COMP1521

TUTORIAL 10

PROCESSES > THREADS> CONCURRENCY > PIPES

NOTE

- Assignment due this Friday 5pm!!!
- Do myexperience please!
- Week 10 Lab is due Monday Week 11!
- Week 10 Weekly Test is due Thursday Week 11!
- Today's lab will be split across practice exam or assignment/lab help
- Extra practice exams are usually released soon

PROCESSES

Running a process

- So earlier in the term we learnt what the basics of a process is, and that when we run the programs that we write it creates a new process that is our program.

- Interestingly clang, dcc, cat, mkdir, touch and many of the commands we use in terminal are programs too, their compiled programs just live in our bin folder

```
brodie in ~
> which cat

/usr/bin/cat

brodie in ~
> cat .diary

hey get out of here this is MY diary
```

- The way our computer knows where the programs exist is through our systems $PATH environment variable

```
brodie in ~
> /usr/bin/cat .diary

hey get out of here this is MY diary
```

Running a process (system)

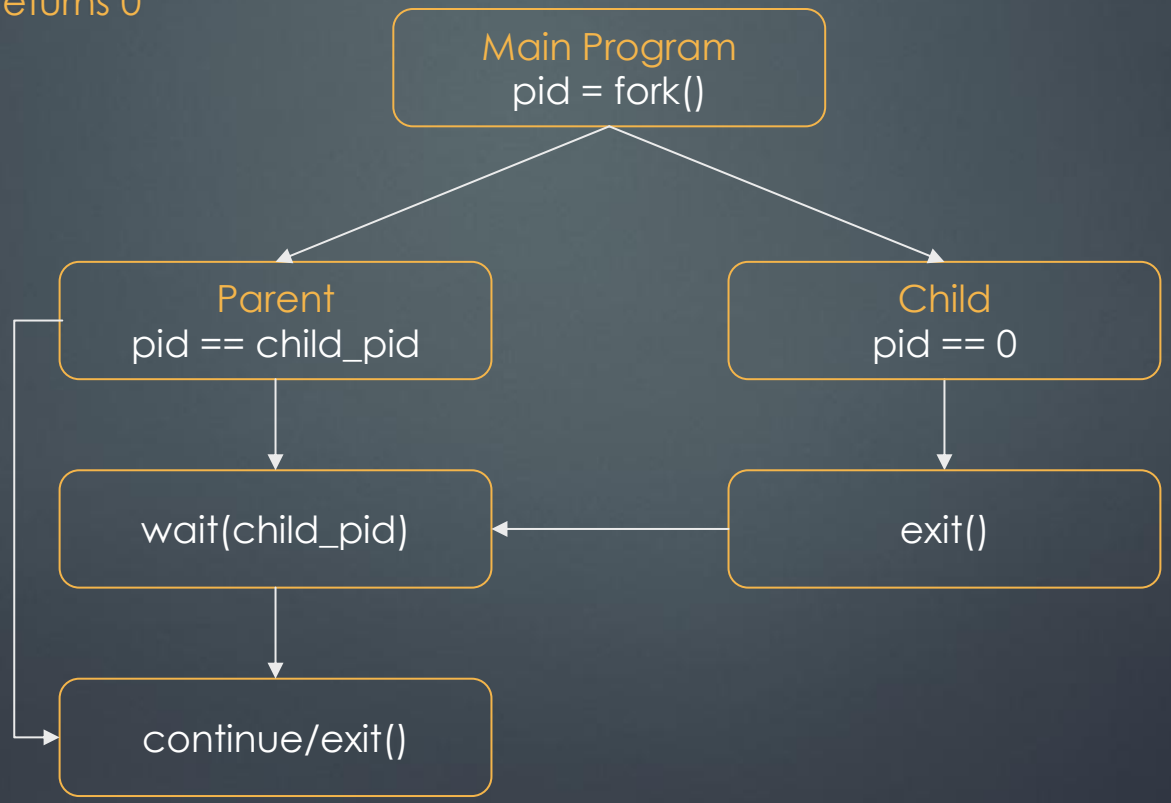
- We can write programs that can run a process for us and the simplest example, is the system() function.

```c
int main(void) {
  system("echo Hello World!");

  return 0;
}
```

- This is uncommon way to run programs, why?
  - Hard to write extensible programs, lacks functionality
  - It can have security issues, especially if there is a user input involved in the system function call

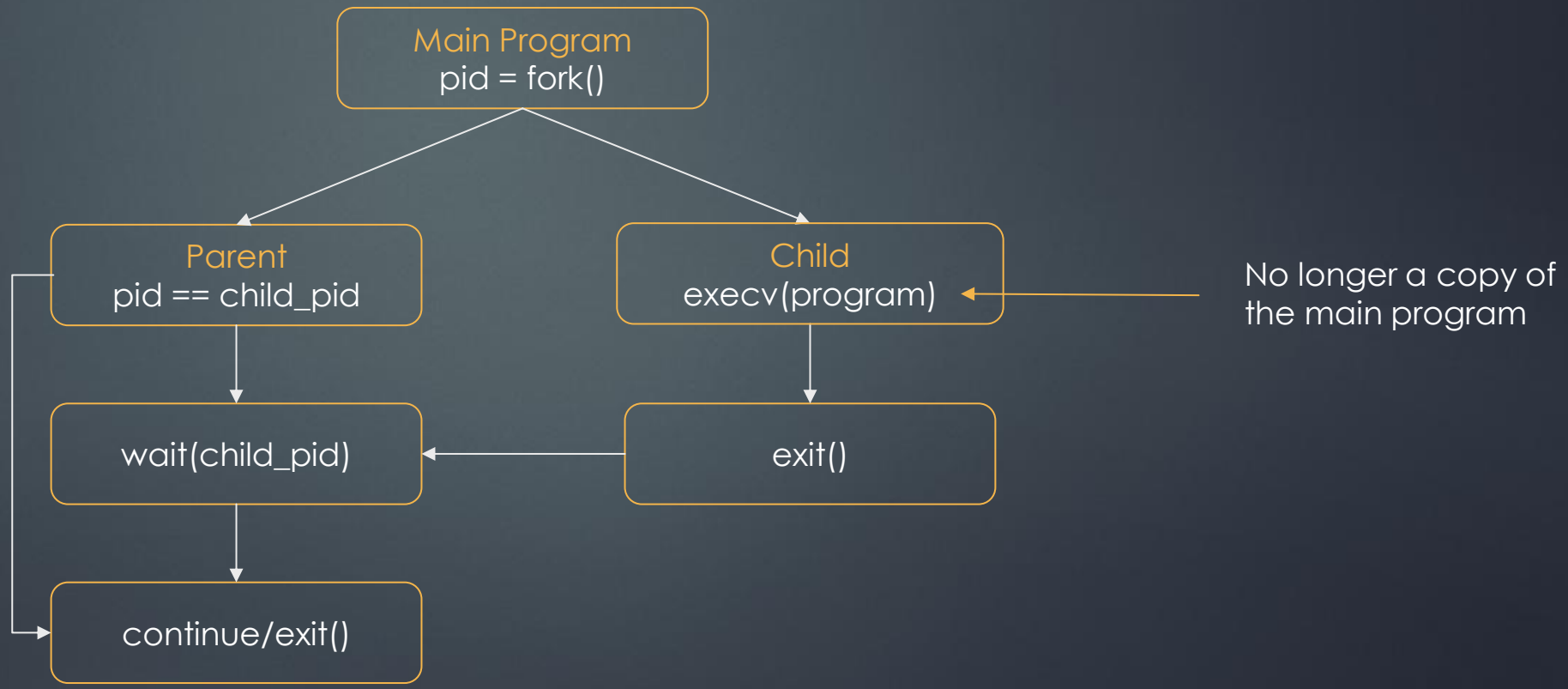
Running a process(fork and execve)

- A more common way to run a process from a program, is to use fork and execve
  - fork(): creates a copy of your current process
- Fork as we know creates a copy, and when we call fork, it returns a pid (process ID), we use the returned pid to determine what to do next
  - In the parent process this returns the childs pid
  - In the child process this returns 0



Running a process(fork and execve)

- execv is a little weird, it doesn't create a new process to run the program, it simply just replaces itself
  - this is why we use fork and execv together
  - otherwise, it would just replace our main program, never be able to complete



Running a process(fork and execve)

- When using fork and execve in C, we use the returned pid from fork to determine which process to wait (parent), or which process we execv in (child) as we know the child receives a pid of 0.

```c
int main(void) {
    pid_t pid = fork(); // copy our current process

    if (pid == 0) {
        //              program  argv[1]   argv[2]  (array terminator)
        char *argv[] = {"echo", "Hello,", "World!", NULL}; // create our arguments
        execv("/bin/echo", argv);
    } else {
        waitpid(pid, NULL, 0); // Wait for the child process to finish
        printf("Child process finished.\n");
    }

    return 0;
}
```

Running a process(posix_spawn)

- However, fork and execv aren't best practice and have many performance issues and possible footguns, that posix_spawn aims to solve.

```
posix_spawn(&pid, "/bin/echo", NULL, NULL, argv, NULL);
```

Address of our variable to store the pid

Path to our executable to run

Our arguments, every program needs one even if it is just the program name

**environ if required

We can still use waitpid() to wait for the spawned process to finish, and we typically use it to ensure things happen in correct and consistent order.

Example: q2.c

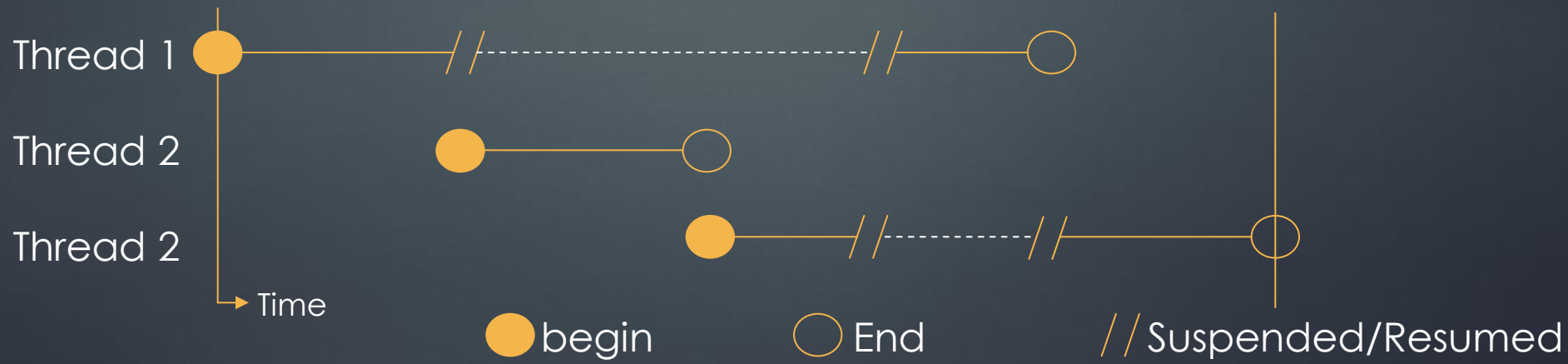
- Write a program q2.c, which spawns a new process to print the following:
  - Current date (date +%d-%m-%Y)
  - Current Time (+%T)
  - Current User (whoami)
  - Current Hostname (hostname –f)
  - Current working directory (realpath .)

THREADS

What is a thread

- Programs typically just run on a single thread, however, with special functions we can split our program across multiple theads

- Surprisingly, this is even beneficial on machines with a single CPU core
  1. When a process is required to wait (user input, syscalls), it can wait in the background
  2. While another different thread can run during this time
  3. This switching between threads is handled by the operating system



Threads (pthread_create)

- Like how we have, posix_spawn() and waitpid() for processes, we have for threads:
  - pthread_create(): creates a new thread
  - pthread_join(): waits for a thread, can capture its function return value

```
pthread_create(&thread, NULL, thread_action, "hello world!\n");
```

address of our pthread_t | our void *function | argument to pass to function

```
pthread_join(thread, &result);
```

our pthread_t | address of our void *variable

- The reason the return types, and the functions we pass in are void * is to allow any types of functions/variables to be used, as C does not have direct generics built into the language
  - This means we would have to cast the value to the expected type

Threads (pthread_create)

- This examples is passing data to a thread, that is running the void *thread_action function, and we receive our return value from the thread through pthread_join

```c
// pthread_create takes in a void *function, to basically allow any type of function to be passed in.
void *thread_action(void *data) {
    // we expect the data to be a string, so we cast it to char*.
    char *str = (char*) data;
    printf("thread received: %s", str);
    return "goodbye world!\n";
}

int main(void) {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_action, "hello world!\n");

    // Similarly the return value of the thread is a void * to allow any type of return value.
    // We can cast it to the type we expect.
    // In this case, we expect a string, so we cast it to char*.
    void *result = NULL;
    pthread_join(thread, &result);
    printf("thread returned: %s\n", (char*) result);
}
```

Example: q5. c

- Write a C program that creates a thread that infinitely prints some message provided by main (e.g “Hello\n”), while the main thread infinitely prints a different message (e.g “there!\n”)

Example: q7. c

- Concurrency can allow programs to perform actions simultaneously that were previously tricky for us, for example we cannot execute code while waiting for input, until now, I guess.

- Write a C program that creates a thread to infinitely print "feed me input\n" once per second (sleep), while the main thread continuously reads in lines of input to print to stdout with the prefix:
  - "you entered: %s"

Example: q6. c

- The following program attempts to say hello from another thread, it compiles correctly, however when run it doesn't print "Hello from thread!\n"
  - Why is this?
  - How do we fix it?

```c
void *thread_run(void *data) {
    printf("Hello from thread!\n");

    return NULL;
}

int main(void) {
    pthread_t thread;
    pthread_create(
        &thread,    // the pthread_t handle that will represent this thread
        NULL,       // thread-attributes -- we usually just leave this NULL
        thread_run, // the function that the thread should start executing
        NULL        // data we want to pass to the thread -- this will be
                    // given in the `void *data` argument above
    );

    return 0;
}
```

- The program does not wait for the thread to finish before exiting from main, so the thread doesn't get time to do its work before the program exits
- To fix this we can use pthread_join in main to wait for the thread to finish

CONCURRENCY

Concurrency and Parallelism

- What is the difference between concurrency and parallelism?
  - Concurrency is a general concept that refers to code running with multiple threads of execution, that may or may not be executing in parallel
  - Parallelism can be thought of as the specific utilisation of concurrency such that multiple threads ARE running truly in parallel, which is usually to increase program performance.

  - This means parallelism is simply not possible without a multi-core capabilities

Concurrency (q8_rc.c)

- We have the following program that aims to use two threads to increment a global variable 5000 times each.
  - We assume since we start at 0 the total will be 10000, however, when running it every time we get different outputs
  - Why?

```c
int global_total = 0;

void *add_5000_to_counter(void *data) {
    for (int i = 0; i < 5000; i++) {
        // sleep for 1 nanosecond
        nanosleep (&(struct timespec){.tv_nsec = 1}, NULL);

        // increment the global total by 1
        global_total++;
    }

    return NULL;
}

int main(void) {
    pthread_t thread1;
    pthread_create(&thread1, NULL, add_5000_to_counter, NULL);

    pthread_t thread2;
    pthread_create(&thread2, NULL, add_5000_to_counter, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // if program works correctly, should print 10000
    printf("Final total: %d\n", global_total);
}
```

```
./q8_rc

Final total: 9997

./q8_rc

Final total: 9998

./q8_rc

Final total: 10000

./q8_rc

Final total: 10000

./q8_rc

Final total: 9997

./q8_rc

Final total: 9996
```

Concurrency (q8_rc.c)

- This is a classic example of a bug in multi-threaded programming called a ‘data race’, which is when multiple threads are modifying a single value, without a synchronisation mechanism
  - As the operation is not atomic each increment is split up into 3 small steps if we think about it as assembly

C

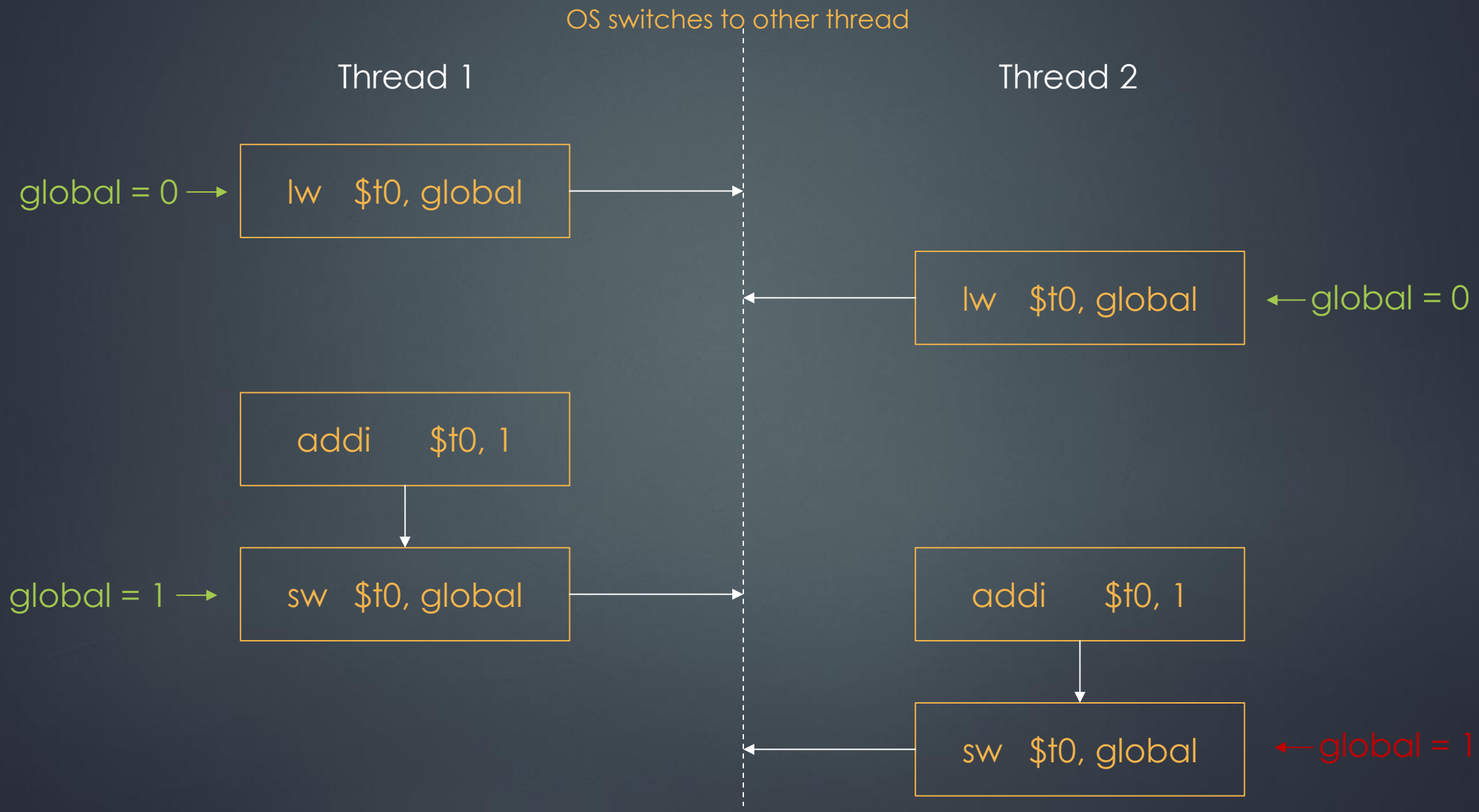
```
global += 1;
```

MIPS

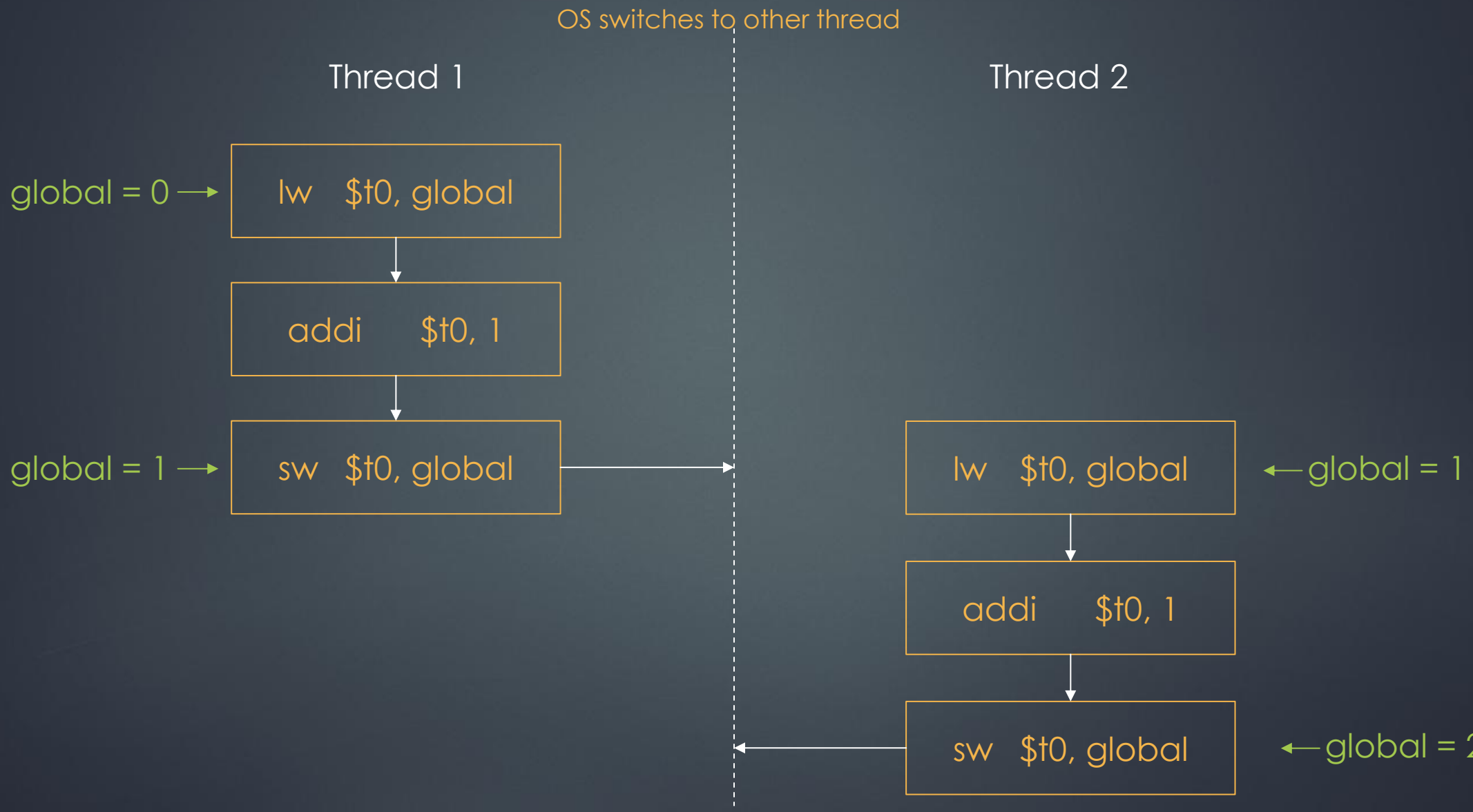
```
lw      $t0, global
addi    $t0, 1
sw      $t0, global
```

- If two threads read the old global_total value, both increment said value, then store that value back, we would lose an increment

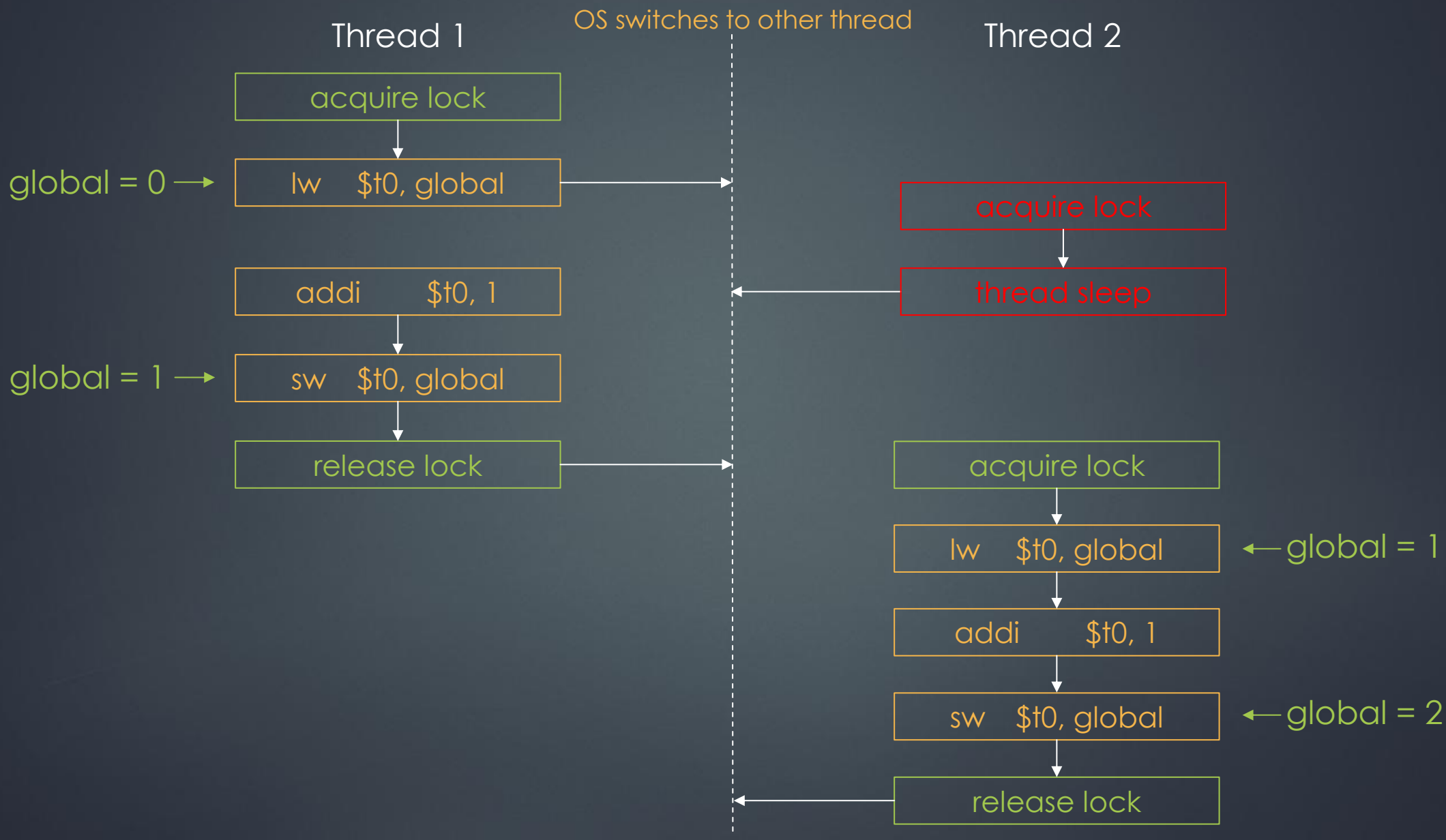
Concurrency (q8_rc.c)



How do we avoid a data race? Desired Behaviour



How do we avoid a data race? Locks (mutex)



Data Races (Locks/Mutex) in C

- We use mutual exclusion (mutex) through locks to make sure only one thread has access to the data at any time, this makes any operation within locks *appear* atomic
- We can do this in C using pthread_mutex_t type to create a lock

```
pthread_mutex_t global_total_mutex = PTHREAD_MUTEX_INITIALIZER
```

- To obtain or release a lock we can use
  - pthread_mutex_lock()
  - pthread_mutex_unlock()

```
pthread_mutex_lock(&global_total_mutex);
pthread_mutex_unlock(&global_total_mutex);
```

Example: q8_mutex.c

- Fix our q8.c solution to eliminate data races using mutual exclusion (mutex)

```c
int global_total = 0;

void *add_5000_to_counter(void *data) {
    for (int i = 0; i < 5000; i++) {
        // sleep for 1 nanosecond
        nanosleep (&(struct timespec){.tv_nsec = 1}, NULL);

        // increment the global total by 1
        global_total++;
    }

    return NULL;
}

int main(void) {
    pthread_t thread1;
    pthread_create(&thread1, NULL, add_5000_to_counter, NULL);

    pthread_t thread2;
    pthread_create(&thread2, NULL, add_5000_to_counter, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // if program works correctly, should print 10000
    printf("Final total: %d\n", global_total);
}
```

Data Races (Atomics)

- Another way for simple data types (ints, bool) to ensure they are updated atomically across threads is to use an atomic type
  - Atomics are types that are incremented in a single atomic instruction avoiding data races
- We can declare an atomic in C:

```
atomic_int global_total = 0;
```

- To increment an atomic int in C we must do it in these two ways, otherwise the operation is not atomic

```
// We must increment with += NOT ++ or var = var + 1
global_total += 1;

// Otherwise we must fetch-and-add function
atomic_fetch_add(&global_total, 1);
```

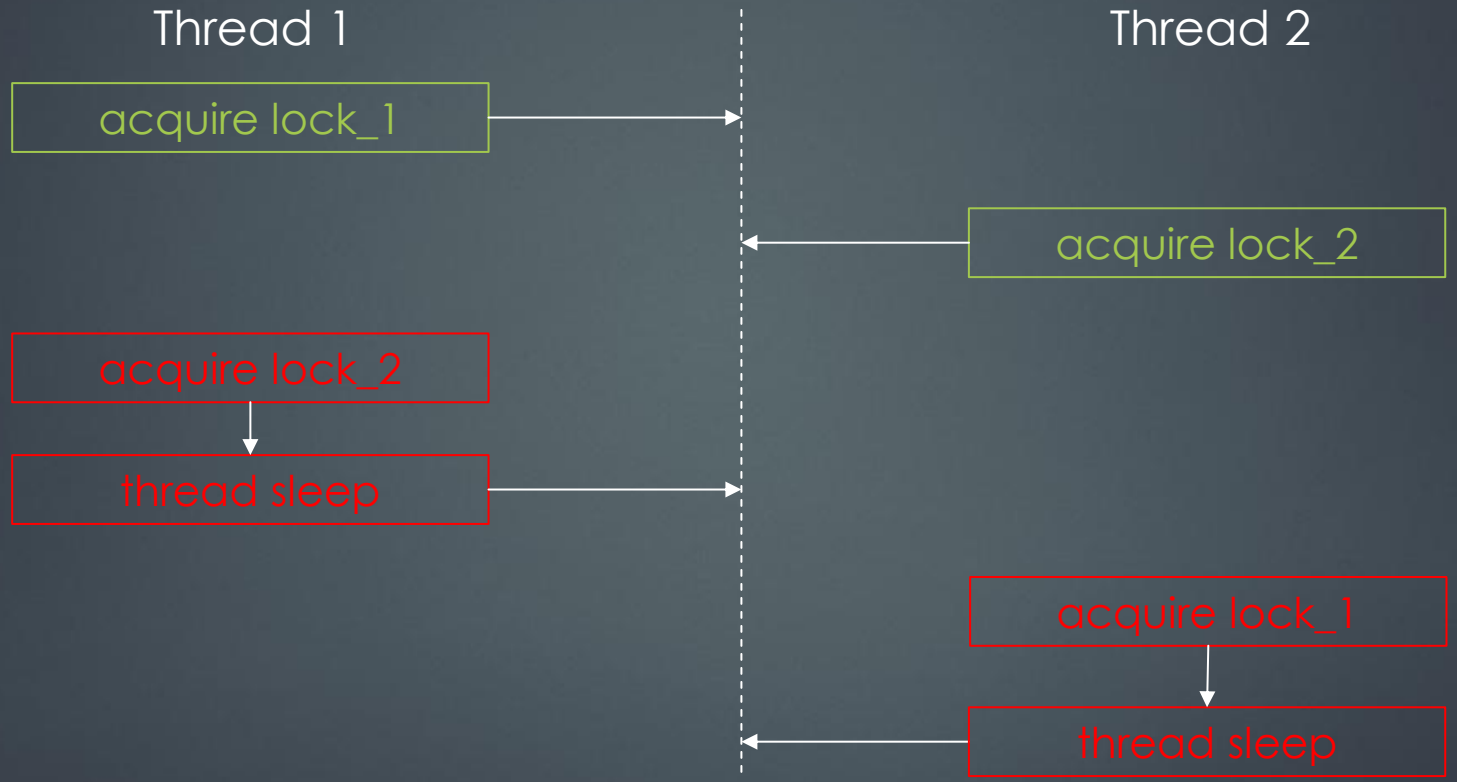
Example: q8_atomic.c

- Fix our q8.c solution to eliminate data races using an atomic int

```c
int global_total = 0;

void *add_5000_to_counter(void *data) {
    for (int i = 0; i < 5000; i++) {
        // sleep for 1 nanosecond
        nanosleep (&(struct timespec){.tv_nsec = 1}, NULL);

        // increment the global total by 1
        global_total++;
    }

    return NULL;
}

int main(void) {
    pthread_t thread1;
    pthread_create(&thread1, NULL, add_5000_to_counter, NULL);

    pthread_t thread2;
    pthread_create(&thread2, NULL, add_5000_to_counter, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // if program works correctly, should print 10000
    printf("Final total: %d\n", global_total);
}
```

Deadlocks

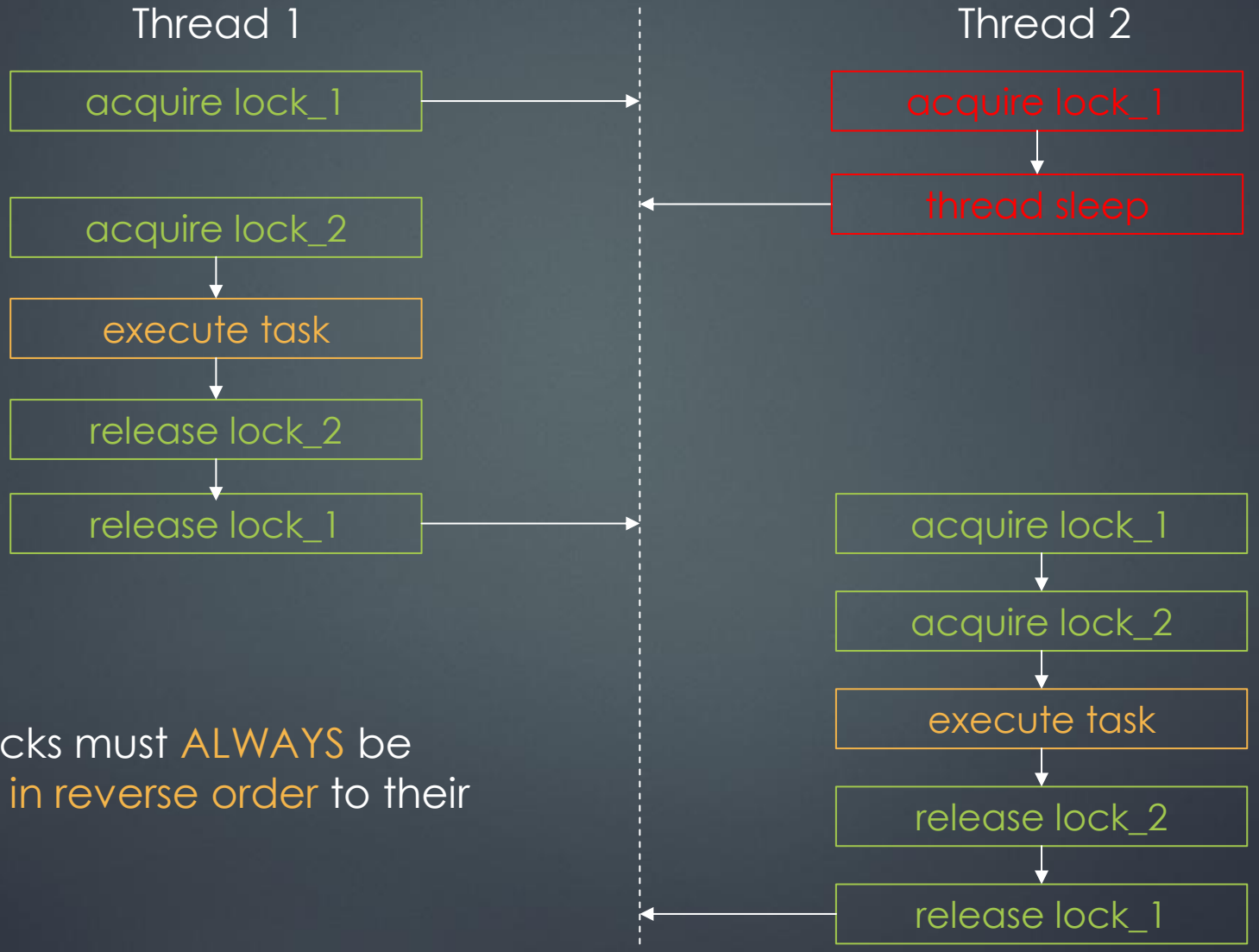
- When we write programs that use multiple locks, we must be careful not to cause 'deadlocks'
  - A deadlock is when threads are endlessly sleeping waiting for a lock to be available, but it never will be, that is because another thread that is holding the lock is waiting on a lock the first one already holds



If threads need multiple resources, it is possible for situations like above to occur

Avoiding Deadlocks

- To avoid deadlocks we must enforce a global ordering of the acquiring of locks, this means any time multiple locks are required, they MUST always be acquired in the same order



NOTE: Locks must ALWAYS be released in reverse order to their retrieval

PIPES (NOT ENOUGH TIME IN ONE WEEK ☹)

FIN