Week 10 – Wednesday

COMP 3400

Last time

- What did we talk about last time?
- Exam 2!
- Before that:
  - Review
- Before that:
  - Thread safety
  - POSIX threads
    - Creating threads
    - Exiting threads
    - Joining threads
    - Passing arguments to threads
    - Reading results from threads

Questions?

Assignment 6

Exam 2 Post Mortem

POSIX Threads

Returning values from threads

- A common model for threads is for them to go and perform some work
- After the work is done, they need to give back the answer
- There are three ways to do this:
  1. Store the answer back into the dynamically allocated struct passed in for its arguments
  2. Use the hack like before to return a "pointer" through the join that's actually an **`int`**
  3. Return a pointer through the join to a dynamically allocated struct containing the answer

Returning in the args struct

```
struct numbers {
  int a;
  int b;
  int sum;
};

void *sum_thread (void *args)
{
  struct numbers *values = (struct numbers*)args;
  values->sum = args->a + args->b;
  pthread_exit (NULL);
}

int main (int argc, char **argv)
{
  pthread_t child;
  struct numbers values = malloc(sizeof(struct numbers));
  values->a = 5;
  values->b = 8;
  pthread_create (&child, NULL, sum_thread, values);
  pthread_join(child, NULL);
  printf ("Sum: %d\n", values->sum);
  free (values);
  pthread_exit (NULL);
}
```

Returning a "pointer" that's an `int`

```
struct numbers {
  int a;
  int b;
};

void *sum_thread (void *args)
{
  struct numbers *values = (struct numbers*)args;
  int sum = args->a + args->b;
  free (values);
  pthread_exit ((void*)sum);
}

int main (int argc, char **argv)
{
  pthread_t child;
  struct numbers *values = malloc(sizeof(struct numbers));
  values->a = 5;
  values->b = 8;
  pthread_create (&child, NULL, sum_thread, values);
  void *sum = NULL;
  pthread_join(child, &sum);
  printf ("Sum: %d\n", (int) sum);
  pthread_exit (NULL);
}
```

Returning a pointer to a dynamically allocated struct

```
struct numbers {
  int a;
  int b;
};

void *calculator (void *args)
{
  struct numbers* values = (struct numbers*)args;
  struct numbers* answers = malloc(sizeof(struct numbers));
  answers->a = values->a + values->b;
  answers->b = values->a - values->b;
  free (values);
  pthread_exit (answers);
}

int main (int argc, char **argv)
{
  pthread_t child;
  struct numbers *values = malloc(sizeof(struct numbers));
  values->a = 5;
  values->b = 8;
  pthread_create (&child, NULL, calculator, values);
  struct numbers *answers = NULL;
  pthread_join(child, (void **)&answers);
  printf ("Sum: %d\nDifference: %d\n", answers->a, answers->b);
  free (answers);
  pthread_exit (NULL);
}
```

Language Approaches to Threading

OpenMP

- All the nitty gritty details of starting threads, sending arguments to them, getting answers back, and joining the threads are annoying
- OpenMP is a library with a set of `#pragma` compiler directives that converts specially formatted code into code that takes care of all the threading details
  - Known as implicit threading, since the programmer doesn't write thread code
- It's ideal for the **fork-join model** where a main thread forks lots of threads to work on parts of a problem and then joins them together, combining their answers
- The book has an example of OpenMP syntax, but I don't want to go into details
- If you do a lot of parallel processing with a simple structure, OpenMP can be worth learning

Object-oriented approaches

- Java, C#, Python, and many other newer languages encapsulate threads as objects
- Data can be provided in the object's constructor
- Methods can be used to read data after the thread has finished running
- Special methods are reserved for starting and joining threads

Java threading example

- The following Java class extends **`Thread`** and is designed to sum up part of an array

```
public class Summer extends Thread {
    private double[] array;
    private int lower;
    private int upper;
    private double sum = 0;

    public Summer(double[] array, int lower, int upper) {
        this.array = array;
        this.lower = lower;
        this.upper = upper;
    }

    public void run() {
        for(int i = lower; i < upper; i++)
            sum += array[i];
    }

    public double getSum() { return sum; }
}
```

Java threading example continued

- The following Java method uses the class from the previous slide to sum up parts of an array in parallel

```
public double sum(double[] array, int threads) throws InterruptedException {
    // Only works if length is evenly divisible
    int stride = array.length / threads;
    Summer[] workers = new Summer[threads];
    for(int i = 0; i < threads; ++i) {
        workers[i] = new Summer(array, i*stride, (i + 1)*stride);
        workers[i].start();
    }

    double result = 0.0;
    for(int i = 0; i < threads; ++i) {
        workers[i].join();
        result += workers[i].getSum();
    }

    return result;
}
```

Modern languages

- Although Java is relatively new, it was designed before the advent of ubiquitous multicore processors
- Threads are still accessed via a library rather than being part of the core language
- Modern languages like Rust and Go have keywords associated with threading

Threading in Go

- Merely putting the keyword **go** in front of a function makes it run on a new thread

```
func main() {
	// Create a channel for communication
	messages := make(chan string)

	fmt.Print("Guess a number between 1 and 10: ")

	// Start keyboard listener as a goroutine with the channel
	go keyboard_listener(messages)

	// Wait until there is data in the channel
	success := <-messages
	if success == "true" {
		fmt.Println("You must have guessed 7.")
	}
}
```

Threading in Rust

- Rust is a new language that competes with C/C++ in systems programming
- It's finicky about ownership
- The **`move`** command in the following code gives the closure its own copy of **`x`** at the current value

```
fn main() {
    let mut x = 10; // Initialize a mutable variable x to 10

    // Spawn a new thread
    let child_thread = thread::spawn(move || {
        thread::sleep(time::Duration::from_secs(1)); // Sleep for one second
        println!("x = {}", x); // Print x
    });

    // Change x in the main thread and print it
    x += 1;
    println!("x = {}", x);

    // Join the thread and print x again
    child_thread.join();
    println!("x = {}", x);
}
```

Concurrent prime number search

- Let's write a threaded program that counts the number of primes less than 100,000,000
- We'll spawn a number of threads and divide up the range of values from 0 to 100,000,000 evenly
- To send data to each thread and get the result, we'll use dynamically allocated versions of the following struct:

```
struct range {
  unsigned long start;
  unsigned long end;
  unsigned long count;
};
```

POSIX thread functions

- As a reminder, here are the POSIX functions we need

```
int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
      void *(*start_routine)(void*), void *arg);
```

  - Create a new thread (not as bad as it looks)

```
void pthread_exit (void *value_ptr);
```

  - Exit from the current thread (giving a pointer to the result, if any)

```
void pthread_join (pthread_t thread, void *value_ptr);
```

  - Join a thread (getting a pointer to its result, if any)

Algorithm

- Divide the total number by the number of threads to determine how many numbers to give each thread
- Loop through all threads:
  - Allocate a **`range`** struct to hold the lower and upper value for each thread
  - Create each thread
- Loop through all threads:
  - Join them
- Inside each thread:
  - Loop from the lower to the upper value and increment a counter if the value is prime
  - Store the count into the **`range`** struct
  - Call **`pthread_exit()`** when done

Upcoming

Next time…

- Synchronization and critical sections
- Locks

Reminders

- **Finish Assignment 6**
  - **Due Friday before midnight**
- Start working on Project 3 as soon as you can
- Read sections 7.2 and 7.3