

CSE 333

Section 10

Threads, Pr-, and Concurrency -ocesses

Logistics

Due Tonight:

HW4 (can use two late days)

Course Evaluations:

Open until Wednesday, June 9, 11:59 pm PDT (please fill these out!)

Final Exam:

Open from Sunday, June 6 to Wednesday, June 9 (inclusive)

Threads and Processes

“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

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)

Processes

- Recall from 351
- Created using `fork()` - the only function that returns twice!
 - Child gets 0
 - Parent gets new pid (process id) of child
- Essentially duplicates the parent process
- Get status of children with `waitpid(...)`
- Replace currently running process with a new one using `exec()`

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	Copies
Communication	Easy	Difficult
Synchronization	Difficult	N/A
Think about overhead and switching between them → "Weight"	"light"	"heavy"
Robustness	One crashes, all crash	Independent of each other

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

Exercise 1

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).

Thread with pthread

POSIX Thread Basics

Declared in `pthread.h` (Compile and link with `-pthread`)

Note: C++11 has its own (different) thread library

Creation	<code>pthread_create</code>	Parent: “Go do this {function}”
Termination	<code>pthread_exit</code> start_routine returns	“I’m done with my task!”
	<code>pthread_cancel</code>	“I changed my mind, you can stop now”
	<code>exec()</code> or <code>exit()</code> is called <code>main()</code> returns	The entire process is terminated
Synchronization	<code>pthread_join</code>	“I’ll wait for you to finish and report back your result” (resource persists until joined)
	<code>pthread_detach</code>	“You’re free now, go forth and prosper” (automatically cleans up on termination)

Synchronize!

- Remember, threads share an address space and system resources
- This makes it easy to communicate, but how do you avoid a total free-for-all?
- Protect your critical sections with locks / mutexes!
 - `pthread_mutex_t lk;` “Create space for a new lock”
 - `pthread_mutex_init(...)` “Initialize the lock”
 - `pthread_mutex_lock(...)` “Grab the lock or wait until I can get it”
 - `pthread_mutex_unlock(...)` “Someone else can use the lock now”
 - `pthread_mutex_destroy(...)` “No one needs the lock anymore, clean it up”

Thread 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);  
}
```

Thread 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;
```

Reasoning About Threads is Hard

- 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 $reg \Leftarrow g$
store $g \Leftarrow reg + 1$

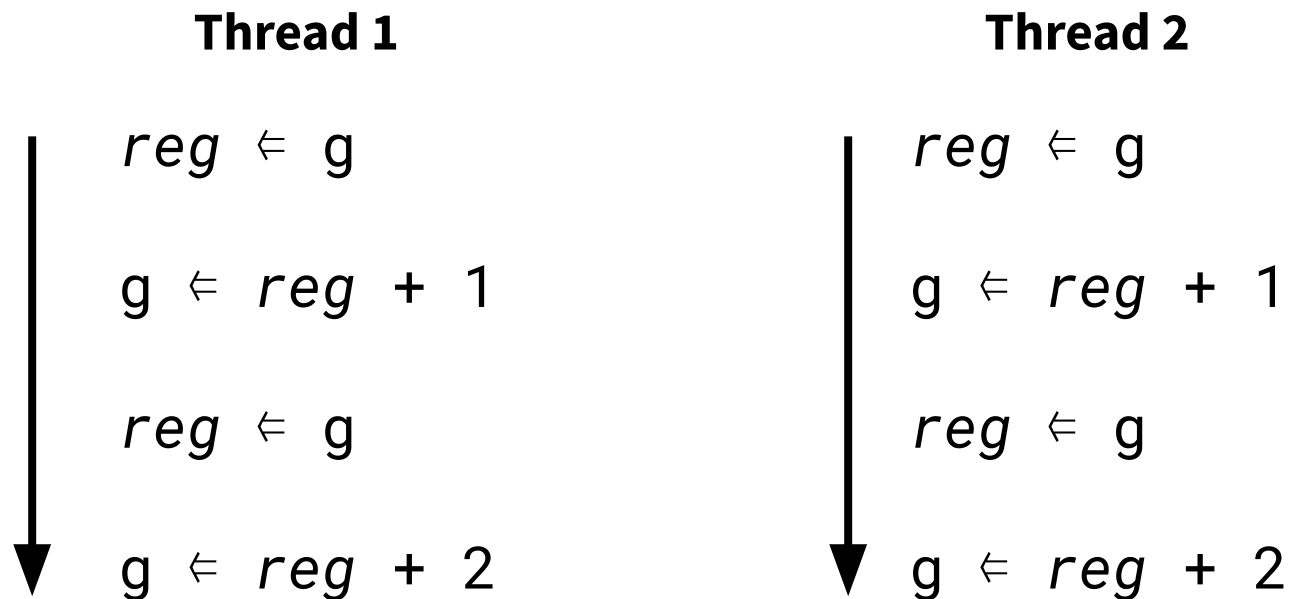
`g += 2;`

`g = g + 2;`

load $reg \Leftarrow g$
store $g \Leftarrow reg + 2$

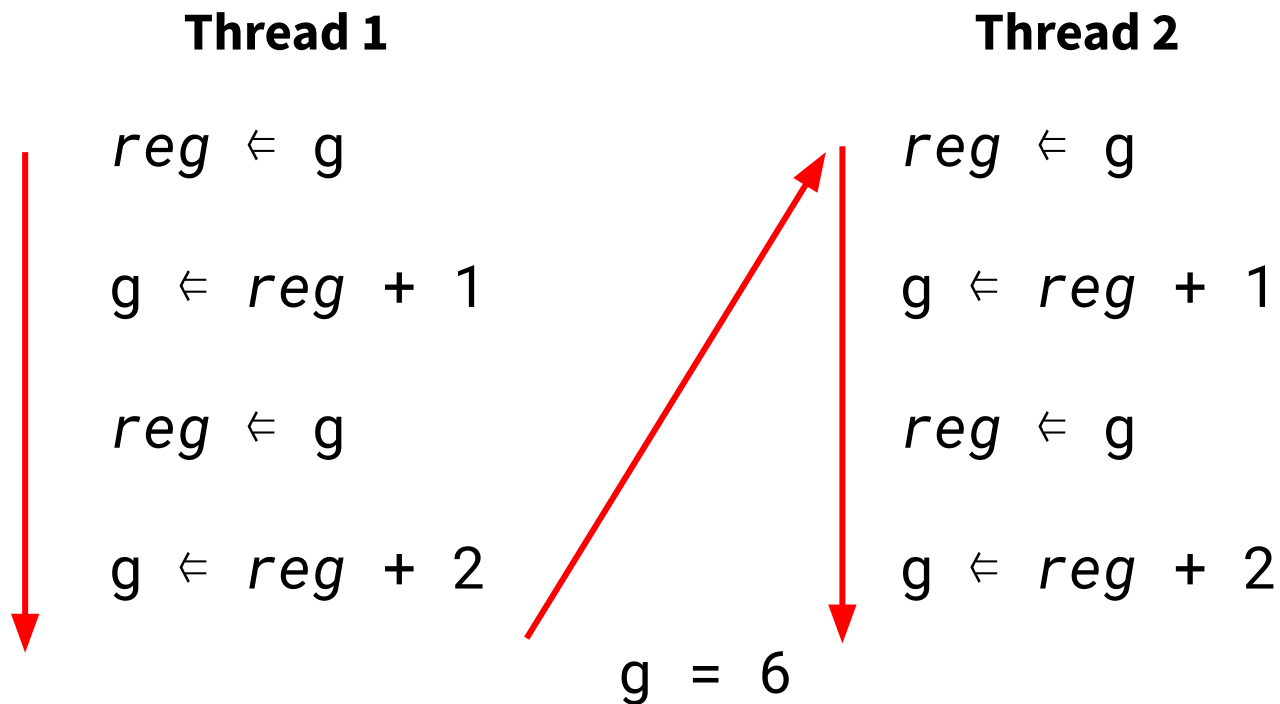
Each thread has its own set of registers, so *reg* can hold different values in different threads

How to Reason about Concurrency

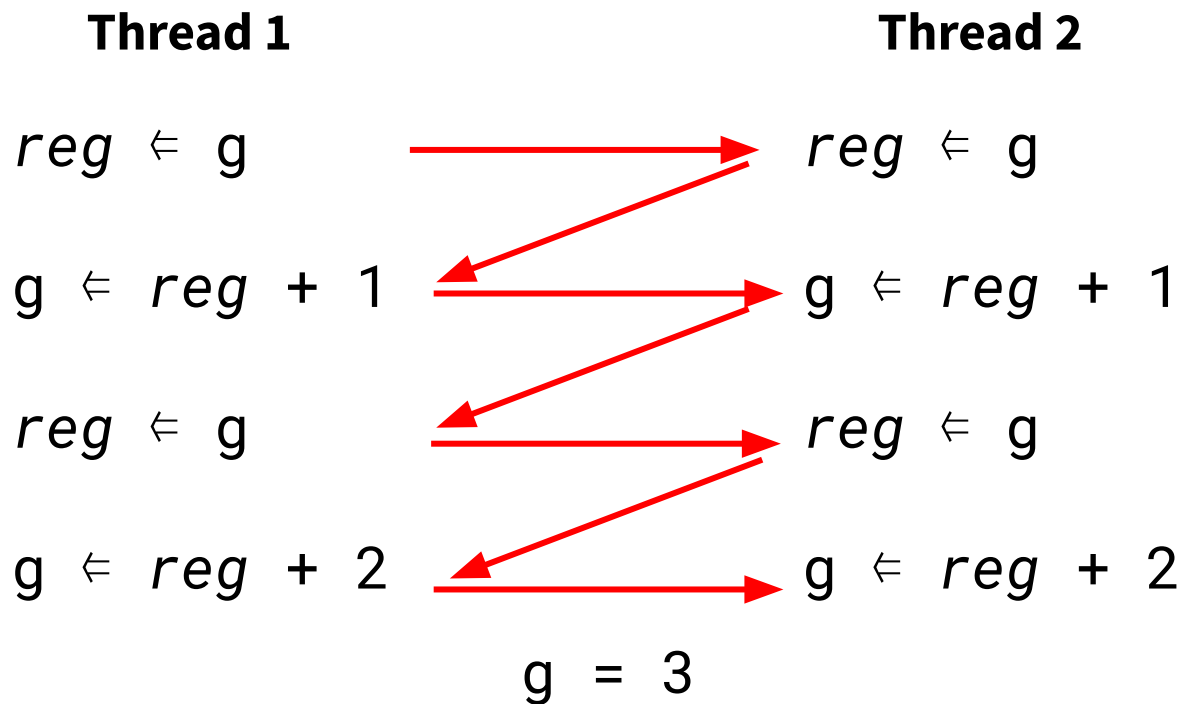


Remember: Each thread must still execute its own code in order sequentially within itself

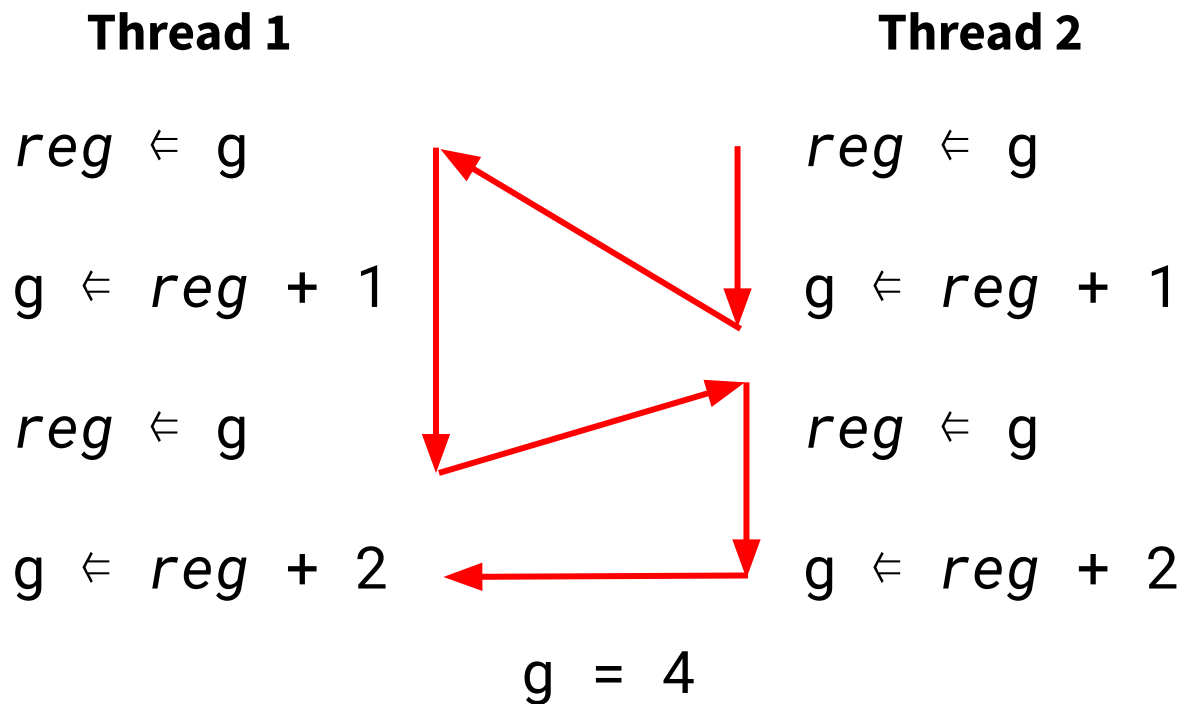
How to Reason about Concurrency



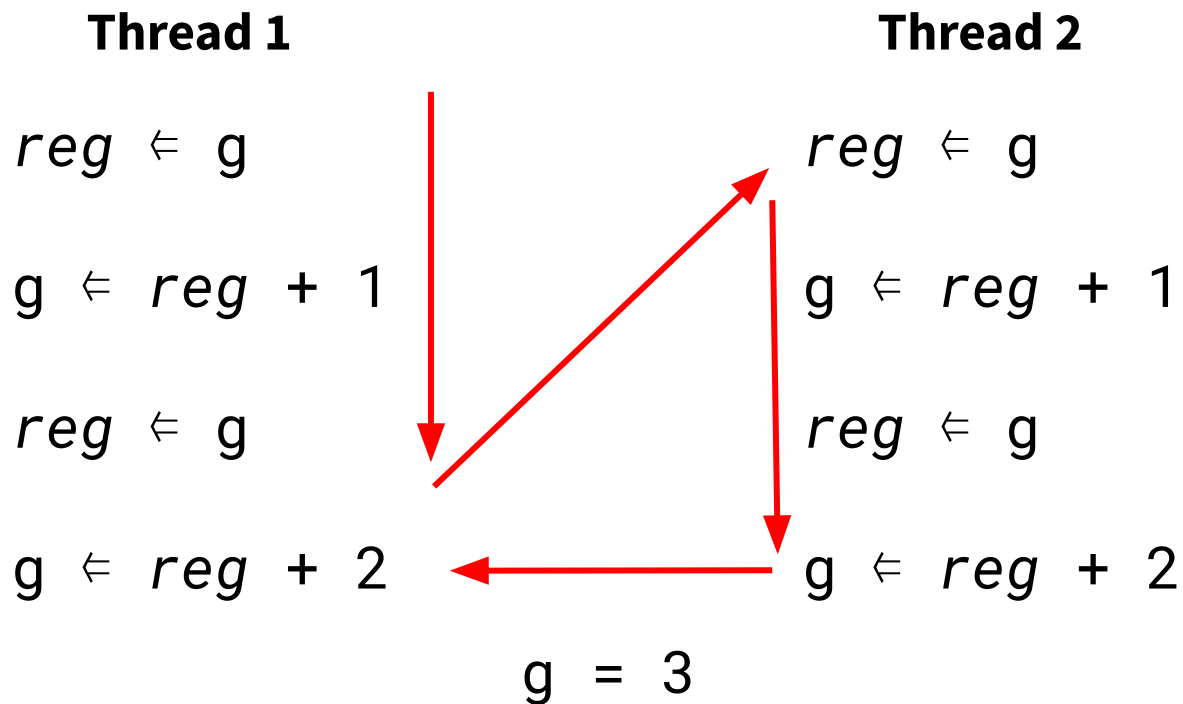
How to Reason about Concurrency



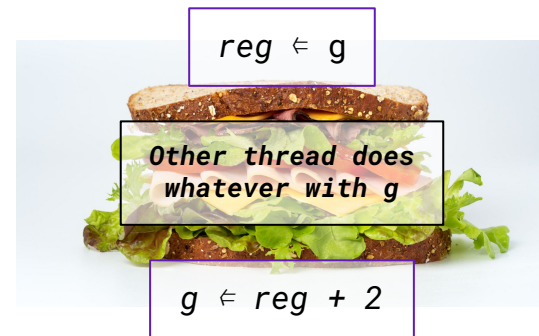
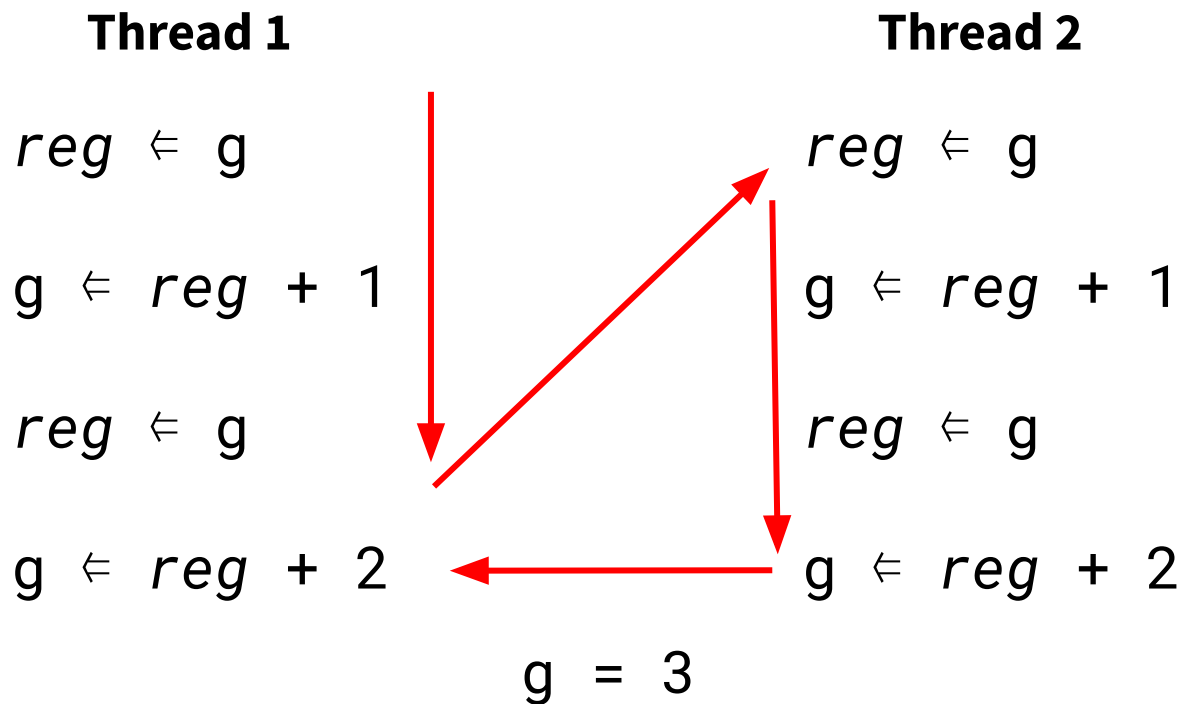
How to Reason about Concurrency



How to Reason about Concurrency



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

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?

(think of as many as you can!)

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

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 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.

How to Get 4 from Exercise 2

Thread 1

$reg \leftarrow g$

$g \leftarrow reg + 1$

$reg \leftarrow g$

$g \leftarrow reg + 2$

$reg \leftarrow g$

$g \leftarrow reg + 3$

$g = 4$

Thread 2

$reg \leftarrow g$

$g \leftarrow reg + 1$

$reg \leftarrow g$

$g \leftarrow reg + 2$

$reg \leftarrow g$

$g \leftarrow reg + 3$

Store 0 in reg

Write $g=1$

Store 1 in reg

Write $g=4$

Exercise 3

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 nullptr;
}
```

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

    for (int i = 0; i < NUM_TAS; i++) {
        int *num = new int(i);
        if (pthread_create(&thds[i], nullptr, &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?
- b. Could we implement this program using processes instead of threads? Why would or why wouldn't we want to do this?
- 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?

Exercise 3

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

No, it's 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.

Exercise 3

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.

Exercise 3

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?

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

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.)

Thank You for a Great Quarter!

You've Learned A Lot! Good Luck!



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.**

Shoutout: Other Classes

- Like C and the “mysterious” kernel? 451 OS
- How about that “mysterious” compiler? 401 Compilers
- Want to write a bunch of C++? 457 Graphics
- Liked html (for some reason)? 154 Web Dev
- Like doing a bunch of concurrency? 452 Distributed
- Want to do C on limited systems? 474 Embedded Sys
- Learn about more low-level stuff? 369 & 371 Digital Design
- Want more 351-esque concepts? 469 & 470 Comp Arch
- Want to understand the networking code you wrote? 461 Networks
- Want to change some colors with C? 455 Vision
- Liked patching up your 333gle? ssh-keys? 484 Security

Bonus Slides

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!

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!

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

