CSE 333
Section 9

Threads, *Pr-*, and Concurrency-*ocesses*

Logistics:

Due tonight :

HW4 @ 11:59 pm (Can use two late days)

Course Evaluations:

Due Friday (8/21) @ 11:59 pm. Please do them <3

Section Objective

There is no final exam for this quarter. So why be in section today?

Threads are really important, (and *really* cool!)

We do not want you to memorize the pthreads api.

We hope that you leave section today with a better high level understanding of threads and concurrency.

Terminology

- **Process**

  The execution environment of a program
- **Thread**

  Some sequential execution of code (Contained within a process)
- **Concurrency**

  Making progress on multiple tasks over the same period of time.
  (Don't have to wait for old tasks to finish before working on next)
- **Parallelism**

  Doing multiple tasks at the same time (e.g. on multiple CPUs)

“Computers are really dumb. They can only do a few things like shuffling around numbers, but they do them really really fast so that they appear smart.”

-Hal Perkins

Threads are just a way of making computers appear to do multitasking, *regardless of whether they are running one or more* CPUs

Threads

- Everything except the stack is shared
- Typically done with POSIX pthreads (C++11 also added thread objects)
  - pthread_create - “Go do this {function}”
  - pthread_exit - “I’m done with my task!”
  - pthread_join - “I’ll wait for you to report back your result”
  - pthread_cancel - “I changed my mind, you can stop now”
  - pthread_detach - “You’re free now, go forth and prosper”
- Faster context switch
- Easy communication (put something in shared memory)
- Synchronization often uses locks (like mutexes)

Processes

- Each has its own separate address space
- File descriptors are inherited from parent (sockets, stdin, etc)
- Created using `fork()` - the only function that returns twice!
  - Child gets 0
  - Parent gets new pid (process id) of child
- Get status of children with `waitpid(...)`

Threads vs Processes

	Multiple Threads	Multiple Processes
Memory / Address space	Shared	Separate
- Stack	Each thread has its own	One stack per contained thread.
- Heap	Shared by multiple threads	Independent Heap for each Process
Resources (e.g. file descriptors)	Shared	Unique copies
Communication	Easy	Difficult
Synchronization	Difficult	N/A
Robustness	One crashes, all crash	Independent of each other

Exercise 1

a) List some reasons why it's better to use multiple threads within the same process rather than multiple processes running the same program

b) What benefits could there be to using multiple processes instead of multiple threads?

c) Which registers will for sure be different between two threads that are executing different functions?

d) How does the OS distinguish the threads?

Exercise 1

a) List some reasons why it's better to use multiple threads within the same process rather than multiple processes running the same program

Processes are more expensive, since they need their own address space.
Threads are more lightweight.

b) What benefits could there be to using multiple processes instead of multiple threads?

Memory safety and (possible) crash tolerance. Processes can’t overwrite each other’s work because they don’t share an address space. Multiple processes can keep running independently if one crashes (depends of the task), whereas one thread seg faulting could crash the whole program.

Exercise 1

c) Which registers will for sure be different between two threads that are executing different functions?

The stack pointer is guaranteed to be different, since threads have their own stacks.

The program counters run independently, but might hold the same value if two threads are running the same function.

d) How does the OS distinguish the threads?

Thread IDs. The OS will track its own data about threads, including the current register states, and the pthread_t type is used as an identifier from the user program (similar to how a file descriptor identifies a file or socket).

Threads - Quick Check

```
MyClass onTheStack;
pthread_t child;
pthread_create(&child, nullptr, foo, &onTheStack);
```

onTheStack is on the parent thread's stack. However, each thread has its own stack! Can we still access onTheStack from the child? Why or why not?

Yes! All threads share an address space

Threads - Gotchas

- Resources (heap-allocated storage, file descriptors, etc)
  - Often shared between multiple threads
  - Must be allocated / deallocated *exactly once*
  - Don’t use deallocated resources from other threads

```
buf = new int[BUFSIZE];
...
if (!handleRequest(buf, req, len)) {
    delete[] buf; // buf was allocated in this thread
    close(fd); // is somebody else going to try to use fd???
    pthread_exit(nullptr);
}
```

Threads - Gotchas

- Locking is hard.
  - Too much, and performance is *worse than sequential*
  - Too little, and threads clash - *often unexpected results*
  - Not careful, and deadlock freezes your program forever!

```
pthread_mutex_lock(&lock);
if (!do_computation(resource)) {
    printf("Error doing computation\n");
    return false;  // !!!
}
pthread_mutex_unlock(&lock);
return true;
```

Threads - Gotchas

- Load / store are separate operations

```
global_ctr += 1; // possible bug here!

global_ctr = global_ctr + 1; // equivalent

// What happens if we switch to another thread
// before storing the new value?
```

How to reason about concurrency?

- There's no *one way* to reason about everything that could happen
- Try to break each problem down as much as possible
  - e.g. *reads*, *writes*, *things that happen only while a lock is held*

Suppose you have some global variable

```
int g = 0;
```

Two threads each run the following code:

```
g += 1;
g += 2;
```

How to reason about concurrency?

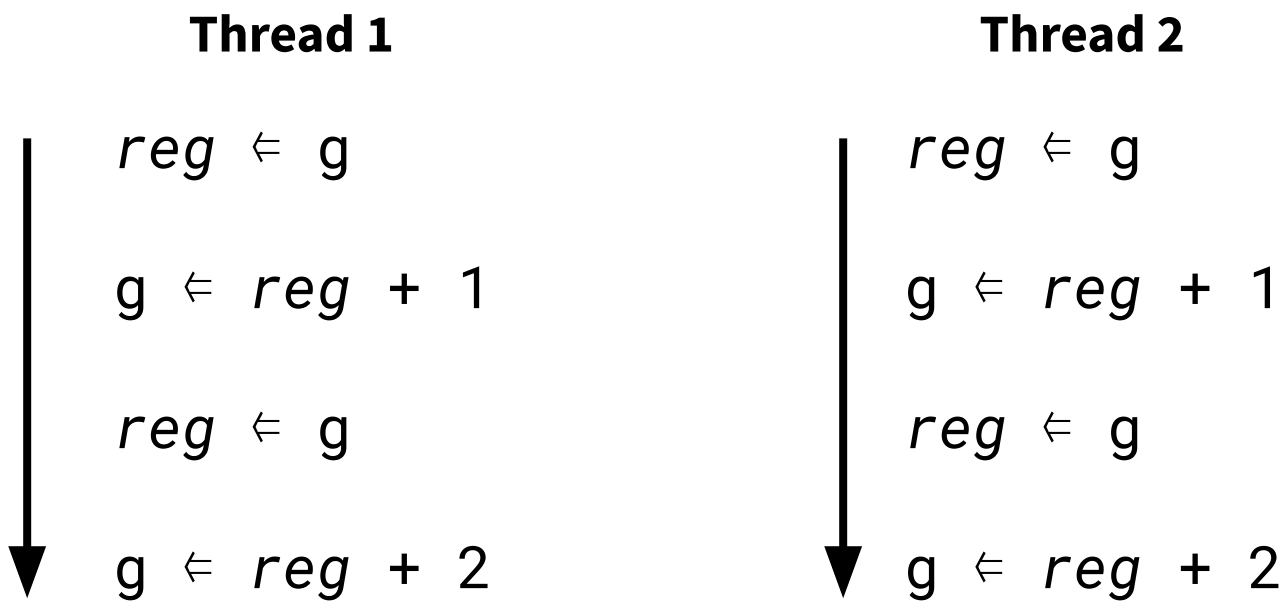
- Load / store are separate operations

```
g +=1;          g = g + 1;          load reg ⇐ g
                                    store g ⇐ reg + 1
```

```
g += 2;         g = g + 2;          load reg ⇐ g
                                    store g ⇐ reg + 2
```

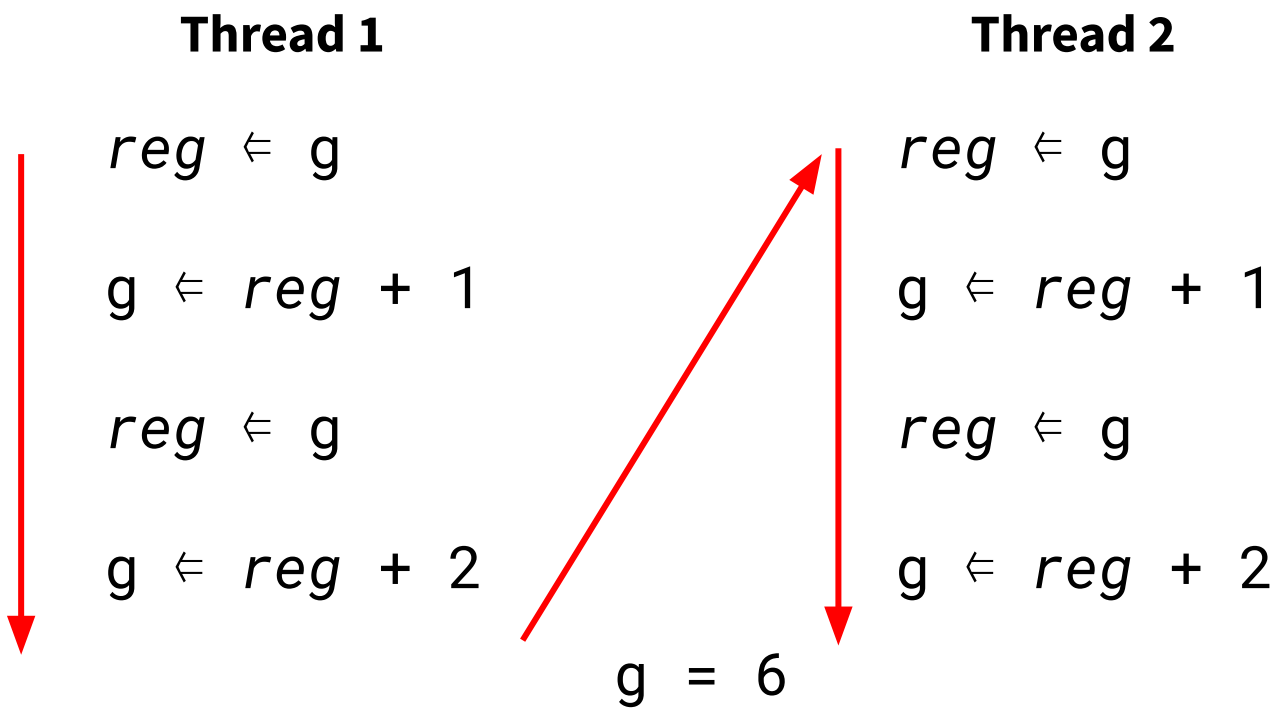
Each thread has its own set of registers, so *reg* can hold different values in different threads

How to reason about concurrency?

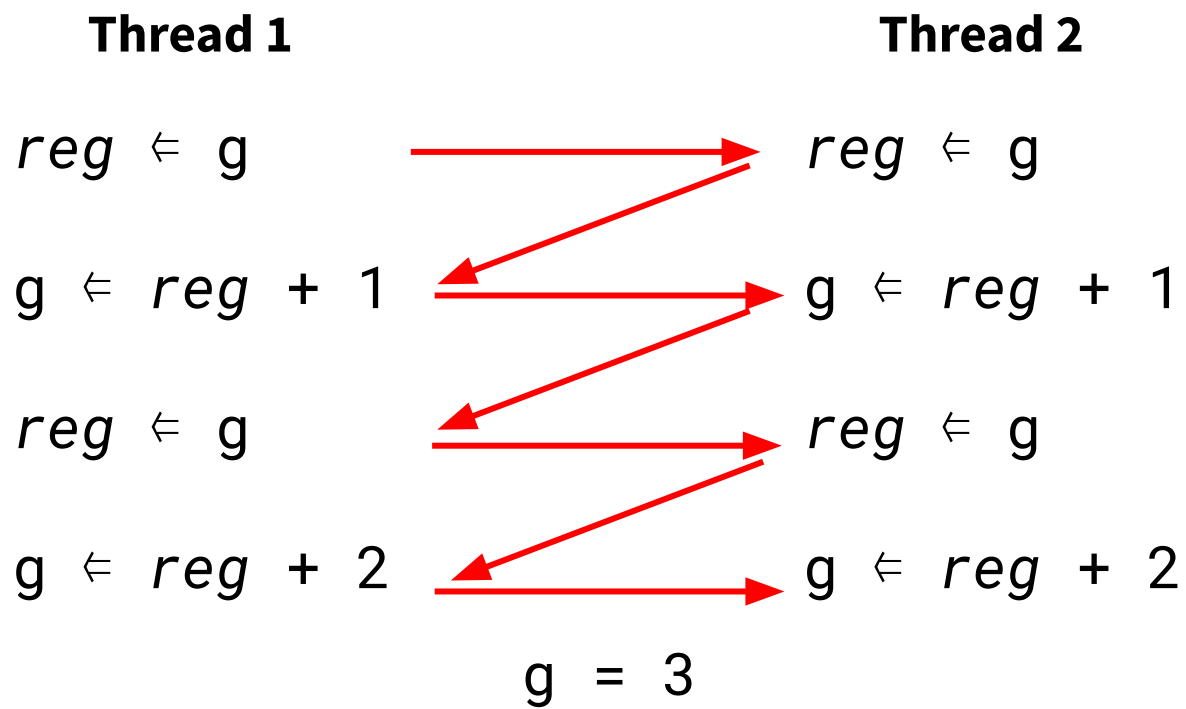


Remember: Each thread must still execute its own code in order sequentially within itself

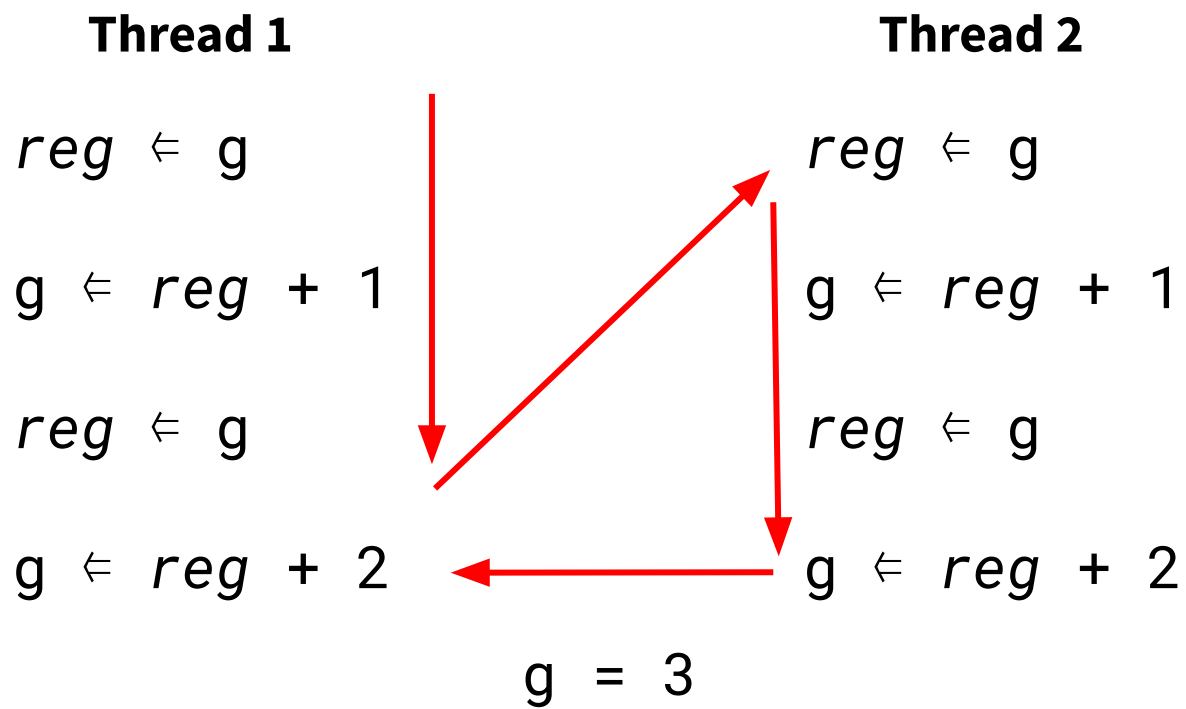
How to reason about concurrency?



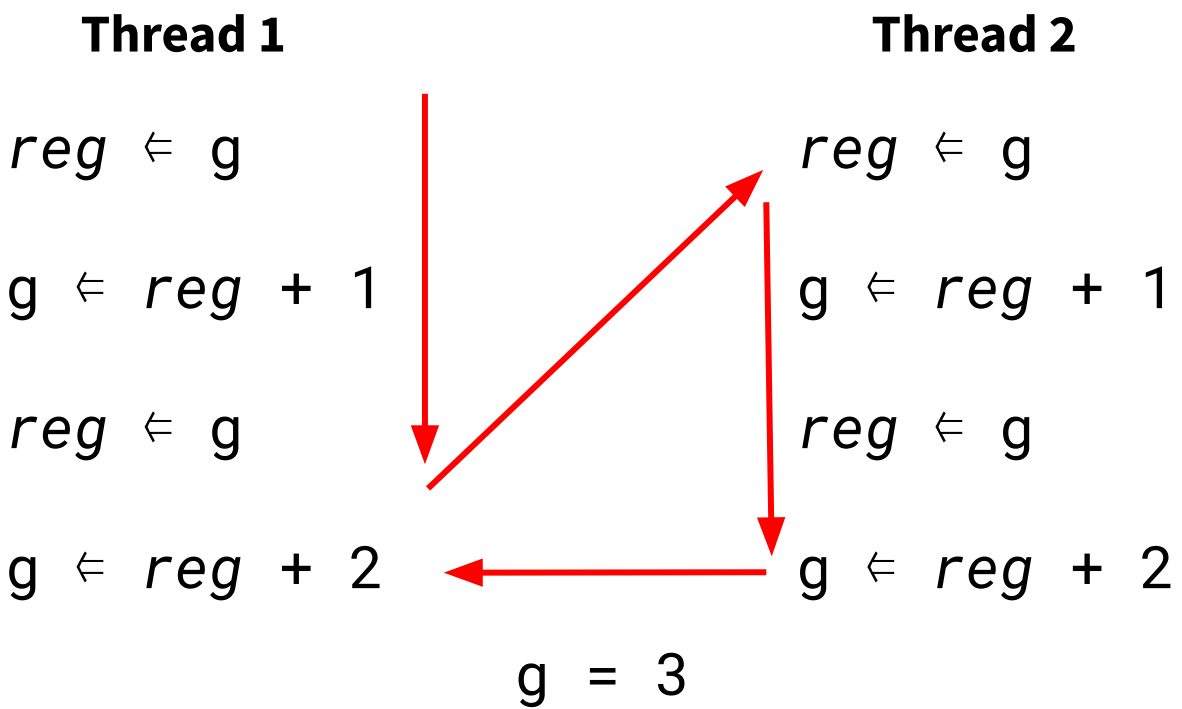
How to reason about concurrency?



How to reason about concurrency?



How to reason about concurrency?



If you "sandwich" work from one thread between a load and store in another thread you can "delete" the work done.

Exercise 2:
Reasoning about threads is hard

Exercise 2

```c
int g = 0;
void *worker(void *ignore) {
  for (int k = 1; k <= 3; k++) {
    g = g + k;
  }
  printf("g = %d\n", g);
  return NULL;
}

int main() {
  pthread_t t1, t2;
  int ignore;
  ignore = pthread_create(&t1, NULL, &worker, NULL);
  ignore = pthread_create(&t2, NULL, &worker, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return EXIT_SUCCESS;
}
```

What are the possible outputs of this program?

What is the range of values that g can have at the end of the program?

Exercise 2

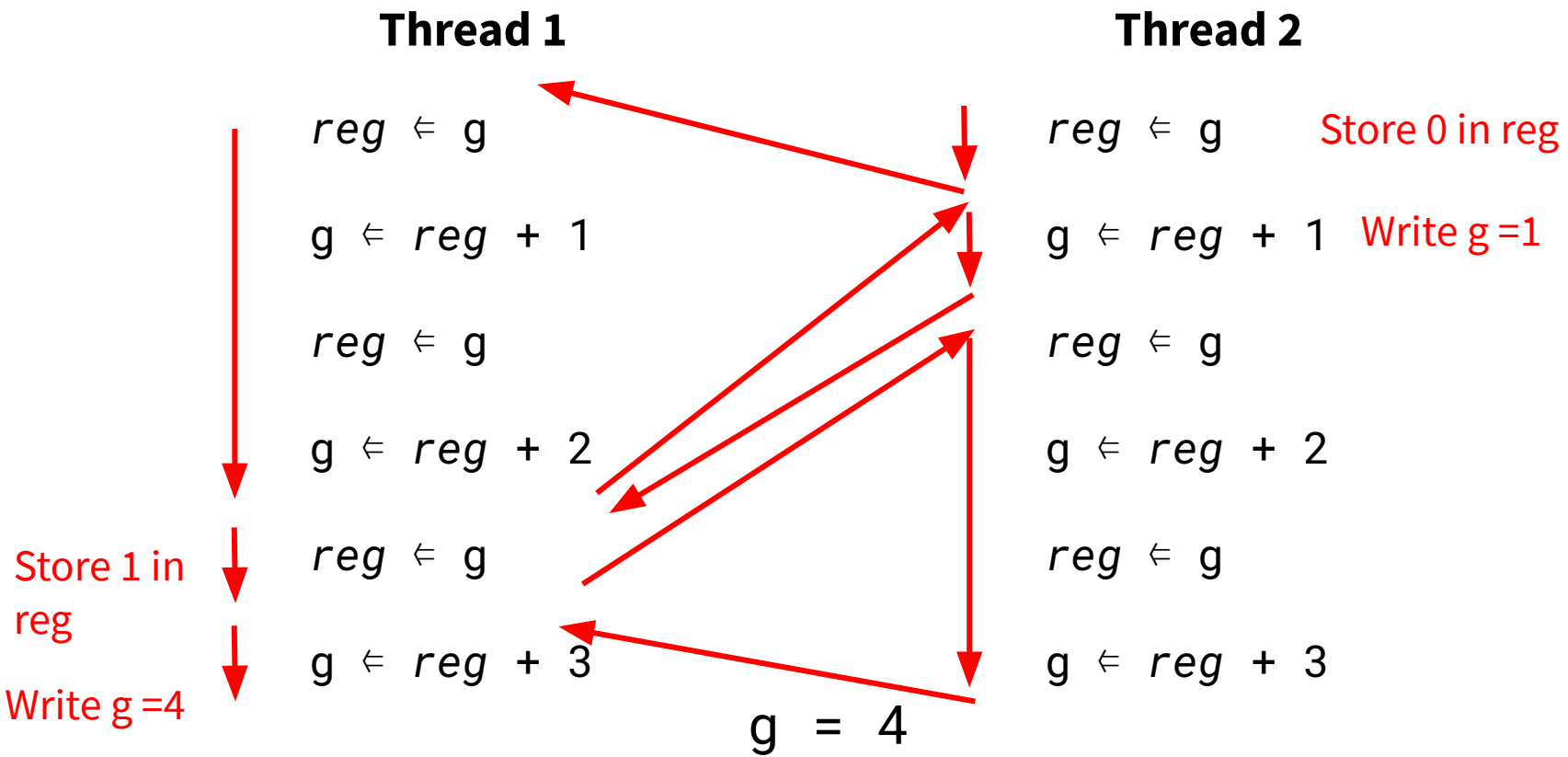
```
int g = 0;
void *worker(void *ignore) {
  for (int k = 1; k <= 3; k++) {
    g = g + k;
  }
  printf("g = %d\n", g);
  return NULL;
}

int main() {
  pthread_t t1, t2;
  int ignore;
  ignore = pthread_create(&t1, NULL, &worker, NULL);
  ignore = pthread_create(&t2, NULL, &worker, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return EXIT_SUCCESS;
}
```

What are the possible outputs of this program?

Lots of possible answers, here are a few:

g = 6 g = 12	g = 12 g = 12	g = 7 g = 9	g = 6 g = 11

How to get 4 from exercise 2



Exercise 2

```c
int g = 0;
void *worker(void *ignore) {
  for (int k = 1; k <= 3; k++) {
    g = g + k;
  }
  printf("g = %d\n", g);
  return NULL;
}

int main() {
  pthread_t t1, t2;
  int *ignore;
  ignore = pthread_create(&t1, NULL, &worker, NULL);
  ignore = pthread_create(&t2, NULL, &worker, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return EXIT_SUCCESS;
}
```

What is the range of values that g can have at the end of the program?

4 5 6 7 8 9 10 11 12

How to get 4 and 5 is tough to see. What you should take away: can't guarantee ordering/interleaving of threads. Need to be careful with shared data.

What the fork?

```
// fork 10 children and count off (random order)
int main(int argc, char **argv) {
   for (int i = 0; i < 10; ++i) {
      if (fork() == 0) {
         printf("%d\n", i);
      }
   }
}
```

How many times do we print?

Exercise 3

```
// Assume all necessary libraries and
header files are included
const int NUM_TAS = 10;

static int bank_accounts[NUM_TAS];
static pthread_mutex_t  sum_lock;

void *thread_main(void *arg) {
  int *TA_index =
reinterpret_cast<int*>(arg);

  pthread_mutex_lock(&sum_lock);
  bank_accounts[*TA_index] +=  1000;
  pthread_mutex_unlock(&sum_lock);

  delete TA_index;
  return NULL;
}
```

```
int main(int argc, char** argv) {
  pthread_t thds[NUM_TAS];
  pthread_mutex_init(&sum_lock, NULL);

  for (int i = 0; i < NUM_TAS; i++) {
    int *num = new int(i);
    if (pthread_create(&thds[i],  NULL, &thread_main, num)!=
0){
      /*report error*/
    }
  }


  for (int i = 0; i < NUM_TAS; i++) {
    cout << bank_accounts[i] << endl;

  }

  pthread_mutex_destroy(&sum_lock);
  return 0;
}
```

Exercise 3

a) Does the program increase the TAs' bank accounts correctly? Why or why not?

No its not correct. It needs to use pthread_join to wait for each thread to finish before exiting the main program. pthread_exit() might not be the best solution here. You want to check the return value of join to make sure the transaction applied rather than just exiting and trusting the threads to finish successfully. Gotta get those TA dolla's.

b) Could we implement this program using processes instead of threads? Why would or why wouldn't we want to do this?

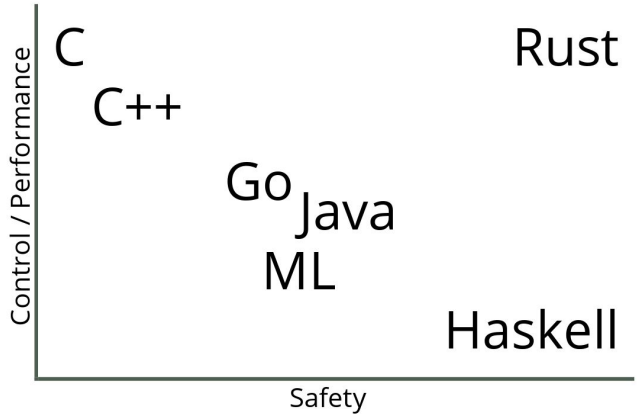
We could, but doing so would require some way for the processes to communicate with each other so that the data structure can be "shared" (remember that inter-process communication can be difficult and time consuming). It is much easier to just use threads since each thread could directly access the data structure.

c) Assume that all the problems, if any, are now fixed. The student discovers that the program they wrote is kinda slow even though its a multithreaded program. Why might it be the case? And how would you fix that?

Because there is a lock over the entire bank account array, so only one thread can increase the value of one account at a time and there is no difference from incrementing each account sequentially. To fix this, we can have one lock per account so that multiple threads can increment the account at the same time. (With the current setup, we could also just not use a lock since we know that no thread will have a conflicting TA_index. For a more generalized program, it would be better to use the first answer.)

Shoutout: The Rust Language

- No memory errors.
- No race conditions.
  - Whaaaaat? Yes.
- Performance close to C/C++ level
- Good abstractions like iterators & closures
  - Optimized down, so it’s as fast as if you wrote it by hand



Shoutout: Other Classes

- Like C and the “mysterious” kernel?
- Want to write a bunch of C++?
- Like doing a bunch of Concurrency?
- Liked html (for some reason)?
- Want to do C on limited systems?
- Learn about more low-level stuff?
- Want more 351-esque concepts?
- Want to understand the networking Code you wrote?

451 OS (best class)
457 Graphics
452 Distributed
154 Web Dev
474 Embedded sys
369 & 371 digital design
469 & 470 Comp Arch
461 Networks!

TA-ing

You are all well enough equipped to TA CSE333, CSE351, CSE374 and others.

You do NOT have to 4.0 a class to TA it

You do NOT have to be a super social person to TA
(Some of us are very introverted)

TAing will reinforce your understanding of any material

TAs are human too. It is ok to start off imperfect and make mistakes

If you think you would be interested, I would highly recommend reaching out and giving it a try. Please feel free to talk to us if you are interested.

Ask Us Anything!!!

Not on the Exam (but cool anyways)

- You've probably run afoul of SIGSEGV (a.k.a. "Seg fault")
  - What is it?
- UNIX processes can communicate with each other!
- signals are notifications sent between processes
  - They all have default handlers, such as "crash the program"
- You can use `signal()` or `sigaction()` to handle them yourself!

Demo: I am unstoppable

Not on the Exam (but cool anyways)

- To send “real” messages between processes, you already have what you need in your toolkit!
  - Set up a socket connection from a process to itself
    - You’ll have to use nonblocking calls for this
  - `fork()`
  - Each process closes one end, and uses the other to communicate
- You can do this with TCP sockets, but there are better options available
- `socketpair()` does all of this for you!

Demo: Crash-proof logging

Enough fun - Back to work(sheet)!

You've Learned A Lot! Good Luck!