Week 9 - Wednesday

COMP 3400

Last time

- What did we talk about last time?
- Finished Internet layer
- Link layer
- Wireless
- Started threads

Questions?

Assignment 5

Race Conditions

Race conditions

- Race conditions are a central problem with threads
- Thread scheduling is non-deterministic
  - It's often impossible to predict when the statements from one thread are going to be executed with respect to those in another thread
  - If the statements modify the same memory, the results can be inconsistent
- One of the most frustrating issues with race conditions is that they can occur rarely
  - This means that you can run your program 1,000 times with no problems, only to crash badly on time 1,001

Race condition scenarios

- The following are common causes of race conditions:
  - Two or more threads trying to modify a global variable at the same time
  - One thread calls `free()` on data that another thread is using
  - Thread A is using variables declared on the stack of Thread B, which become invalid when Thread B terminates
  - Two or more threads calls a non-thread-safe function at the same time

Critical sections

- A **critical section** is a series of statements that *must* be executed atomically to get the right result
- **Atomic** execution means that all the statements happen as if they happened at once, without other statements from other threads interfering
- Even statements that look atomic like **`i++`** are actually several different operations in assembly language

```
movq   _globalvar(%rip), %rsi     # copy from memory into %rsi register
addq   $1, %rsi                   # increment the value in the register
movq   %rsi, _globalvar(%rip)     # store the result back into memory
```

Incrementing variables

- Consider two threads that share an **`int`** variable called **`global`** that is initially set to **0**:

Thread A

```
for (int i = 0; i < 200; ++i)
    ++global;
```

Thread B

```
for (int j = 0; j < 300; ++j)
    ++global;
```

- What are the largest and smallest values that **`global`** could have after these threads run to completion?

Thread safety

- Many functions are **thread safe**, meaning that they can be called by many threads at the same time and still give the right answers
- Other functions are not thread safe
- The usual reason that functions are not thread safe is because they contain static local variables
- Because these variables are shared by all threads, they can become corrupted

Non-thread safe function (innocent version)

- The **`rand()`** function *isn't* thread safe
- Internally, it keeps a value for the next random number
- If two threads call **`rand()`**, they won't get the sequence of random numbers they're supposed to
- Strange, but it doesn't matter too much in this case since the numbers are supposed to be random

Example with rand()

SINGLE THREADED VERSION

```
int a = rand() % 5;   // 3
int b = rand() % 5;   // 0
int c = rand() % 5;   // 1
```

TWO THREADS RUNNING (SPACING SHOWS EXECUTION ORDER)

```
int a = rand() % 5;   // 3

int b = rand() % 5;   // 1

int c = rand() % 5;   // 2
```

```

int d = rand() % 5;   // 0

int e = rand() % 5;   // 4

```

Non-thread safe function (terrifying version)

- The **`strtok()`** function *isn't* thread safe
- This function is used to divide up a string by some delimiter
- The first time you call it, you give it the string you're trying to divide
- For future calls, you call it with **`NULL`**, and it uses the location it's at in the string you gave it before

```
char[] sentence = "bears beets Battlestar Galactica";
char* word = strtok (sentence, " ");
while (word != NULL)
{
	printf ("%s\n", word); // Prints each word on a separate line
	word = strtok (NULL, " ");
}
```

Example with `strtok()`

- As before, two threads, and the spacing shows execution order

Thread A

```
char[] sentence = "bears beets";
char* word = strtok (sentence, " ");
printf ("%s\n", word); // bears


word = strtok (NULL, " ");
printf ("%s\n", word); // streets
```

Thread B

```



char[] phrase = "mean streets";
char* thing = strtok (phrase, " ");
```

How you can prevent race conditions

- We will spend quite bit of time in this class discussing tools that can be used
- For now, be careful about not using non-thread safe functions
- Both **`rand()`** and **`strtok()`** have **reentrant** versions
  - **`rand_r()`** and **`strtok_r()`**
  - Instead of keeping data as static variables, the reentrant versions require you to pass the current state back to them as an extra variable
  - Slightly annoying but *so much* safer
- Reentrant functions are usually thread safe because they can be interrupted

POSIX Threads

POSIX threads

- Just as we could create a new process with **`fork()`**, there are libraries for making new threads
- POSIX threads (also called pthreads) are perhaps the most widely used thread library
  - Windows (of course) has its own threading library, though people have built POSIX-like libraries on top of it
- Key POSIX concepts
  - Creating a thread starts it running
  - A thread can exit, stopping its running
  - Joining a thread means waiting for a thread to finish (and potentially getting its result)
  - We keep track of processes with an ID of type **`pid_t`**, but we keep track of threads with an ID of type **`pthread_t`**

POSIX thread functions

- Here are POSIX functions mapping to concepts from the previous slide

```
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)

Creating a thread

- Creating a thread is the most complicated function, partly because it takes a function pointer and potentially arguments

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

  - **`thread`** is a pointer to a **`pthread_t`** that will get filled in with the thread's ID
  - **`attr`** is a pointer to possible thread attributes (often left **`NULL`**)
  - **`start_routine`** is a pointer to a function that takes a **`void*`** and returns a **`void*`**
  - **`arg`** is a pointer to arguments, **`NULL`** if no arguments needed

Simple threading example

```
#include <stdio.h>
#include <pthread.h>        // POSIX thread library
#include <assert.h>

void *
start_thread (void *args)   // Function to start thread with
{
  printf ("Hello from thread!\n");
  pthread_exit (NULL);
}

int
main (int argc, char **argv)
{
  pthread_t child_thread;

  // Create new thread with function start_thread
  assert (pthread_create (&child_thread, NULL, start_thread, NULL) == 0);

  pthread_join (child_thread, NULL);   // Wait for other thread to finish
  pthread_exit (NULL);                 // main() exits like any other thread
}
```

Common mistakes

- Passing in a garbage **`pthread_t*`** instead of the address of a real **`pthread_t`**

```
pthread_t *thread; // No!
```

- Calling the threading function (with parentheses) instead of passing a function pointer in

```
pthread_create (thread, NULL, start (), NULL); // No!
```

- Joining with a **`pthread_t*`** instead of a **`pthread_t`**

```
pthread_join (thread, NULL); // No!
```

Attached and detached threads

- Normal threads are attached, meaning that they can be joined
- It's possible to create detached threads, which can never be joined
  - By passing in a **`pthread_attr_t`** struct with the right options
  - Or by calling **`pthread_detach()`** on a thread's ID
- Note that you can get your own ID by calling the **`pthread_self()`** function

```
pthread_t pthread_self (void);
```

Passing arguments

- Passing arguments to threads is tricky
  - Passing addresses to objects on the stack is dangerous in case the function creating the threads returns
  - Passing pointers to the same object to multiple threads can cause problems if they fight over it
  - There are no timing guarantees over which thread will run when

A useful hack

- On most modern machines, a pointer is either 32 bits or 64 bits
- An **`int`** is usually 32 bits
- We can cast an **`int`** to a pointer and pass that to the thread
- The thread will then cast the pointer back to an **`int`**
- Since the size of an **`int`** is almost always less than a pointer, we don't lose any information
- It's icky, but it allows us to pass simple values like a **`char`**, **`short`**, or **`int`**
  - Both floating-point types are harder since they have to be tricked into behaving like integers (which pointers fundamentally are)
  - And **`double`** is risky since it needs a 64-bit pointer to hold it all

A thread function that uses a pointer like an `int`

```
void * child_thread (void *args)
{
  int value = (int) args; // Now, I pretend it's an int!
  printf ("I'm a thread with value: %d\n", value);
  pthread_exit (NULL);
}

int main (int argc, char **argv)
{
  pthread_t threads[10];  // Array to hold thread IDs

  // Start up those threads, pretending ints are pointers
  for (int i = 0; i < 10; i++)
    pthread_create (&threads[i], NULL, child_thread, (void*)i);

  for (int i = 0; i < 10; i++)
    pthread_join(threads[i], NULL);
  pthread_exit (NULL);
}
```

Passing multiple arguments to a thread

- To pass multiple arguments, they're often grouped in a struct
- Remember that threads all have their own stacks
- Thus, we need to pass in a struct that has been dynamically allocated on the heap (which is shared)
  - Also, any pointers that struct contains should point at memory that isn't on the stack

Multiple argument example

```
struct thread_args
{
  int value;
  const char* string;
};

int main (int argc, char **argv)
{
  pthread_t thread;
  struct thread_args* args = malloc(sizeof(struct thread_args));
  args->value = 42;
  args->string = "wombat";

  // Thread casts void* to struct thread_args* when it gets it
  pthread_create (&thread, NULL, child_thread, args);

  pthread_join(thread, NULL);
  pthread_exit (NULL);
}
```

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 = values->a + values->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 = values->a + values->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(result));
  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);
}
```

Ticket Out the Door

Upcoming

Next time…

- Review for Exam 2

Reminders

- **Exam 2 on Monday!**
- Finish Assignment 5
  - Due Friday by midnight!
- Read sections 6.6, 6.8, 7.1, and 7.2