# CSE 333
# Section 10

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

# Logistics:

Due tonight :

HW4 (at most one late day)

Final Exam Review :

TBD this weekend

Course Evaluations:

Due Sunday (12/8) Please do them <3

Final Exam:

Wednesday (12/11)

# Terminology

- **Thread**

  Some sequential execution of a program

- **Process**

  The execution environment (may contain one or more threads)

- **Parallelism**

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

- **Concurrency**

  Making progress on multiple tasks without having to wait for old tasks to finish

*“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 vs Processes

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

# 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)

# 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(NULL);
}
```

# 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?
```

# 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(shared_resource)) {
    printf("Error doing computation\n");
    return false; // !!!
}
pthread_mutex_unlock(&lock);
return true;
```

# 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;
```

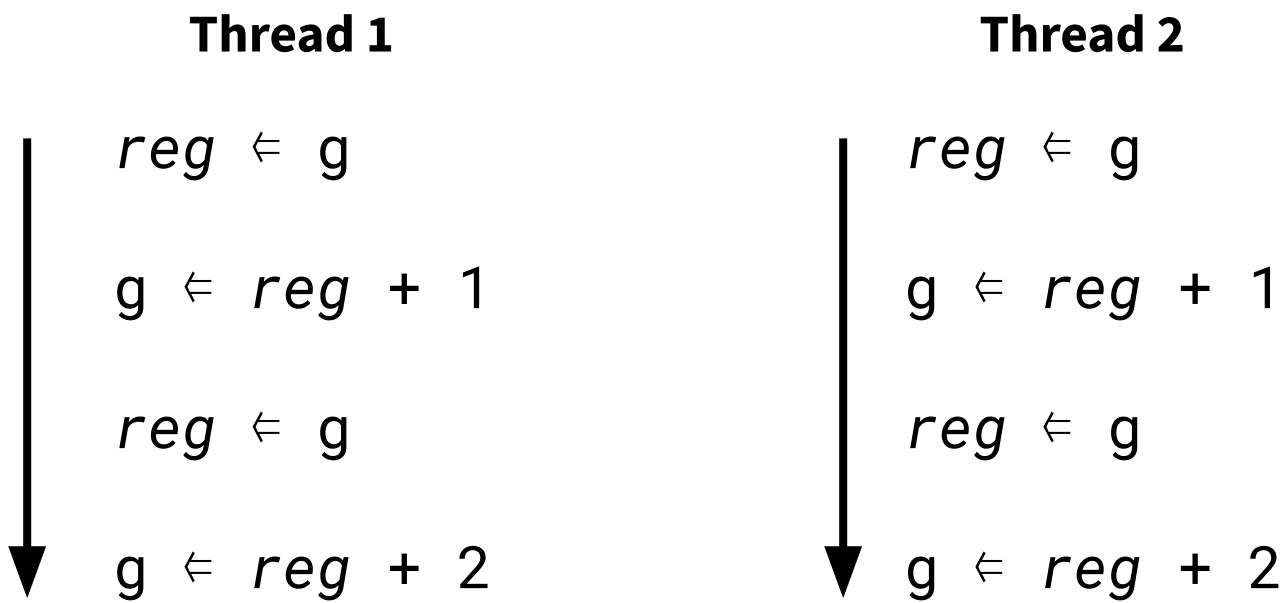
# How to reason about concurrency?

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

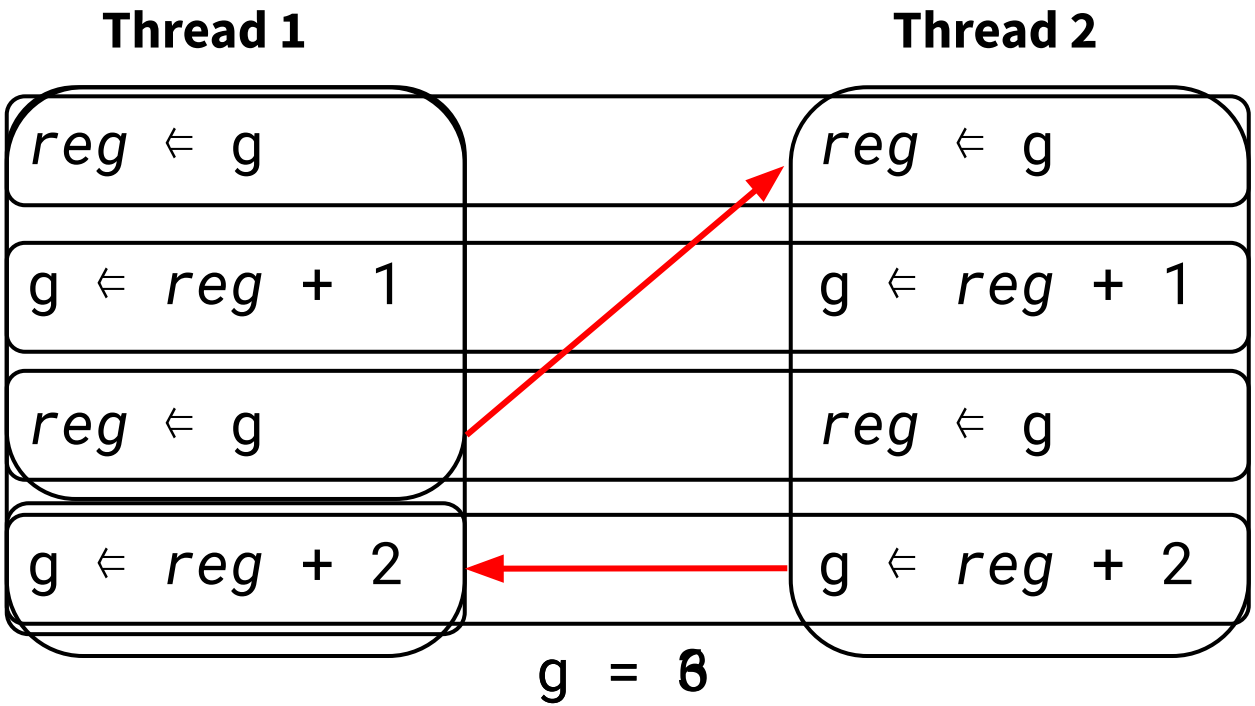
```
g += 2;         g = g + 2;          load reg ⇇ g
                                    store g ⇇ reg + 2
```

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

# How to reason about concurrency?



# How to reason about concurrency?



# Exercise 1:
# Reasoning about threads is hard

**What are the possible final values of the global variable ‘g’?**

```
read g
store g + 1
```

```
read g
store g + 1
```

```
read g
store g + 2
```

```
read g
store g + 2
```

```
read g
store g + 3
```

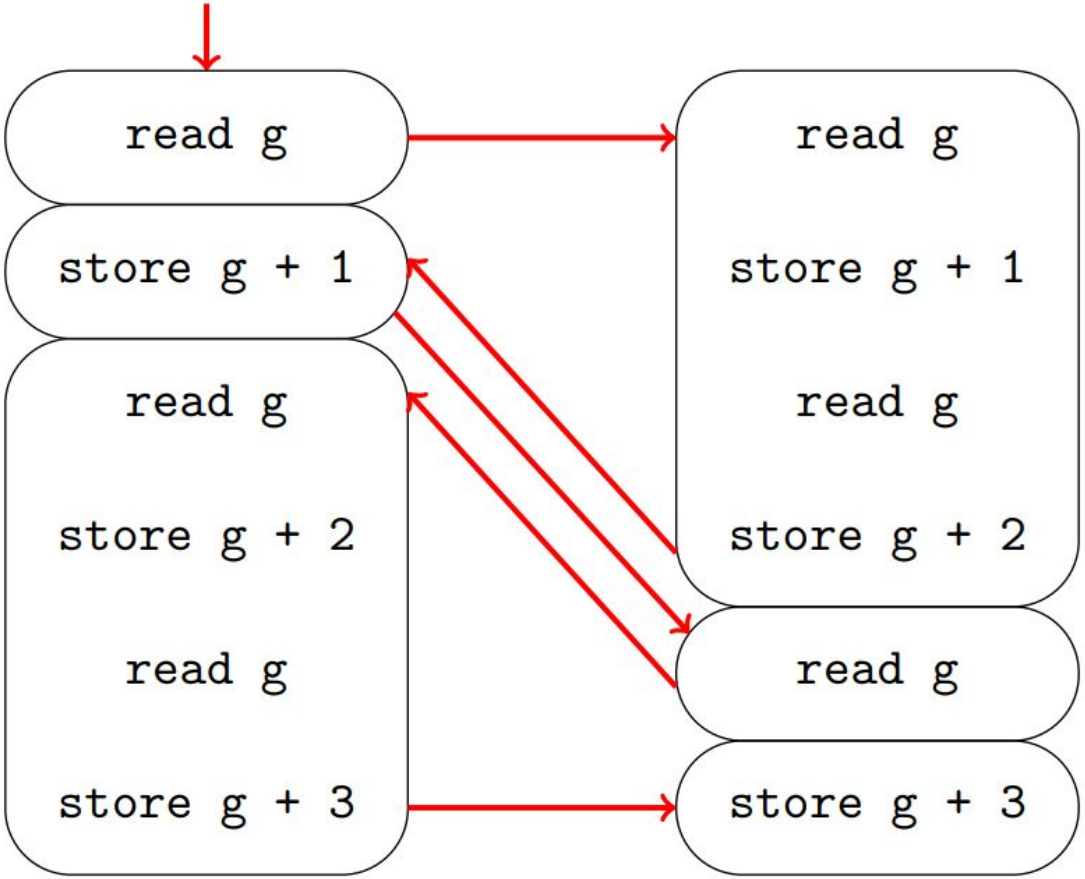
```
read g
store g + 3
```

What are the possible final values of the global variable ‘g’?

```
read g
store g + 1
read g
store g + 2
read g
store g + 3
```

```
read g
store g + 1
read g
store g + 2
read g
store g + 3
```

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15+


read g
store g + 1
read g
store g + 2
read g
store g + 3
read g
store g + 1
read g
store g + 2
read g
store g + 3

g = 4

# 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(...)`

# 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);
      }
   }
}
```

Can you spot the bug?

# Exercise 3:
# Synchronization with threads is hard

```
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*/
    }
  }
```

//No pthread_join()!!!!!!!

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

  }
```

// No pthread_exit()!!!!!

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

```
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;
}
```

The data structure is in shared memory, easier for threads to share.

```
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;
}
```

```
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;
}
```

The Lock is on the entire array when we only need one index!

# Exercise 4:
# Threads vs Processes

# Reminder: Threads vs Processes

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

# **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

# Off-Topic: Teaching

# 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 (Travis didn't 4.0 this class)

You do NOT have to be a super social person

TAing will reinforce your understanding of any material

**If you think you would be interested, I would highly recommend reaching out and giving it a try.**

# Ask Us Anything!!!