CIT 5950
Section 5

Threads, Processes, and Concurrency

Logistics

Due This Friday (tomorrow night):

Homework 1 @ 11:59 pm

Threads and Processes

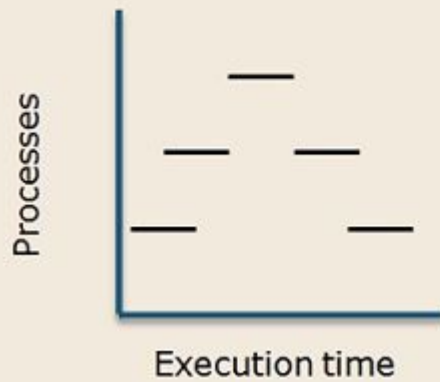
“Computers are really dumb. They can only do a few things like shuffling around numbers, but they do them really really fast so that they appear smart.”

Hal Perkins

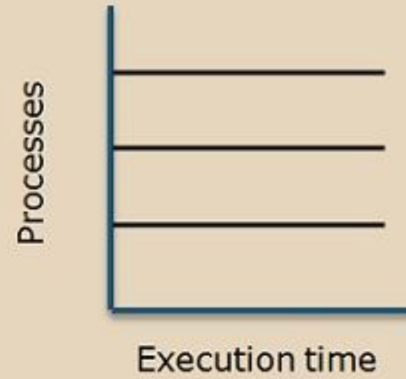
Threads are just a way of making computers appear to do multitasking, *regardless of whether they are running one or more* CPUs

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)



VS



Processes

- 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
“Weight”	“light”	“heavy”
Robustness	One crashes, all crash	Independent of each other

Think about overhead and switching between them → “Weight”

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	pthread_**create**	Parent: "Go do this {function}"
Termination	pthread_**exit** start_routine returns	"I'm done with my task!"
	pthread_**cancel**	"I changed my mind, you can stop now"
	exec() or exit() is called main() returns	The entire process is terminated
Synchronization	pthread_**join**	"I'll wait for you to finish and report back your result" (resource persists until joined)
	pthread_**detach**	"You're free now, go forth and prosper" (automatically cleans up on termination)

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);
}
```

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

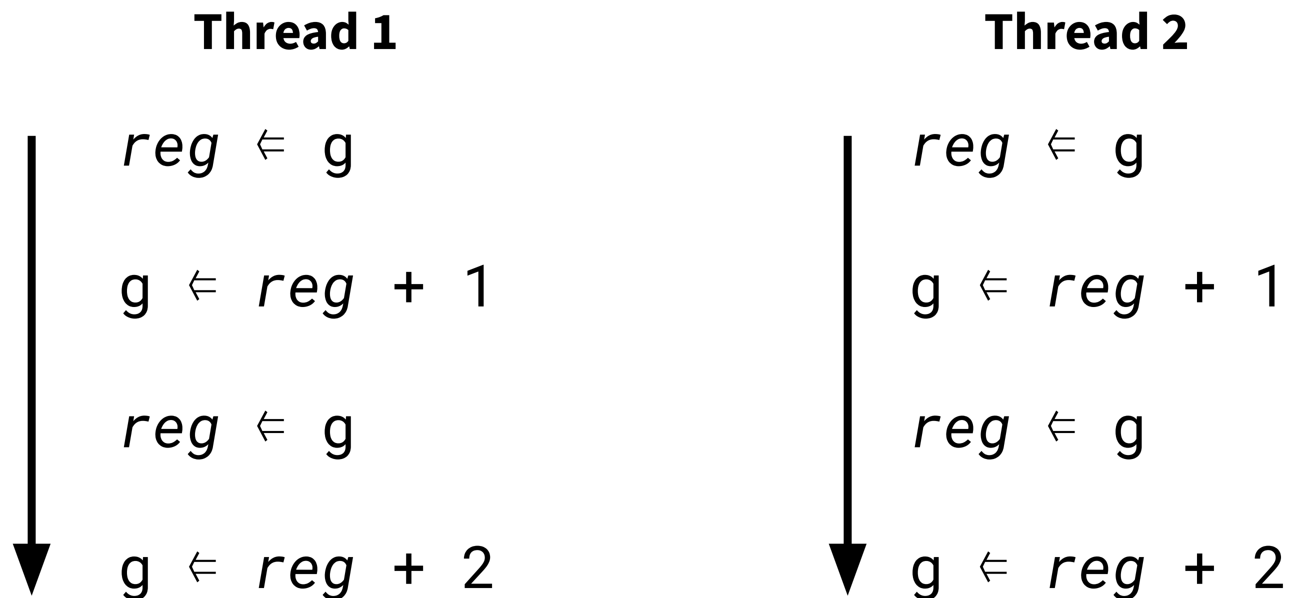
- Load / store are separate operations

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

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

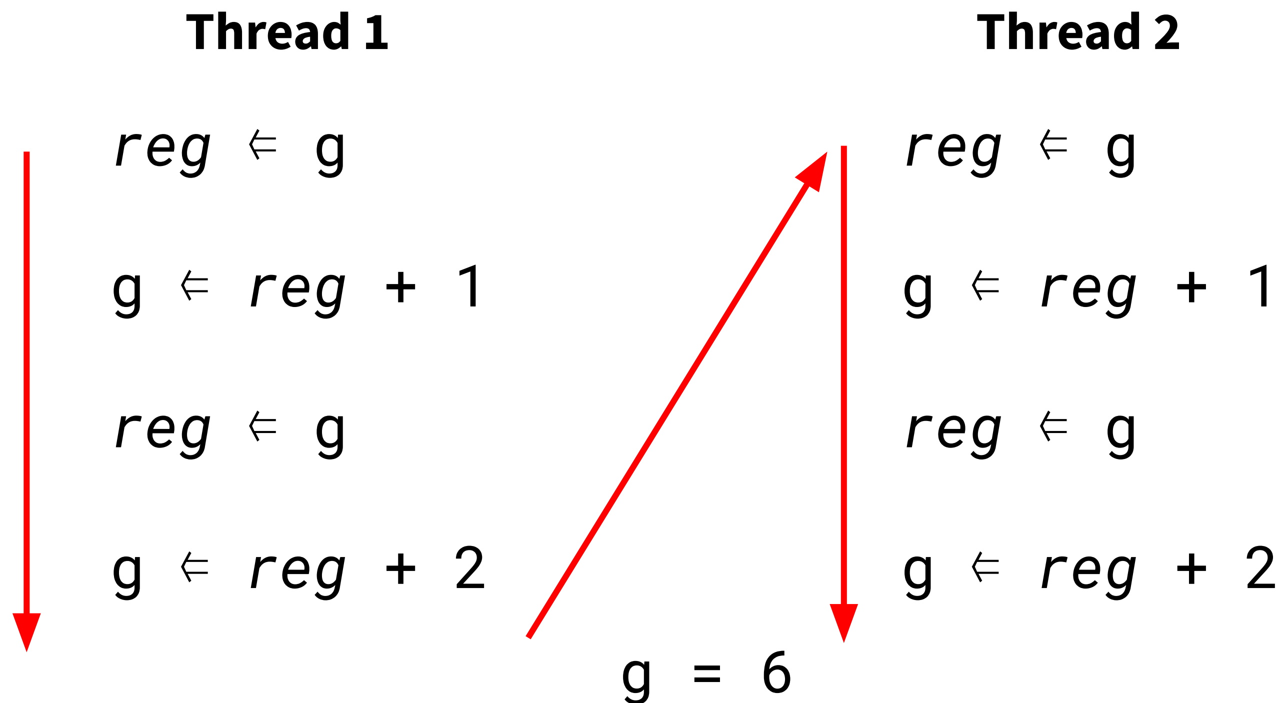
Each thread has its own set of registers, so *reg* can hold different values in different threads

How to Reason about Concurrency

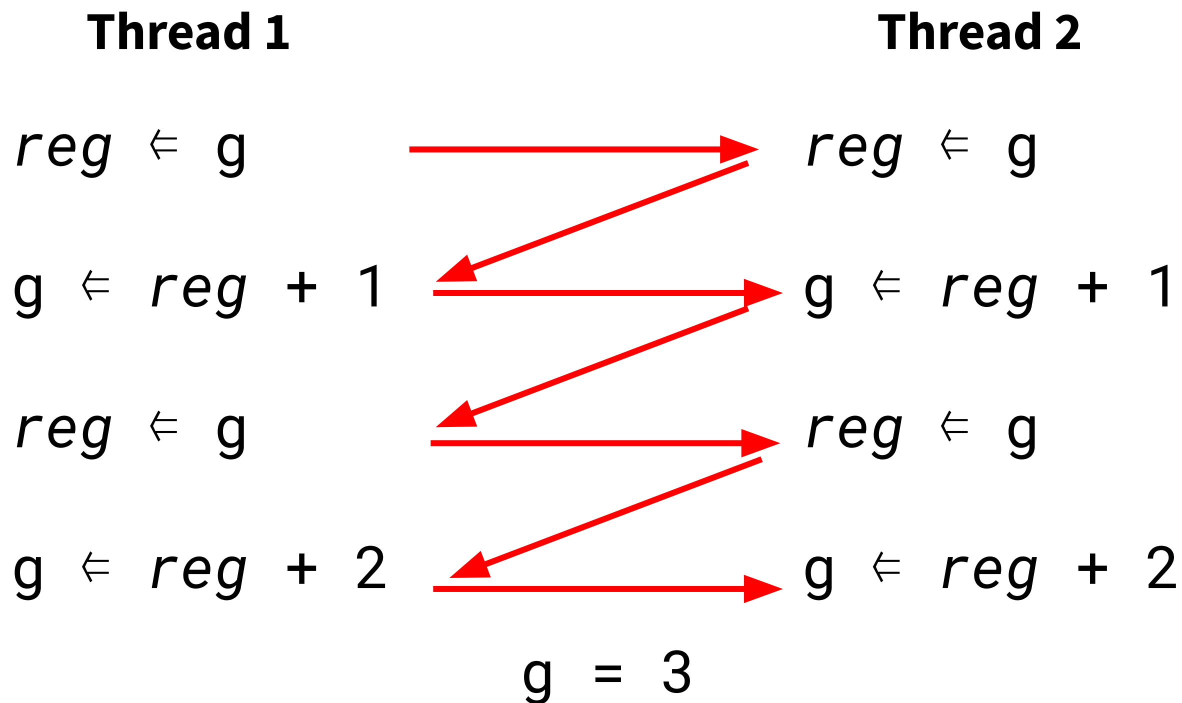


Remember: Each thread must still execute its own code in order sequentially within itself

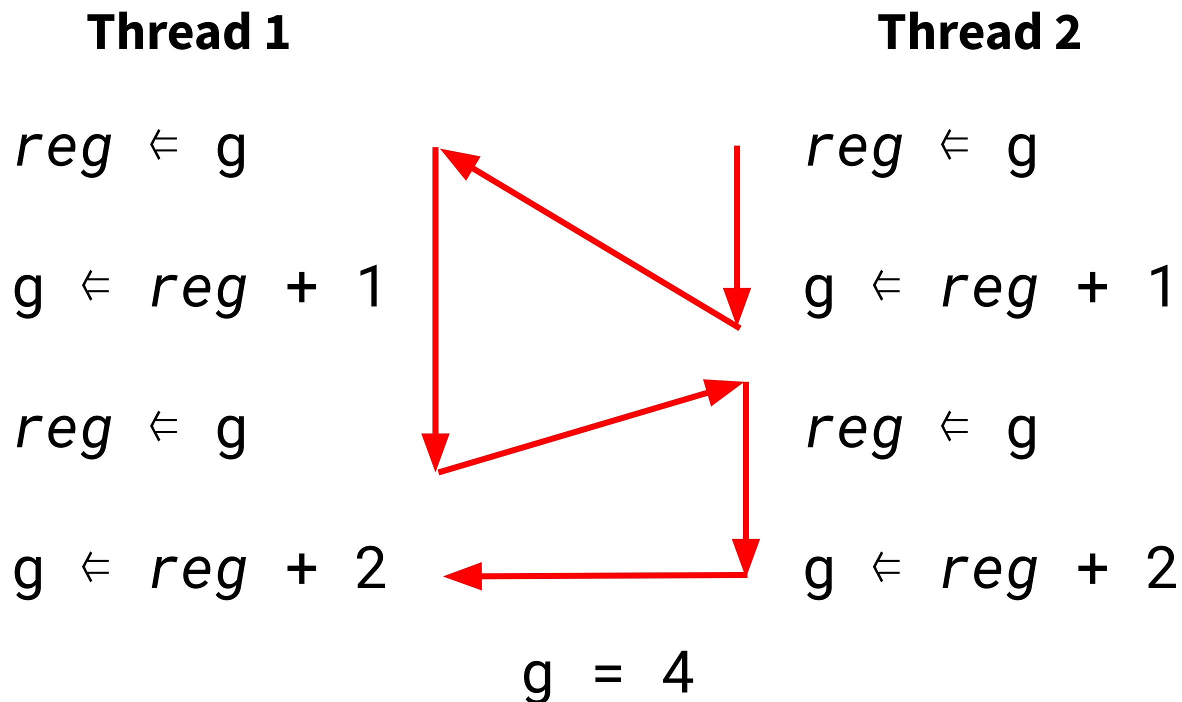
How to Reason about Concurrency



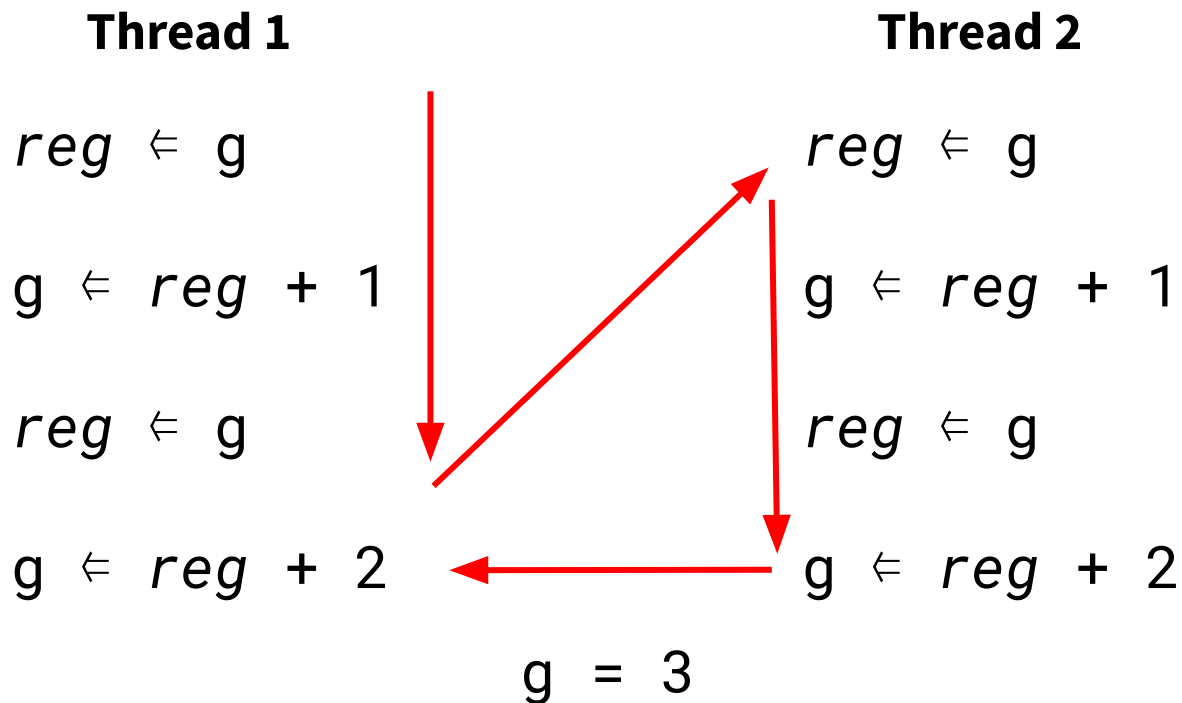
How to Reason about Concurrency



How to Reason about Concurrency



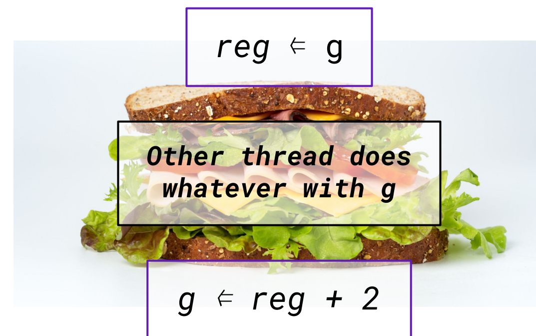
How to Reason about Concurrency



How to Reason about Concurrency

Thread 1	Thread 2
reg ⇐ g	*reg* ⇐ g
g ⇐ *reg* + 1	g ⇐ *reg* + 1
reg ⇐ g	*reg* ⇐ g
g ⇐ *reg* + 2	g ⇐ *reg* + 2

g = 3



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

```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?

Lots of possible answers, here are a few:

g = 6	**g = 12**	**g = 7**	**g = 6**
g = 12	**g = 12**	**g = 9**	**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	Thread 2
reg ⇐ g	*reg* ⇐ g
g ⇐ *reg* + 1	g ⇐ *reg* + 1
reg ⇐ g	*reg* ⇐ g
g ⇐ *reg* + 2	g ⇐ *reg* + 2
reg ⇐ g	*reg* ⇐ g
g ⇐ *reg* + 3	g ⇐ *reg* + 3

Store 0 in reg

Write g =1

Store 1 in reg

Write g =4

g = 4