* lock_var)
```

- Used to clean-up
- Destroy a lock after it's no longer necessary
  - End of the program
  - Don't try to destroy locks that any thread is still *locking!*

Usage (extended):

1. Declare a (global) variable for the lock
2. Initialize it with the *init* before any threads are spawned
3. For critical regions:
    a. Thread uses the *lock* function
    b. Thread performs critical code
    c. Thread uses the *unlock* function
4. **Destroy the lock before exiting**

How do locks work?

Promises and Others

- **Promises (Must have these to be correct!)**
  - **Only one thread can grab the lock at a time**
    - Mutex
  - **Every blocked thread *eventually* grabs the lock**
    - Non-starvation
- **Other nice features (optional):**
  - Fairness
  - Speed! (We'll ignore this for now...)

Simple (and wrong) "Spinlock"

```
short lock_var = 0; //My "lock" is just a short
//0 if no one has the lock, 1 if someone controls it

void lock(short* mutex){
    while (mutex!=0){
        //Spin around and around and around!
    }
    *mutex = 1;
}

void unlock(short* mutex){
    *mutex = 0;
}
```

Wrong!

```
void lock(short* mutex){
    while (mutex!=0){
        //Spin around and around and around!
    }
    *mutex = 1;
}
```

This is a race condition!

We must not have weaving
in the *lock* function!

Atomic Instructions

- The upsides:
  - ***Atomic Instructions cannot cause race! (No weave)***
  - Atomic instructions are single instructions, fast!
    - If there's an atomic increment, it's faster than grabbing a lock and then incrementing a variable, etc...
- The downsides:
  - System dependent!
  - Restrictive in form!

Common Atomic Instructions

- ***Test and Set (TAS)***
- ***Compare and Swap (CAS)***
- **Fetch and Add (FAD)**
- Load-Linked, Conditional-Store

Standard Test and Set

```
int test_and_set(int* var, int value)
```

- ***Test and Set (TAS)***
  - Operates on the variable *var* points to
    - Sets the value of **var* to *value*
  - Returns the *old* value before the set
- Note that this is pseudo code
  - Real code on a later slide

Standard Fetch-and-Add

```
int FAD(int* var, int value)
```

- ***Fetch-and-Add (FAD)***
  - Operates on the variable *var* points to
  - Returns the "old" value of **var*
  - Increments the value of **var* by *value*

Standard Compare-and-Swap

```
bool CAS(int* var, int* expect, int value)
```

- ***Compare-and-Swap (CAS)***
  - Operates on the variable *var* points to
  - Compares the value of **var* and **expect*
    - **If** they are equal, ***then*** sets **var* to *value*
    - **Else** writes **var* into **desire*
  - Returns **True** or **False** depending on which operation was done

Pseudocode for Spinlock

```
short lock_var = 0; //My lock is just a short

void lock(short* mutex){
    while (TAS(mutex, 1)){ //Test and set to 1
    }
}

void unlock(short* mutex){
    *mutex = 0;
}
```

__atomic_test_and_set

```
bool __atomic_test_and_set(void* var, int memorder)
```

- Two arguments:
  - Pointer to variable to test-and-set on
  - Int corresponding to "memory ordering"
    - Complicated, just use **__ATOMIC_RELAXED**
    - (Which is just a macro for 0 in most cases)
- Function:
  - Does a TAS on *var* with and always sets it to *True*
  - Returns *True* if prev. value was *True*, else *False*

Pseudocode for Spinlock

```
short lock_var = 0; //My lock is just a short

void lock(short* mutex){
    while (__atomic_test_and_set(mutex, 0));
}

void unlock(short* mutex){
    *mutex = 0;
}
```

Spinlocks!

- **Guarantees:**
  - Mutex
  - Non-starvation
- **Does not have:**
  - Fairness
  - Speed!

Better lock!

- **Guarantees:**
  - Mutex
  - Non-starvation
  - Fairness

Idea - Ticketing!

- Wait in line:
  - Grab a ticket with a number
  - Then you wait! You get to go when the "Next Up" is your number!

Ticket Idea (Doesn't work without atomics!)

```
short lock_var = 0; //My lock is just a short
short curr_turn = 0; //Currently, 0 is the first to go!

void lock(short* mutex){
    short my_ticket = *mutex; //Grab my ticket!
    *mutex++; //This is the next ticket available!
    while (my_ticket != curr_turn) {
     //Spin and spin and spin!
    }
}

void unlock(short* mutex){
    curr_turn++; //I end my turn!
}
```

Spinning can be bad!

- *Spinning* or *Busy-Waiting*
- Disadvantages
  - Lots of quick, repeated access to try to read a variable
  - Consumes power and time
  - **Wasteful**
- Advantages?

Alternatives to Spinning

- *Yielding*
  - Tells the processor to delay the current thread
  - We'll worry over this later, but, IRL:
    - Processors have *many* threads to run
    - It usually keeps a queue of them
      - Works for a few moments on one
      - Switches frequently
    - *Yielding* just moves the current thread to the end of queue

Other Locks

CSCI 320

Mutex

- Mutually Exclusive
- *Only allow one thread to execute a region of code at a time*
  - **Critical region**

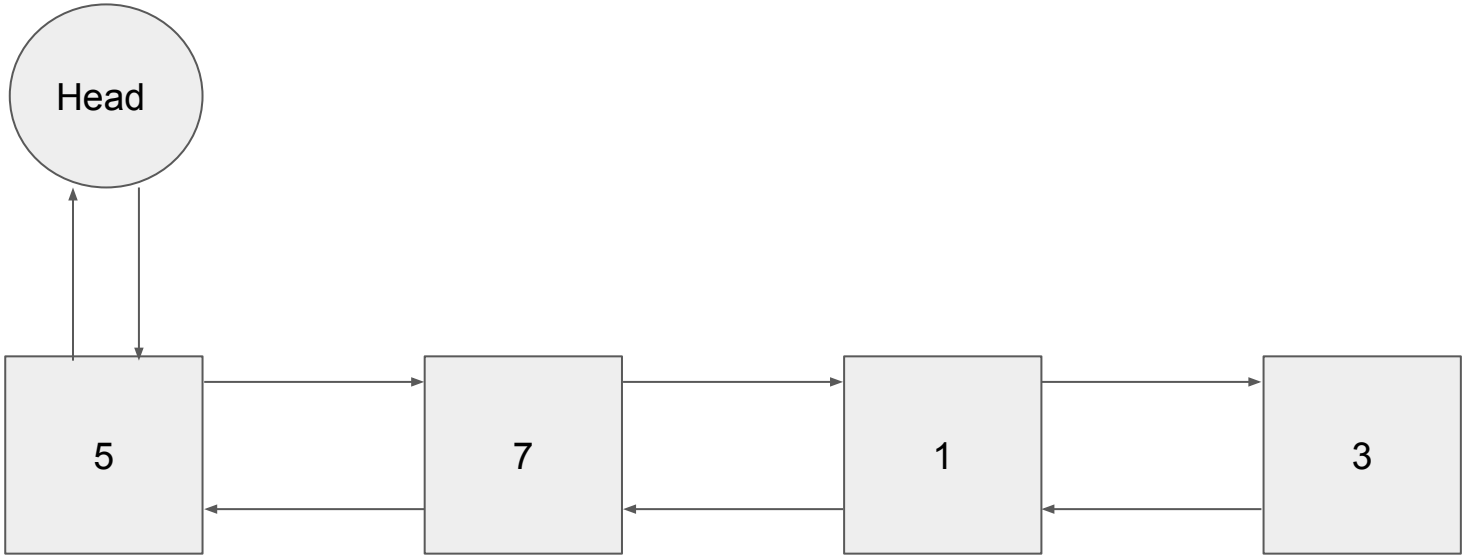
Semaphores

- Basically an *int*
  - Binary Semaphore - only 0 or 1
  - Counting Semaphore - can be any non-neg. Value
- Think of it as a special *flag* any thread can set
  - *Unlocked* when the value is >0
  - *Locked* when the value is 0

Semaphores

- Three Functions
  - *Initialize*
    - Create the semaphore with a default value of 0
  - *Wait*
    - Thread waits here until the semaphore is non-zero
  - *Post*
    - Increments the value of the semaphore

Linked List


Head
5
7
1
3

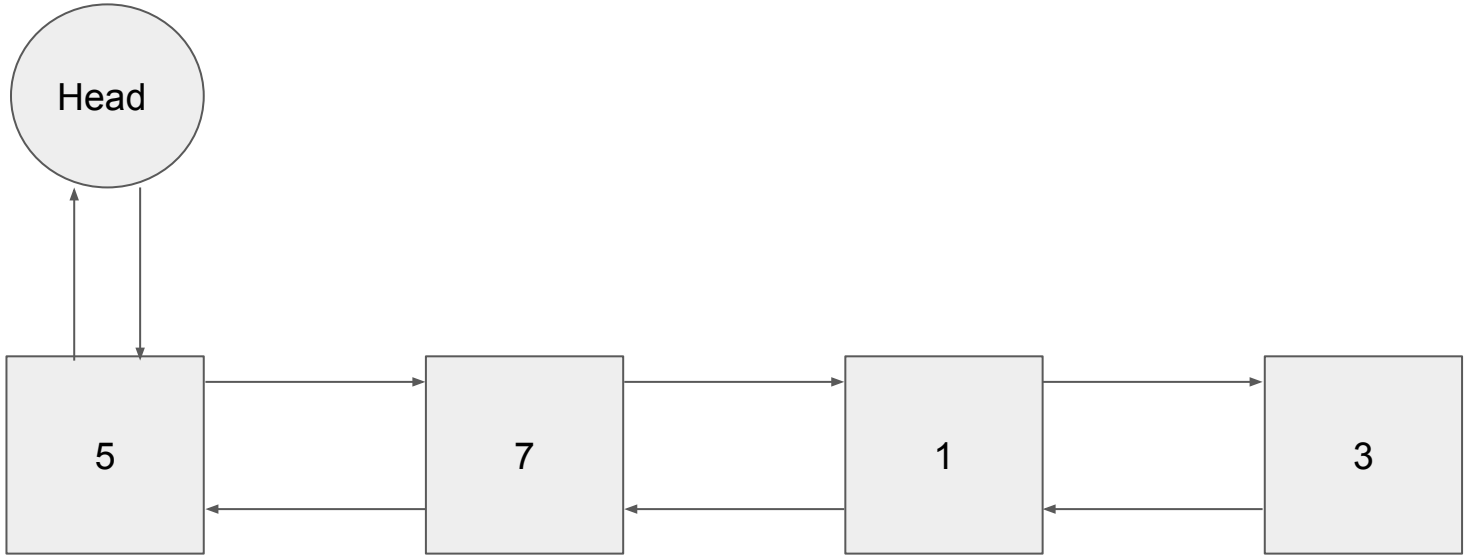
In C

```
struct Node {
    uint64_t value;
    struct Node* previous;
    struct Node* next;
}
```

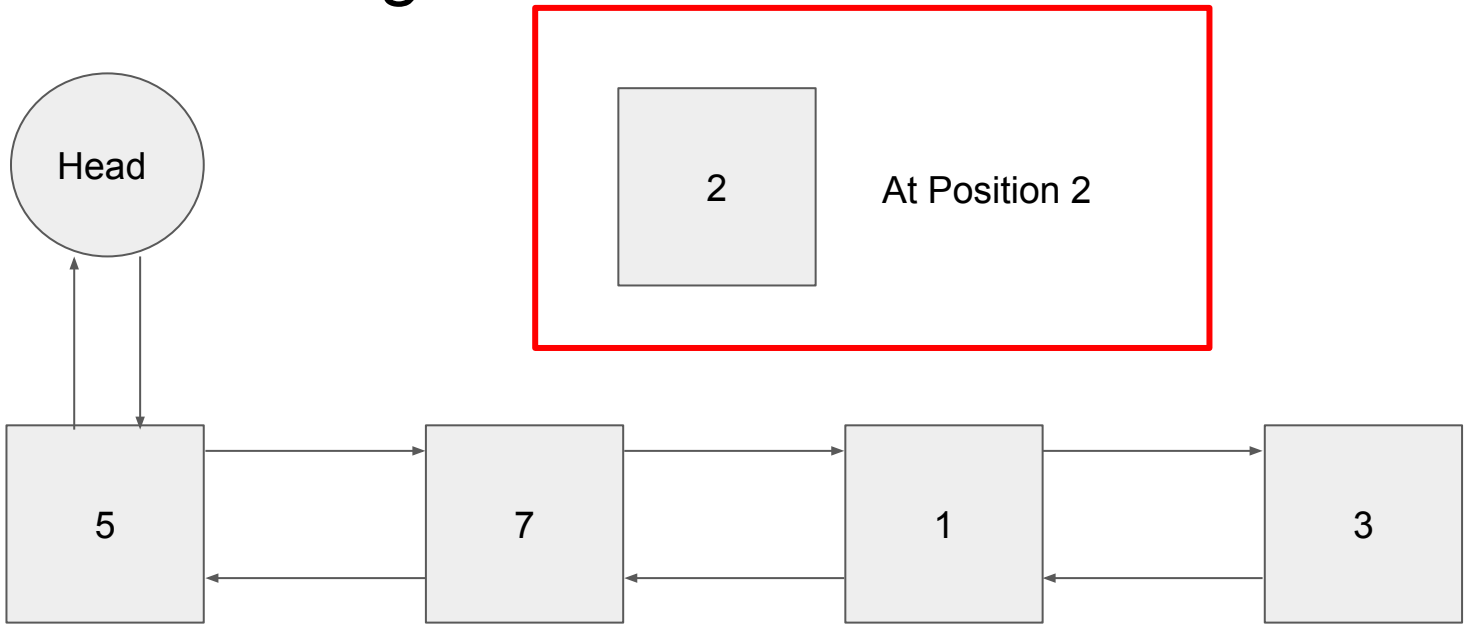
Some List Operations

- Find
  - Find whether a certain value is in the list
- Insert
  - Insert a value at the end
  - Or at any position (argument)
- Delete
  - Delete a value at any position (argument)

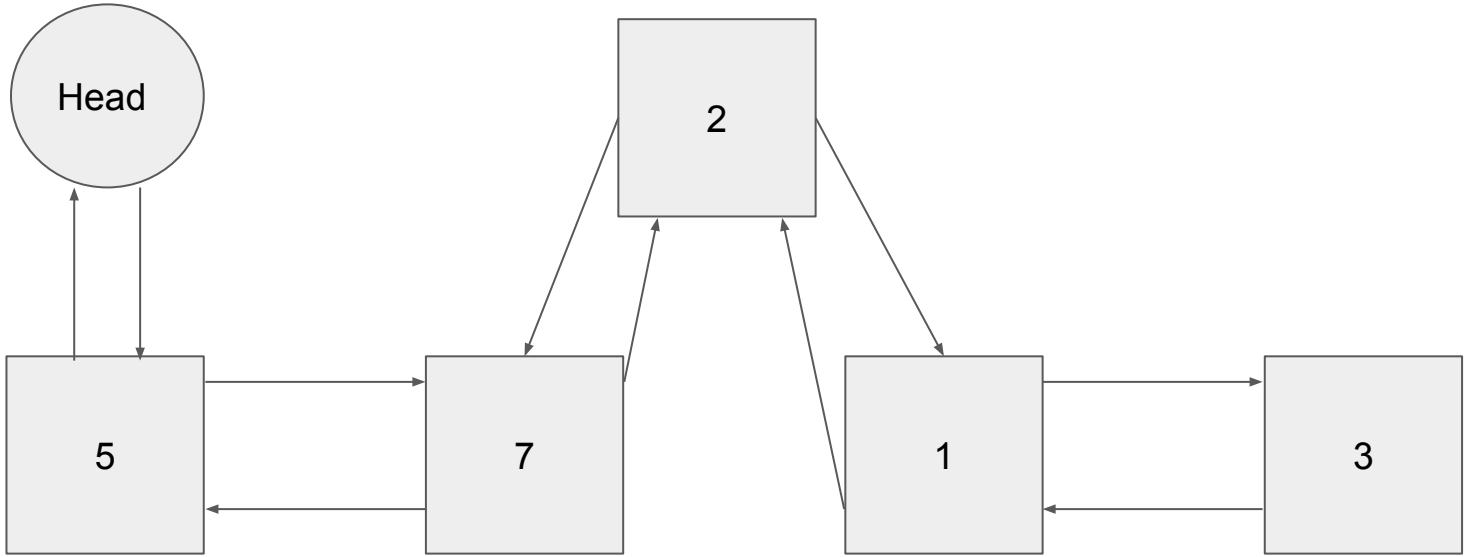
Find if something is in the linked list?



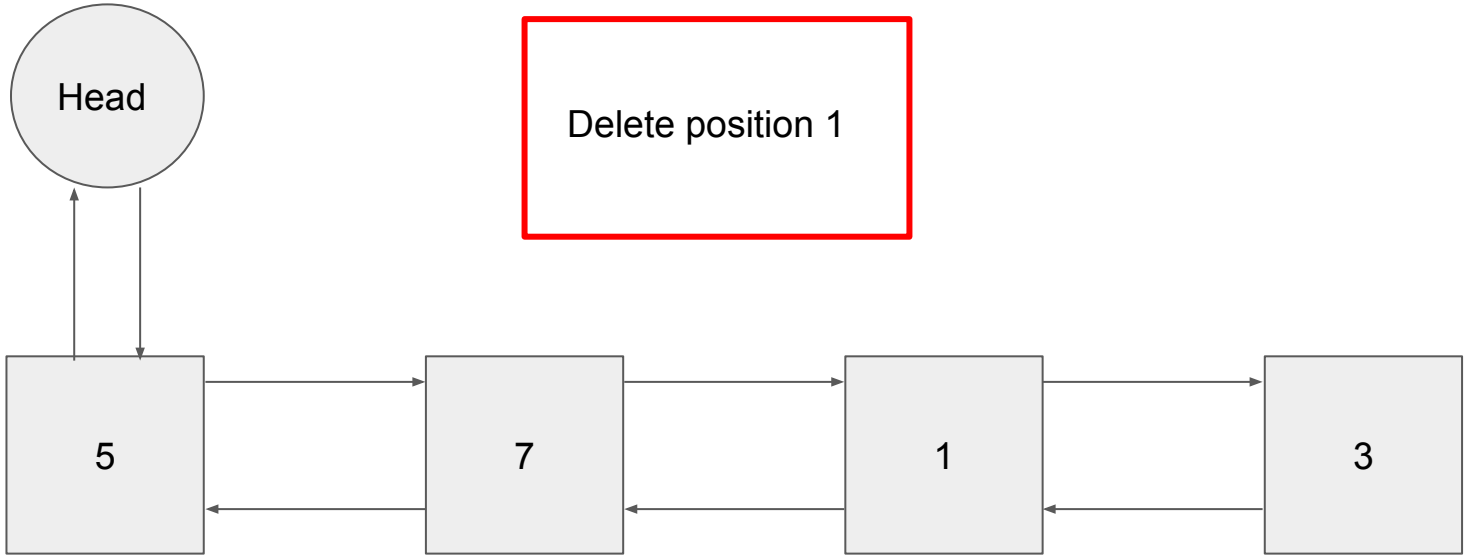
Insert Something?



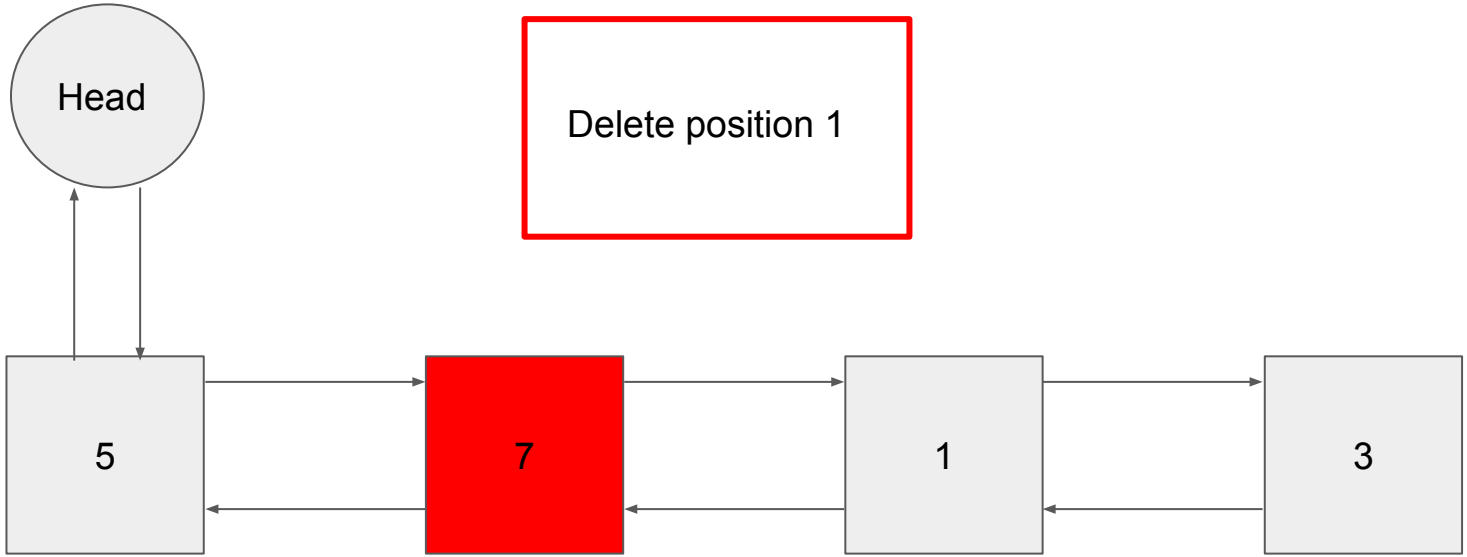
Insert Something?



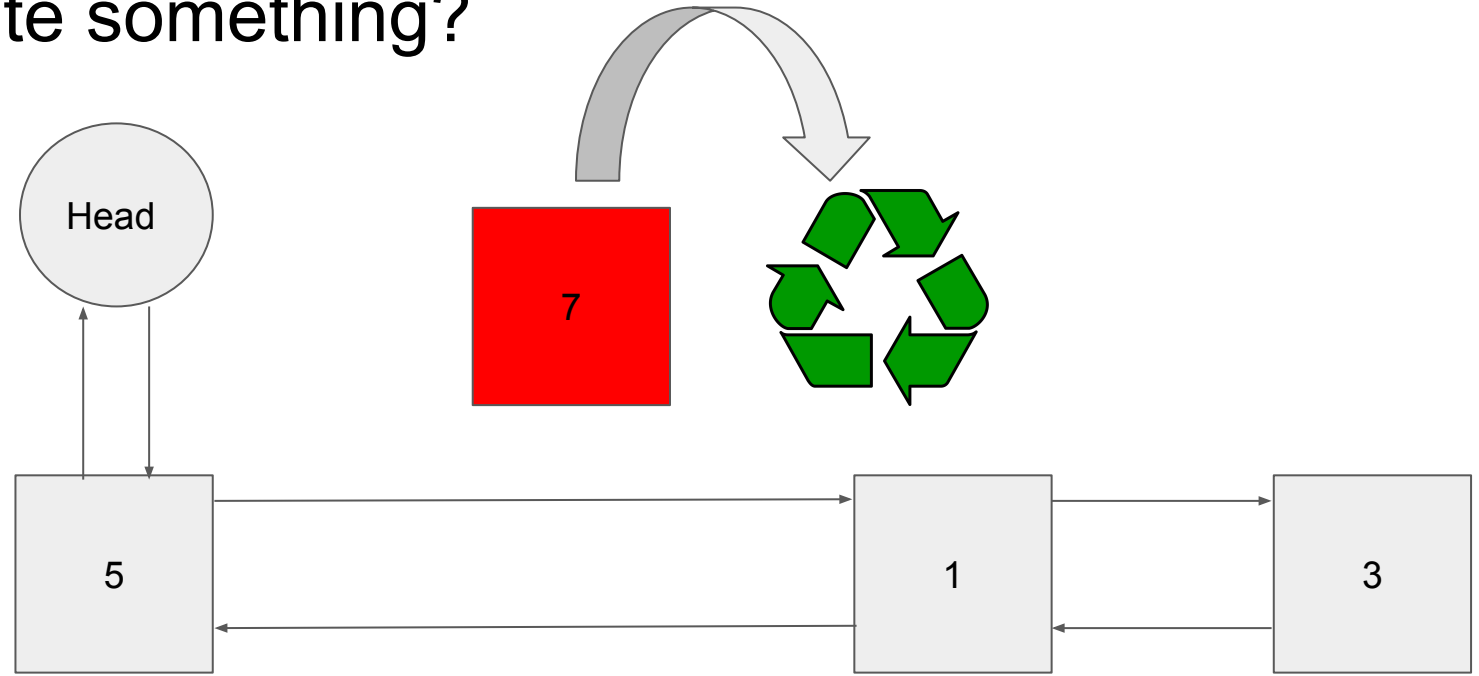
Delete something?



Delete something?



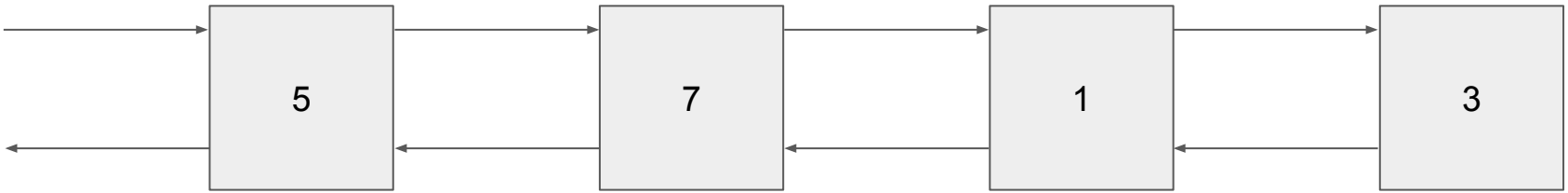
Delete something?



Let's think about these operations in parallel!

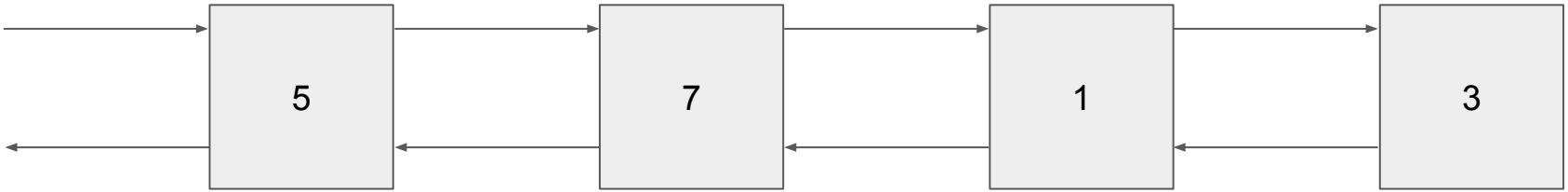
Find if something is in the linked list?

Thread 1: Find where 7 is in the list
Thread 2: Find where 3 is in the list



Insert something...

Thread 1: Insert 2 at position 3
Thread 2: Insert 8 at position 1



Insert something...

Before: [5,7,1,3]

Thread 1: Insert 2 at position 3
Thread 2: Insert 8 at position 1

After: ?

1. [5,7,1,2,3]
2. [5,8,7,1,2,3]
3. [5,8,2,7,1,3]
4. [5,7,8,2,1,3]
5. [5,8,7,2,1,3]

Insert something...

Before: [5,7,1,3]

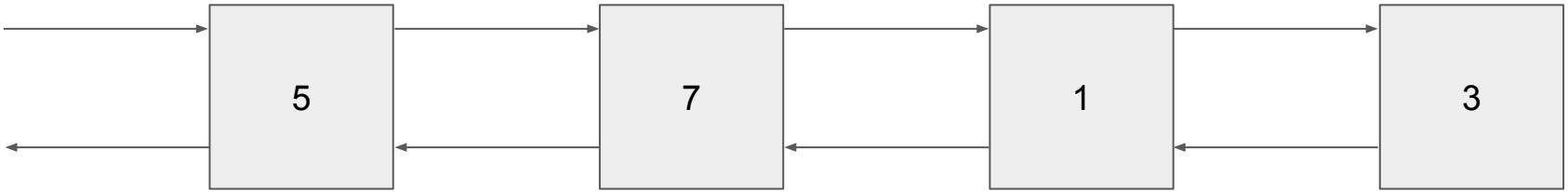
Thread 1: Insert 2 at position 3
Thread 2: Insert 8 at position 1

After:

1. [5,7,1,2,3]
2. **[5,8,7,1,2,3]**
3. [5,8,2,7,1,3]
4. [5,7,8,2,1,3]
5. **[5,8,7,2,1,3]**

Insert something...

Thread 1: Insert 2 at position 2
Thread 2: Insert 8 at position 2



Insert something...

Before: [5,7,1,3]

Thread 1: Insert 2 at position 2
Thread 2: Insert 8 at position 2

After: ?

Insert something...

Before: [5,7,1,3]

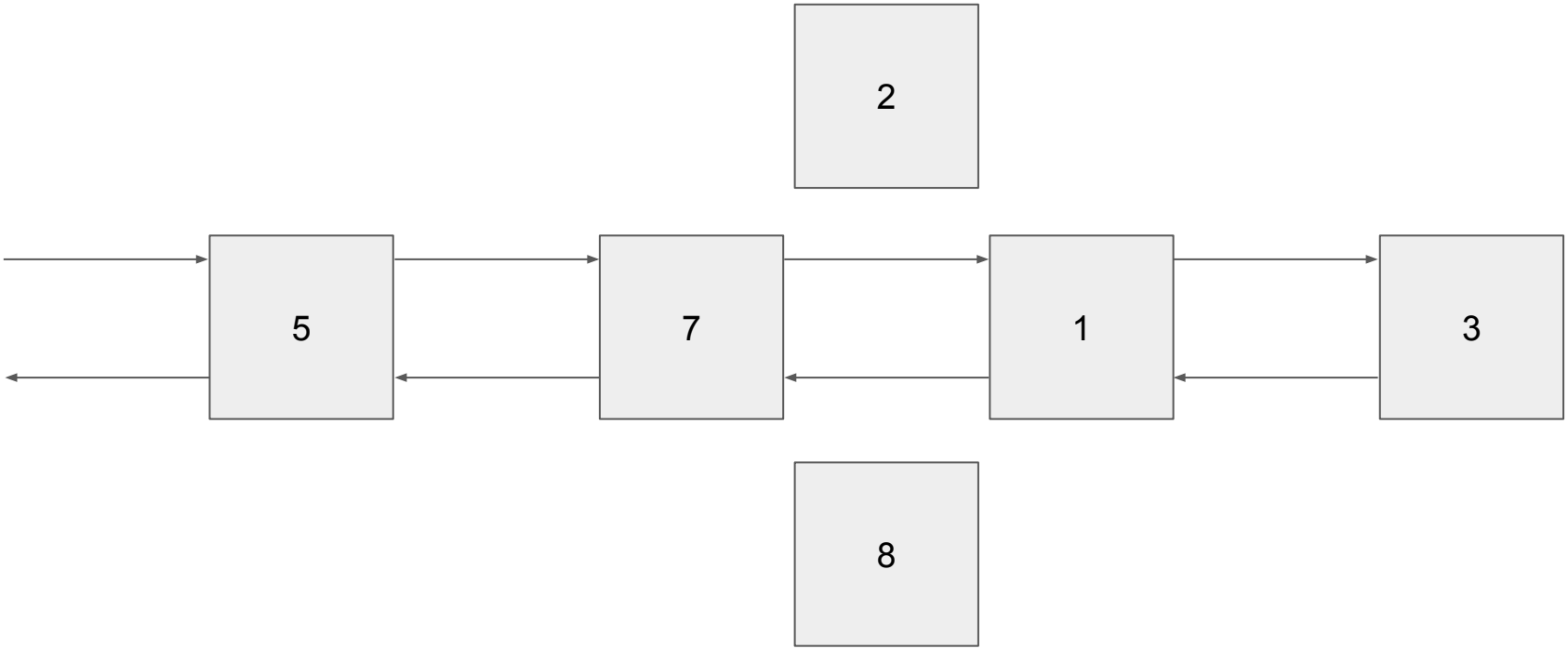
Thread 1: Insert 2 at position 2
Thread 2: Insert 8 at position 2

After:

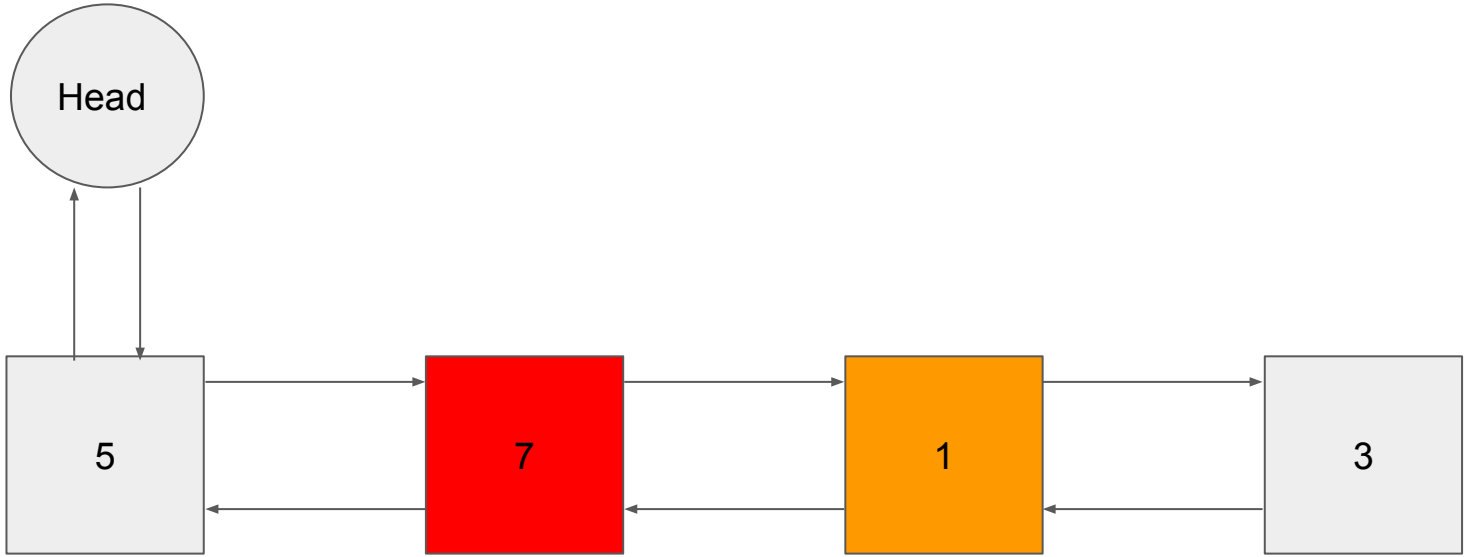
1. **[5,7,2,8,1,3]**
2. **[5,7,8,2,1,3]**

Insert something...

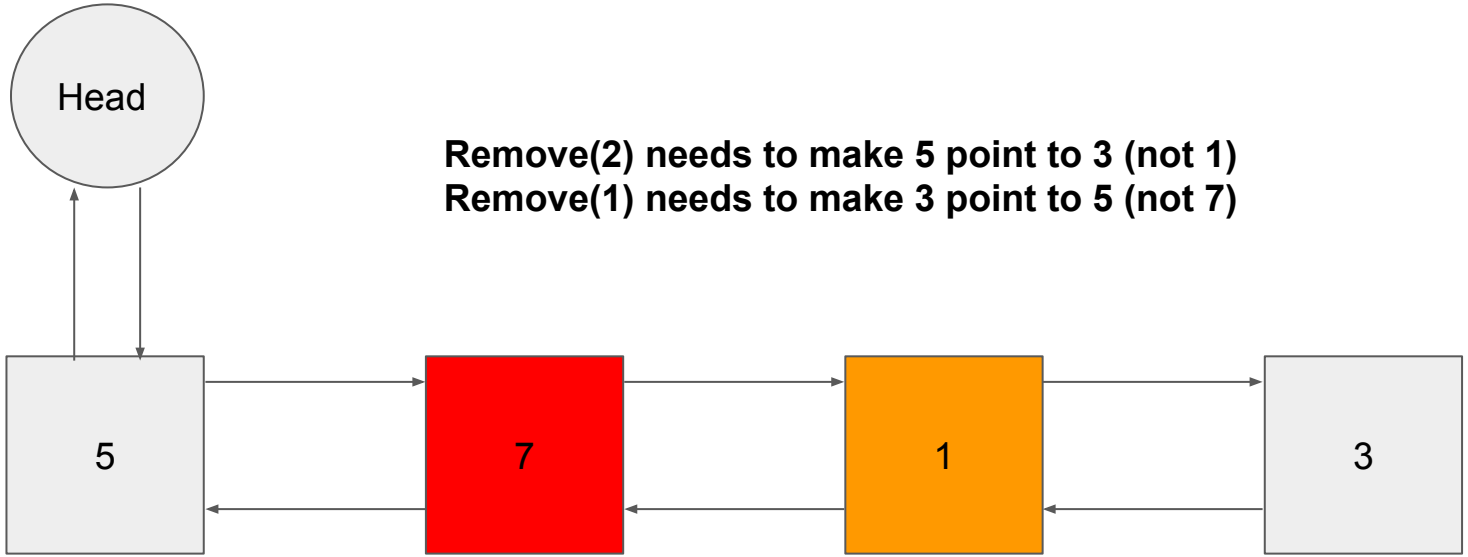
Thread 1: Insert 2 at position 2
Thread 2: Insert 8 at position 2



Delete something?



Delete something?



It's basically a mess

- Parallel access to data structures is tough
  a. Usually many threads try to access at the same time
  b. Usually, the operations take some time to run!
     - **Find** has to run many lines of code
- More code to run => More possible interweaving => More likely races!

Making Linked-List Safe

- Choices
  a. Synchronize access to individual items
  b. Synchronize the entire list
  c. Other ways!

Synchronize Individual Items

- Can be complicated!
  - Find
    - Which items does it need? (Any?)
  - Insert
    - Which items does it need?
    - Insert + Remove interaction?
  - Remove
    - Which items does it need?

Synchronize Entire List

- Only 1 thread can access the list at a time
- Too strict?
  - Insert/Remove
    - Fair enough
    - Makes *writes* to the list
  - Find?
    - **Can multiple finds happen at the same time?**
    - Only *reads*

Read-Write Lock Idea

- **Readers:**
  - Multiple threads should be able to read
  - Reading causes no problems for other readers
  - *Shouldn't coexist with writers!*
- **Writers:**
  - Only one writer should write at a time
  - Writers also can't coexist with readers
- Idea: A "stricter" lock for writers, a "nicer" lock for readers

RW Lock

```
pthread_rwlock_t lock; //Type for RW lock
pthread_rwlock_init(pthread_rwlock_t*, NULL);
//Initializes the lock
```

- Another opaque lock object
- Initialize in the same way as mutex:
  - Pointer to lock
  - Pointer to attributes

RW Lock

```
pthread_rwlock_wrlock(pthread_rwlock_t*)
pthread_rwlock_rdlock(pthread_rwlock_t*)
pthread_rwlock_unlock(pthread_rwlock_t*)
```

- Writer lock - (blocks until all readers and writers release)
- Reader lock - (blocks until all writers release)
- Unlock - (releases current lock)

A Bank

Bank

- A bank allows customers to store their money!
- Each customer has an account
- Account has some money in them
- Customers should be able to access their account

Which of these should have locks?

- Bank account:
  - Check info
  - Check balance
  - Deposit
  - Withdraw
  - Make transactions
  - Change info, etc...

Which of these should have locks?

- Bank account:
  - Check info
  - Check balance
  - **Deposit**
  - **Withdraw**
  - **Make transactions**
  - **Change info**

Deposit (pseudocode)

```
deposit_money(me, amount){
    lock(me);
    me -> balance+=amount;
    unlock(me);
}

/*
Pretend that lock(me) and unlock(me) correspond to functions which locks/unlocks a
mutex that corresponds to me
*/

//This is pseudocode because technically "me" needs to be a pointer, etc etc
```

Withdraw (pseudocode)

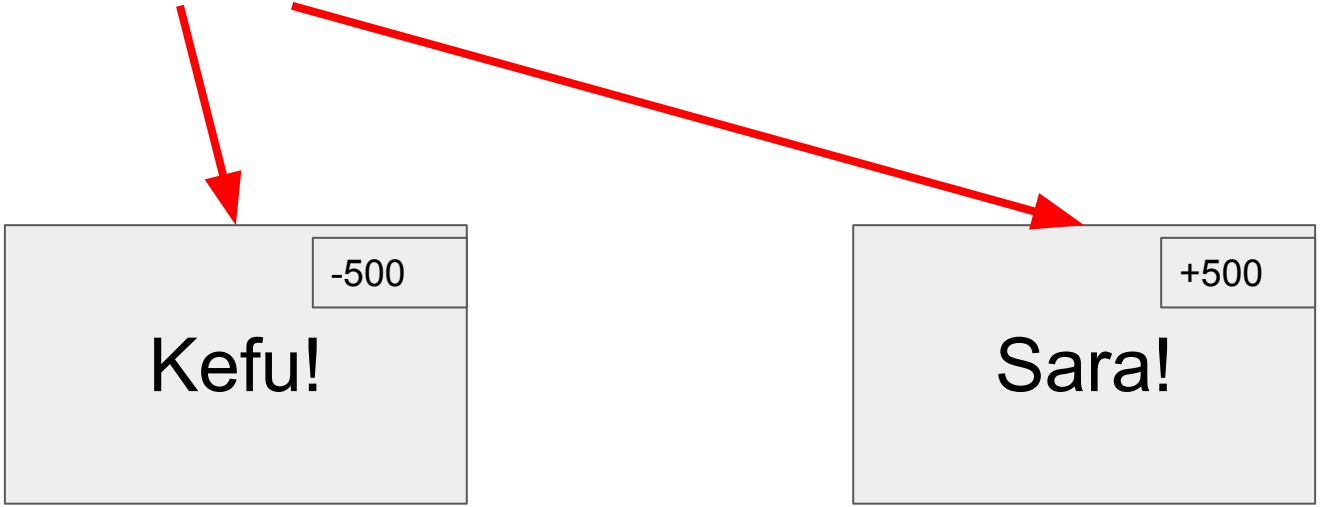
```
withdraw_money(me, amount){
    lock(me);
    me -> balance-=amount;
    unlock(me);
}

/*
Pretend that lock(me) and unlock(me) correspond to functions which locks/unlocks a
mutex that corresponds to me
*/
```

Transfering money (pseudocode)

```
transfer_money(me, other, amount){

    lock(me);
    lock(other);
    me -> balance -=amount;
    other->balance += amount;
    unlock(me);
    unlock(other);

}
```

Transfer!

```
transfer(kefu, sara, 500);
```



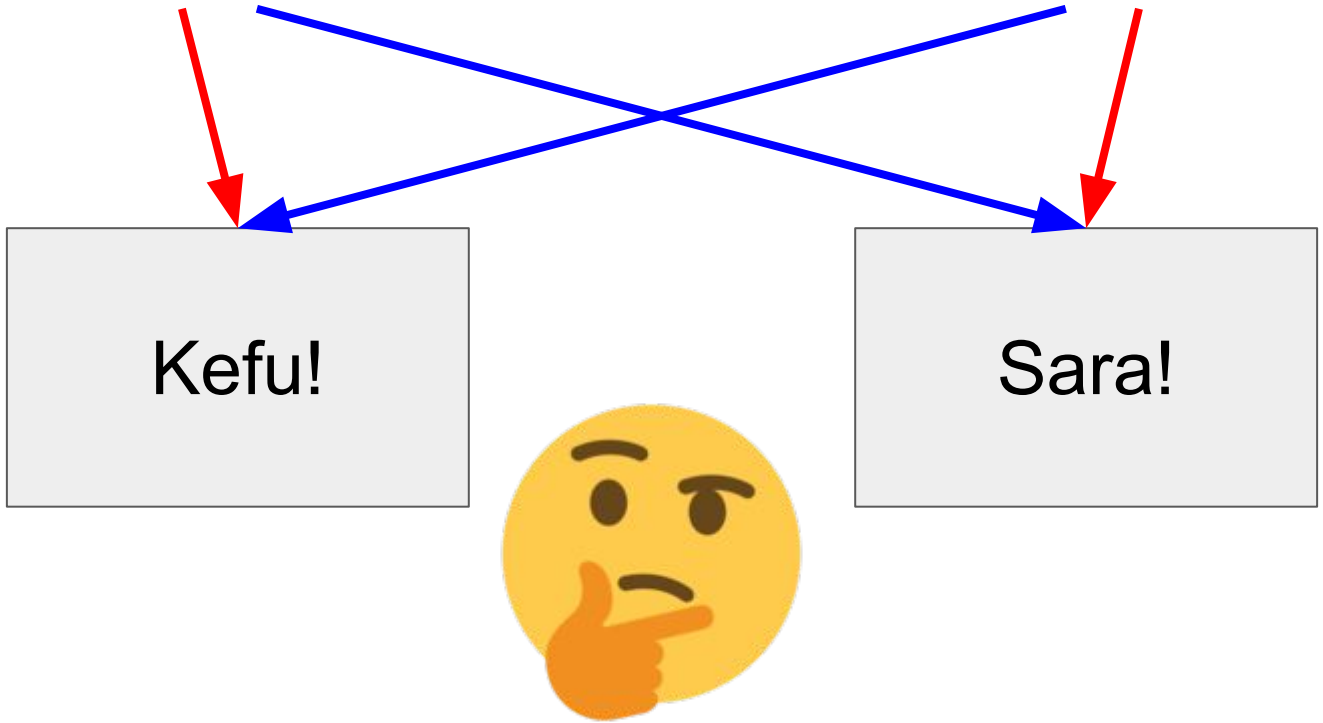
Is there a problem?

```
transfer_money(me, other, amount){
    lock(me);
    lock(other);
    me -> balance-=amount;
    other->balance += amount;
    unlock(me);
    unlock(other);
}
```

Hmmmm….

```
transfer(kefu, sara, 500);
```

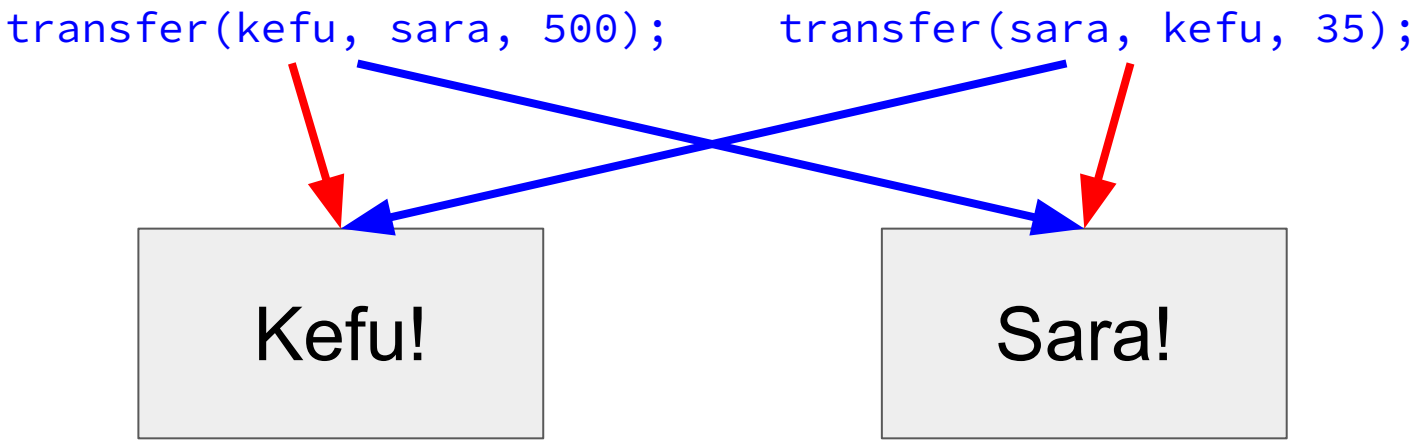
```
transfer(sara, kefu, 35);
```



Deadlock

- When a set of threads are waiting for some resource to become available
  - But another thread is holding the resources!
  - The other thread is also waiting for resources…

Deadlock



- Here:
  - Transfer 1 has *Kefu's lock* but needs *Sara's* lock
  - Transfer 2 has *Sara's lock* but needs *Kefu's* lock

Deadlock

- Threads will just wait indefinitely
- Four conditions:
  - **Mutual Exclusion**
  - **Hold and Wait**
  - **Circular Waiting**
  - No Preemption

Conditions

- **Mutual Exclusion**
  - There are some resources that only 1 thread can hold at a time
  - The mutex/lock
- **Hold and Wait**
  - Threads can demand more resources when already having some
  - Need another lock for a diff. item

Conditions (2)

- **Circular Waiting**
  - (non-technical definition)
  - There is some pattern of resource demands that forms a cycle
- **No Preemption**
  - Threads cannot be forced to give up resources
  - Only the thread itself can "unlock"

Conditions (3)

- ***If*** all four conditions are met, ***then*** there is a chance of deadlock
- ***"There is a chance"***
  - Doesn't *always* happen
  - Might be very hard to trigger
  - Over many executions, it *will* definitely happen

Avoiding Deadlocks

The (Common) Lazy Strategy

The Lazy Strategy

- Ignore the possibility
- Just restart the system once it freezes!
- …
- But let's talk about the non-lazy strategies...

In theory, easy!

- Four necessary conditions:
  - Mutual Exclusion
  - Hold and Wait
  - Circular Waiting
  - No Preemption

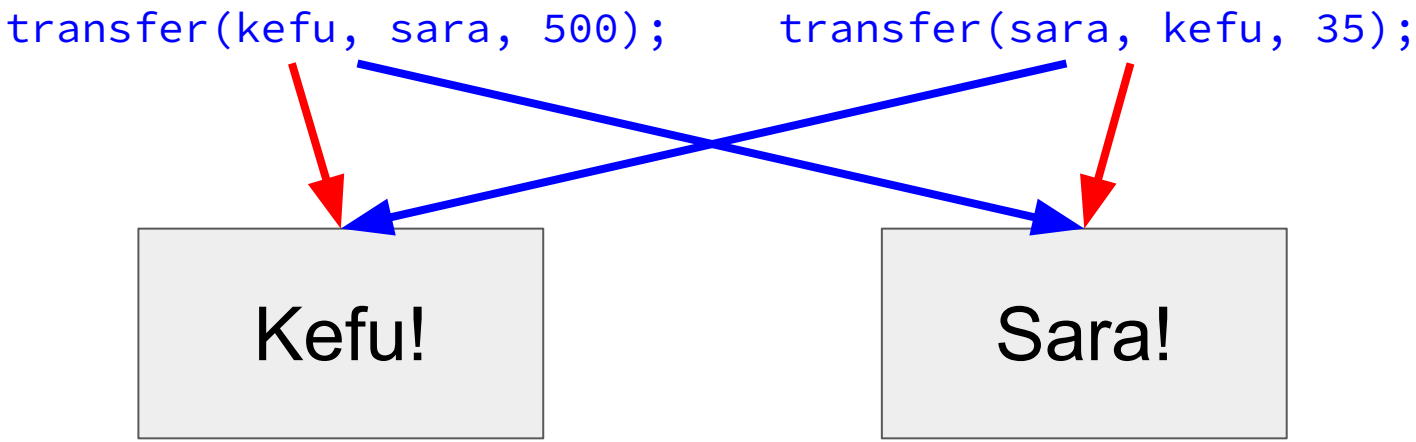
"Hold and Wait"

- Don't allow threads to grab more than a single lock!
  - **Restructure your code to only need one lock at a time!**
- This definitely avoids deadlock!
- But in some cases:
  - Impossible to do
  - Undesirable to do

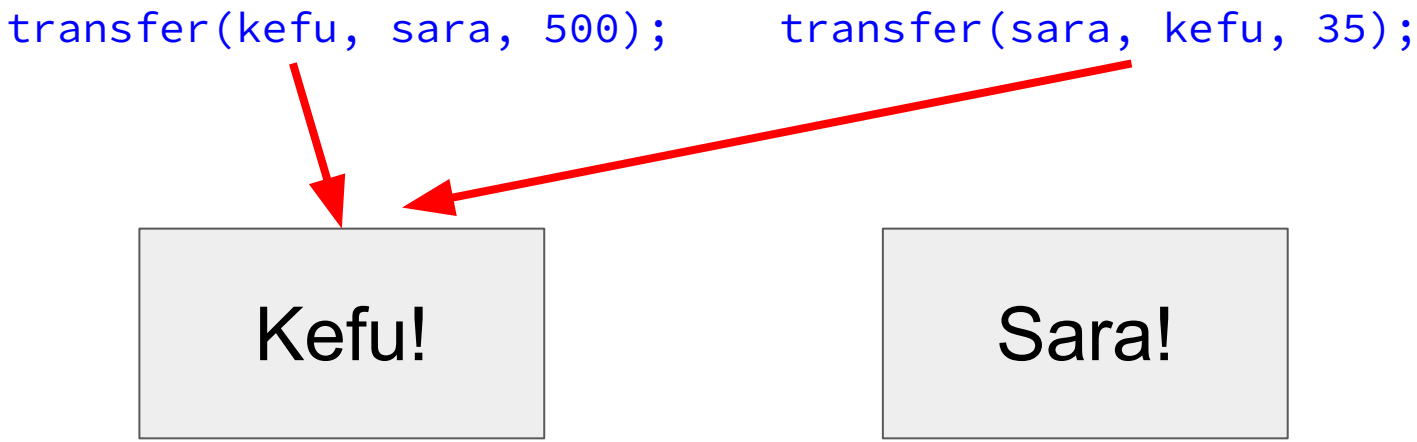
For our case:

```
transfer_money(me, other, amount){
    lock(me);
    me -> balance-=amount;
    unlock(me);

    lock(other);
    other->balance += amount;
    unlock(other);
}
//In terms of code this is probably okay
```

Circular Waiting



Ordering



- Idea:
  - Impose an ordering on which locks to get!

Idea for an ordering...

- Make each account have an *account number*
- When transferring between accounts:
  - Still grab both locks before the transfer
  - Always go for the lock for *lower number* account first

For our case:

```
transfer_money(me, other, amount){
    first = Smaller(me, other);
    sec = Bigger(me, other);
    lock(first);
    lock(sec);
    me -> balance-=amount;
    other->balance += amount;
    unlock(first);
    unlock(sec);
}
```

Ordering

- ***This can get very complicated if threads need more than 2 locks!***

In Summary

- Locks are good!
- Locks also cause problems…
  - Deadlock
    - Know the 4 conditions
    - Know the 2 common ways to avoid
  - *Performance*

Beyond Pthreads

CSCI 320

Upcoming Work

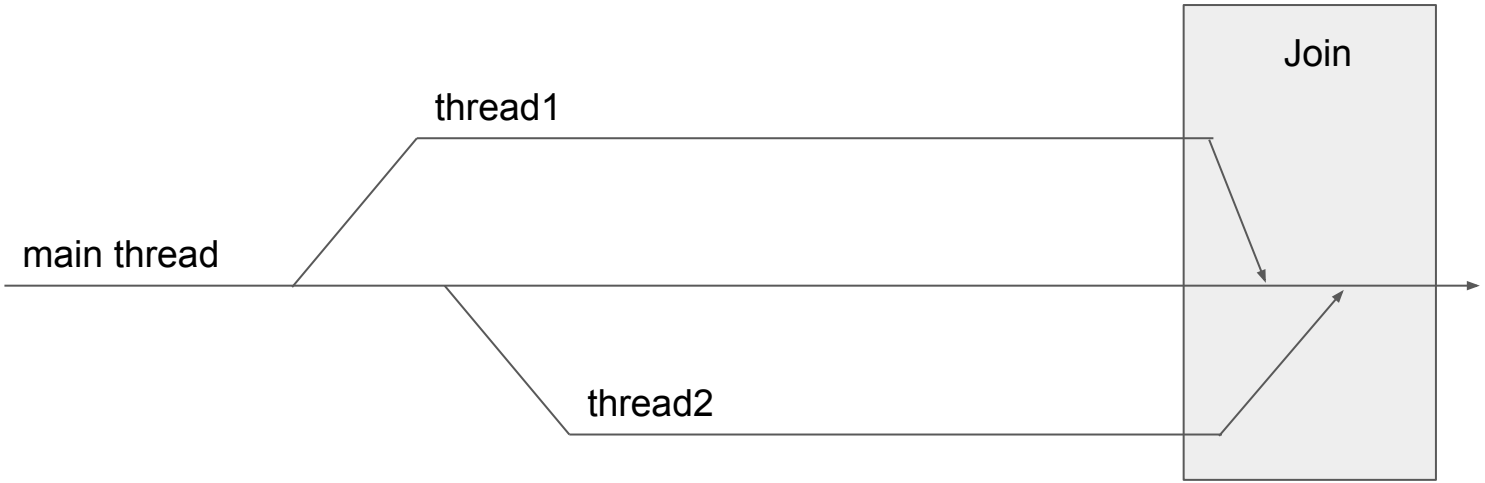
- Homework 3?
- Homework 4?
- Exam on Friday of next week
  - (Right before break!)

Pthreads Recap

Thread Creation

```
pthread_create(...)

pthread_join(...)
```

- Two main operations:
  - **Spawn, or *create***
  - **Join, or *synchronize***


Join
thread1
main thread
thread2

```
//Global variables are "shared"
double total = 0;
int thread_count = 10;
uintmax_t terms = 100000;


//Threads run functions
void* partial(void* rank){
     ...
}
```

Also, importantly...

```
uint64_t sum = 0;

//Function with void*

void* Sum(void* ptr){

    uintptr_t rank = (uintptr_t) ptr;

    int64_t block = NUMBERS/THREADS;

    for (int64_t i = block*rank; i < block*(rank+1); i++){

        sum += i;

    }

    return NULL;

}
```

Races are bad! Locks are useful!

Mutex Lock

```
pthread_mutex_init(pthread_mutex_t* lock_var,...)
pthread_mutex_lock(pthread_mutex_t* lock_var)
pthread_mutex_unlock(pthread_mutex_t* lock_var)
```

- Use of locks
  a. Lock
  b. Critical Region
  c. Unlock

Semaphores

```
sem_init(sem_t* sem_p, int shared, unsigned value)
sem_wait(sem_t* sem_p)
sem_post(sem_t* sem_p)
```

- Allows sending simple signals

The downsides of locks?

- Correctness:
  - Deadlocks
    - 4 conditions
    - Avoidance
- ***Performance...***

About Pthreads

- Advantages:
  - Relatively lightweight
  - Simple, easy to use
  - Ideas are unchanged in other threading libraries:
    - Java Threads
    - C++ <Threads>

Disadvantages:

- Very *manual* in some sense
- Missing advanced features

Overview

- Past weeks:
  - Programming
  - Pthreads
  - Locks
- We learned
  - How to create a parallel program
  - How to identify unsafe races
  - But **not** how to *design and analyze* efficient programs

Overview (2)

- Next two weeks:
  - Theoryland!
  - Goal: Learn how to analyze parallel programs!
    - Runtime analysis
    - Design ideas
- Today:
  - I'll bring some of the *questions*
  - We'll figure the *answers* slowly over time

Preparation:

- Do you know the basics of Big-O notation?
  - (You don't need to know the formal definition that has c_0 and "for sufficiently large numbers" etc…)
  - What does it mean for a program to run in say…
    - O(n) time?
    - O(n^2) time?
- If these terms are completely alien to you, please try to review a bit of the basics

Gaps in Knowledge (so far)

Performance

```
//Pseudocode for a Parallel Sum

sum = 0;
n = 10000;

Spawn 10 threads to do partial sums
...
Join

//Why spawn 10 threads? Not say, 20? 40?
```

Why use more threads than processors?

```
//Pseudocode for a Parallel Sum

sum = 0;
n = 100000000000000;

Spawn thread 1 for 0...100000
Spawn thread 2 for 100001...200000
Spawn thread 3 for 200001...300000
...
Join
```

Load?

```
Spawn thread 1 for 0...100000
Spawn thread 2 for 100001...200000
Spawn thread 3 for 200001...300000
```

- Is it more efficient to say, split up the sum differently?

Is it the programmer's job?

- To make sure work between threads are balanced?
  - Yes:
    - It let's you eke out extra performance
    - Allows fine-tuning!
    - Programmers should be responsible for everything!
  - No:
    - Depends on knowledge of the *specific computer*
    - Looks really hard for complicated programs!
    - The "runtime system" should figure it out...

On that topic…

For Pthreads, your opinion doesn't matter!

Cache:

```
Spawn thread 1 for 0...100000
Spawn thread 2 for 100001...200000
Spawn thread 3 for 200001...300000
```

- Can 0...100000 fit in the cache for the CPU?
- **Suppose cache can only fit 1000 numbers…**
  - Wouldn't it be better to have smaller blocks?

Cache (2):

```
Spawn thread 1 for 0...100000
Spawn thread 2 for 100001...200000
Spawn thread 3 for 200001...300000
```

- **Suppose cache can fit can fit 100000 numbers…**
  - Now, having blocks of 1000 seems bad!
    - More threads needed
    - Creating a thread *does* cost something!

Locks...

...

//What do locks do to performance?

```
void* Sum(void* ptr){

    uintptr_t rank = (uintptr_t) ptr;

    int64_t block = NUMBERS/THREADS;

    for (int64_t i = block*rank; i < block*(rank+1); i++){
            lock(sum_lock);
            sum += i;
            unlock(sum_lock);

    }

    return NULL;

}
```

Locks...

- **Only one thread can execute the critical region at a time….**
- Hold on…
  - Doesn't that make having multiple threads useless?

Recall: Threading in Python

- You can actually do multithreading in Python
- Python uses a Global Interpreter Lock (**GIL**)
  - This is a *mutex* on the **interpreter**
  - Only *one* thread may execute a line of code at a time!
- Why was this done?
  - To prevent race conditions! :)

Summary

- We are now pretty much done with *Pthreads*
- **Know how creating/joining threads work!**
  - Consistent in most languages
  - **Remember this!**
- **Know what race conditions are!**
- We will next learn how to:
  - **Identify specific performance issues**
  - **Analyze Parallel Programs**

Task Graph

CSCI 320

Analysis

```
//Sum up the first n numbers
//What is the running time of my code?

int64_t sum = 0;
for (x = 0; x < n; x++) {
   sum = sum + x;
}

printf("Total is %d \n", sum);
```

Analysis

- Taking the sum of *n* numbers sequentially is O(n)
- Suppose we write a parallel program…
  a. The program will look very different!
  b. Suppose we use *p* processors…
  c. Is it O(*n/p*)?
     - What does this imply?

```
//Pseudocode for mix

main(){

   Do some sequential O(n) stuff.
   Spawn 10 threads to do some stuff…
   Do some sequential O(n log n) stuff
   Join 10 threads

}
```

```
//Pseudocode

threadfunction(int size)
   if size > 10 {
      Recursively spawns 2 "smaller" threads
   }
   Join the threads...
}
```

Questions to consider:

1. How much faster can my program get if I incorporate parallelism?
2. How to describe runtime for parallel programs?
3. How to determine which parallel programs are faster?

Amdahl's Law

- Roughly: *Unless **most** of a program is parallelized, the possible speedup is small.*
- Consider program with a **parallel** part and a **sequential** part
  - Let's even assume the parallel part gets linear speedup
  - Let T_s be the time for the sequential
  - Let T_p be the time for the parallel part

Amdahl's Law (2)

- Program
  - Suppose T_s takes 5 seconds
  - Suppose T_p takes 20/p seconds (linear speedup)
- Then, on 1 processor this program takes *25* seconds
- On 10000 processors, this program still takes 5 seconds
- This is only ~*5* times faster… on effectively infinite processors - not very good!

Amdahl's Law (3)

- 5 times speedup is still speedup!
- Good parallel programs *scale* nicely based on problem size:
  - The sequential part is a constant, fixed cost
    - Creating threads
    - Setting up memory
  - The parallel part has a large increase in cost as the problem grows!

Amdahl's Law (4)

- Program on input of size 1000
  - Suppose T_s takes 5 seconds
  - Suppose T_p takes 20/p seconds (linear speedup)
- Program on input of size 10000000
  - Suppose T_s takes 7 seconds
  - Suppose T_p takes 200000/p seconds (linear speedup)
- **Therefore, it is important to know how to analyze the asymptotic running time of parallel programs!**

Analyzing Running Times

- We eventually want to analyze the performance of a parallel program
- Key ideas:
  - We must figure out how to "count" the runtime
  - We must have a good way to model the execution of the program

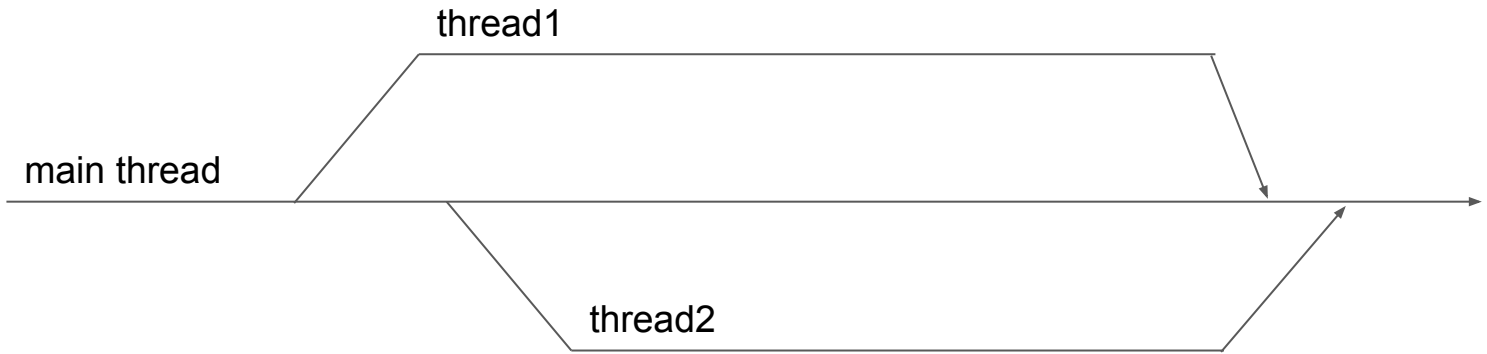
Sequential Programs

```
//Sum up the first n numbers
int64_t sum = 0;                    //O(1)
for (x = 0; x < n; x++) {           //Loops O(n) times
    sum = sum + x;                  //O(1) operation in loop
}
```

- Count each line
- Count execution times for lines
- Add them together
- One long "linear strand" of execution

One linear strand of execution

Parallel Programs Have Branches!

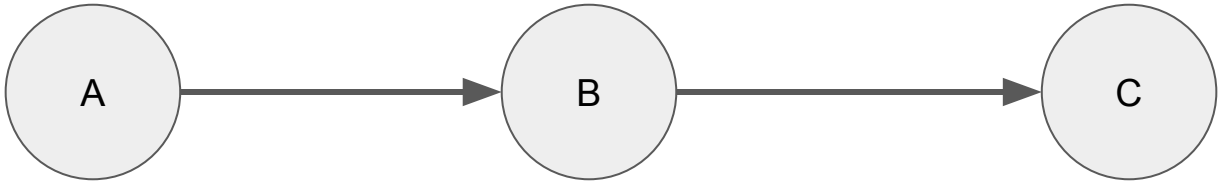


The Task Graph

Graphs

- Nodes
- Edges
- Directed Graph
  - Each edge has a direction

Directed Graph

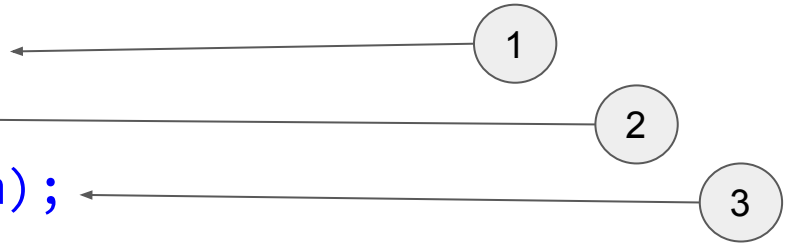


Modelling Programs

- We will model programs as Directed Graphs
- Nodes
  - "Work" or instructions
- Edges
  - Some sort of "ordering" on the instructions
- *We will represent the sequence of operations of the program*

Example

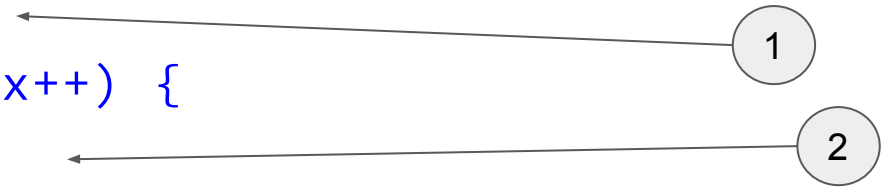
```
int64_t sum = 0;
sum = sum + 10;
printf("%d\n", sum);
```



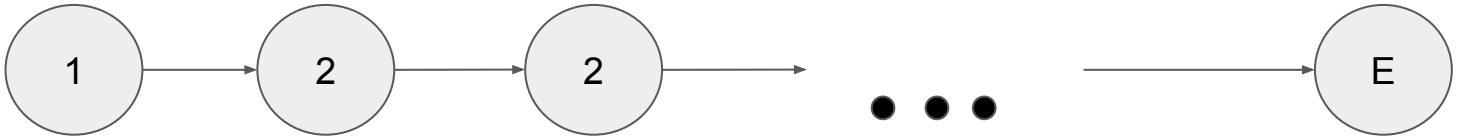
Program:

The Loop Program

```
int64_t sum = 0;
for (x = 0; x < n; x++) {
    sum = sum + x;
}
```



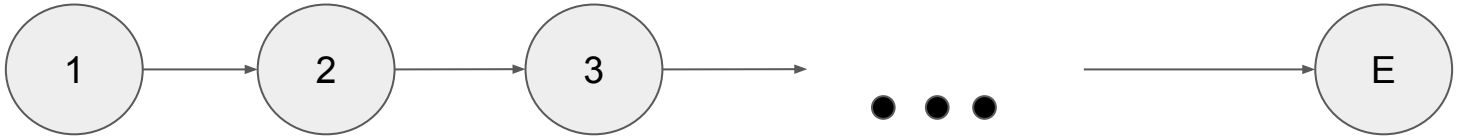
Program:



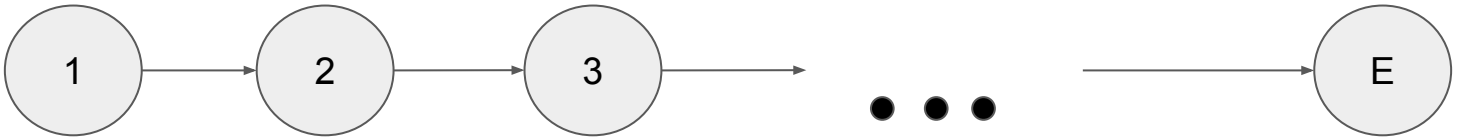
Modelling

- Each node is exactly 1 unit of work
  - We'll relax this later
  - (It becomes tedious to draw otherwise)
- Edges represent *precedence constraints*
- The *parent* of a node must be done before the node itself can be done

All Sequential Programs



What is the running time of this program?



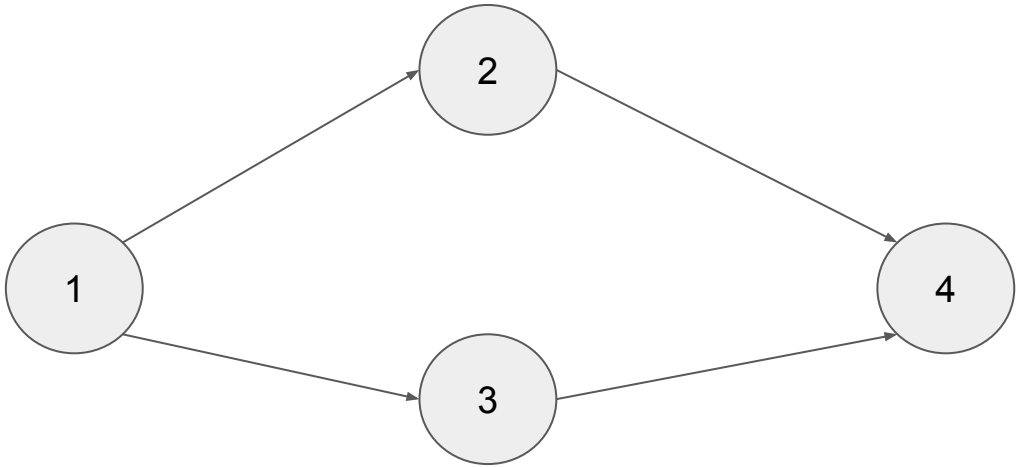
Sequential Programs

- A sequential program is just a linear directed graph
- Running time:
  - The total number of nodes in the graph
  - (In the case of non-unit work nodes)
    - Sum of the total work of all nodes in the graph

A Parallel Program

```
int64_t sum = 0;                                  //1
Spawn Thread 1 to sum 0 through 10001.            //2
    for (x = 0; x < 10001; x++) {
        part = part + x;
    }
Spawn Thread 2 to sum 10001 to 20001.             //3
    for (x = 100001; x < 20001; x++) {
        part = part + x;
    }

Join.  Combine answers.                           //4
```

A Parallel Program



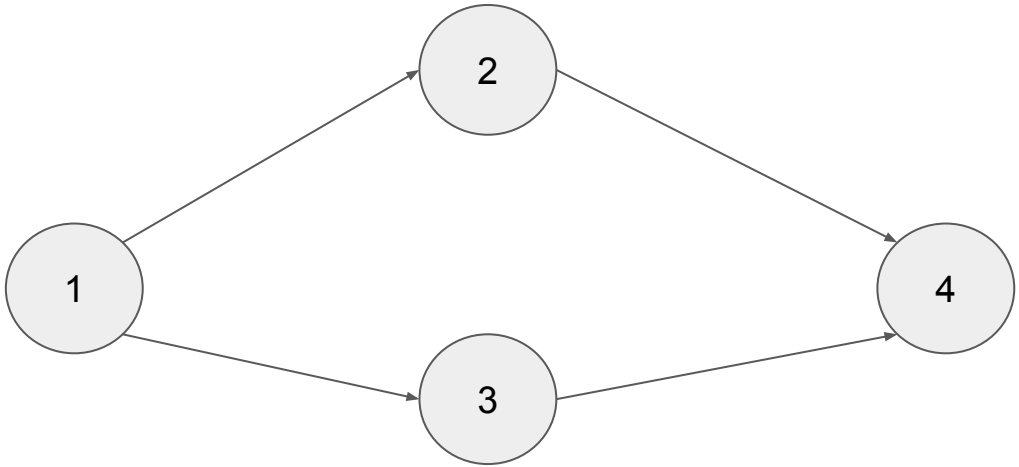
Parallel Programs

- Can also be expressed as a graph:
  - **Directed Acyclic Graph** (DAG)
  - This is a graph of the tasks/work!
  - *NOT* a graph of the flow of code
- **The Task Graph** or The Work Graph

Parallel Programs (2)

- The program is considered *finished* when all of its tasks have been executed
- Tasks may only be done if **all of its predecessors** have been done
- Nodes that do not have a predecessor/successor relationship **may be done** in parallel

A Parallel Program



A Parallel Program

```
int64_t sum = 0;                                  //1
Spawn Thread 1 to sum 0 through 10001.            //2
    for (x = 0; x < 10001; x++) {
        part = part + x;
    }
Spawn Thread 2 to sum 10001 to 20001.             //3
    for (x = 100001; x < 20001; x++) {
        part = part + x;
    }

Join.  Combine answers.                           //4
```

To Think About

- So, what is the *running time* of the parallel program?
- How is that represented on the task graph?