CS170 Lecture notes -- Introduction to Threads

- Directory: **/cs/faculty/rich/public_html/class/cs170/notes/IntroThreads**
- Lecture notes: http://www.cs.ucsb.edu/~rich/class/cs170/notes/IntroThreads/index.html
- examples: http://www.cs.ucsb.edu/~rich/class/cs170/notes/IntroThreads/example or on github in https://github.com/richwolski/cs170-lecture-examples.git in the IntroThreads subdirectory

---

Introduction

We'll spend quite a bit of time in this class discussing the concept of "*concurrency*" -- the notion that independent sets of operations can be occurring at the same time inside the machine when it is executing a program. For example, when a program does I/O (say to a disk), the CPU may be switched that it is able to work on some other task while the I/O is taking place. Operating systems both allow the user to manage concurrency and also (in many cases) exploit concurrency on behalf of the user to improve performance. As a result, concurrency and concurrency management/control will be themes that recur throughout this course.

Threads

Threads are a programming abstraction that is designed to allow a programmer to control concurrency and asynchrony within a program. In some programming languages, like Java, threads are "first class citizens" in that they are part of the language definition itself. For others, like C and C++, threads are implemented as a library that can be called from a program but otherwise are not considered part of the language specification.

Some of the differences between having threads "in the language" and threads "as a library" are often subtle. For example, a C compiler need not take into account thread control while a Java compiler must. However one obvious difference is that in the library case, it is possible to use different thread libraries with the same language. In this class, we'll be programming in C, and we'll use both POSIX Threads and a thread library specifically designed for the OS project called *Kthreads*. Kthreads and POSIX threads are similar in that they are both thread abstractions and they are both implemented as libraries that can be called from a C program. They are different in that POSIX threads requires operating system support to work properly and, thus, can't be used directly to implement the operating system. In contrast, Kthreads can be implemented without the OS using only the C language compiler and a little bit of the C runtime. For this reason, we can use Kthreads as an abstraction with which to build an operating system (i.e. there is no circular dependence).

We'll study both before the end of the class but we'll start with POSIX threads since they a standard.

So what is a Thread?

There are many different kinds of thread abstractions. In this class, we'll use the typical "OS thread" abstraction that underpins POSIX threads, but particularly for language-defined threads, different definitions are possible.

Abstractly, for our purposes, a thread is three things:

- a **sequential list of instructions** that will be executed
- a set of **local variables** that "belong" to the thread (thread private)
- a set of **shared global variables** that all threads can read and write

It is no accident that this definition corresponds roughly to the C language sequential execution model and variable scoping rules. Operating systems are still, for the most part, written in C and thus thread libraries for C are easiest to understand and implement when they conform to C language semantics.

Threads versus Processes

Recall from your C programming experiences, that your compiled program becomes a "process" when you run it. We'll discuss processes and what they really are at length, but at this stage, it is enough to know that a C program runs as a process when it is executed on, say, a Linux system.

A C program also defines a sequential list of instructions, local variables, and global variables so you might be asking *"What is the difference between a thread and a process?"*

The answer is "not much" as long as there is only one thread. However, as discussed previously, threads are an abstraction designed to manage concurrency which means it is possible to have multiple threads "running" at the same time. Put another way,

A standard C program when executing is a process with one thread.

However, it is possible (using a thread library) to write a C program that defines multiple threads. That is

A threaded C program, when executing, is a process that contains one or more threads.

Furthermore, these threads are logically independent and thus *may* be executed concurrently. They don't have to be (it depends on the implementation) but the abstraction says that the threads are independent.

Why threads?

There are many reasons to program with threads. In the context of this class, there are two important ones:

- They allow you to deal with asynchronous events synchronously and efficiently.
- They allow you to get parallel performance on a shared-memory multiprocessor.

You'll find threads to be a big help in writing an operating system.

A Simple Example

Before we dive into an anatomical and physiological exploration of POSIX threads, which heretofore will be referred to as **pthreads**, it is probably helpful to walk through a simple example. Pthreads are widely used and their full interface is somewhat complicated. We'll eventually discuss much of it, but "the basics" as most easily understood through an example.

C Code, No threads

To begin with, consider a simple program that computes the average over a set of random numbers. In the following examples we'll use a Linux-internal random number generator rather than numbers from a file to keep the code easier to read. You might also think that you know what the answer will be ahead of time. For example, if the random number generator generates numbers on the interval (0,1) then you'd expect the average to be 0.5. How true is that statement? Is it affected by the number of numbers in the set? This example can also be used to investigate these kinds of questions but mostly it is designed to introduce the way in which pthreads and C interact.

The basic program generates an array that it fills with random numbers from the interval (0,1). It then sums the values in the array and divides by the number of values (which is passed as an argument from the command line).

Here is the C code. I've put a commented version of the code in http://www.cs.ucsb.edu/~rich/class/cs170/notes/IntroThreads/example/avg-nothread.c that also includes argument sanity checks. To improve readability in these notes, however, the in-lined code will remove parts that are good practice but don't shed light on the use of threads. We'll also describe how to build and run the examples in this lecture in its last section.

Here is the code

```
#include < unistd.h >
#include < stdlib.h >
#include < stdio.h >

/*
 * program to find the average value of a set of random numbers
 *
 * usage: avg-nothread count
 *
 * where count is the number of random values to generate
```

```
 */

char *Usage = "usage: avg-nothread count";

#define RAND() (drand48()) /* basic Linux random number generator */

int main(int argc, char **argv)
{
        int i;
        int n;
        double *data;
        double sum;
        int count;

        count = atoi(argv[1]);  /* count is first argument */

        /*
         * make an array large enough to hold #count# doubles
         */
        data = (double *)malloc(count * sizeof(double));

        /*
         * pick a bunch of random numbers
         */
        for(i=0; i < count; i++) {
                data[i] = RAND();
        }

        sum = 0;
        for(i=0; i < count; i++) {
                sum += data[i];
        }

        printf("the average over %d random numbers on (0,1) is %f\n",
                        count, sum/(double)count);

        return(0);
}
```

There are a few C language features to note in this simple program. First, it uses the utility function *malloc()* to allocate dynamically a one-dimensional array to hold the list of random numbers. It also casts the variable *count* to a double in the print statement since the variable *sum* is a double.

Now is a good time to take a moment to make sure that you understand each line of the program shown above -- **each line.** If there is something you don't recognize or understand in this code you will want to speak with the instructor or the TAs about brushing up on your C programming skills. This code is about as simple as any C program will be that you will encounter in this class. If it isn't completely clear to you it will be important to try and brush up because the assignments will depend a working knowledge of C.

Computing the Average using One Thread

The next program performs the same computation, but does so using a single thread rather than in the main body as in the previous program. The full version of this program is available from http://www.cs.ucsb.edu/~rich/class/cs170/notes/IntroThreads/example/avg-1thread.c

```
#include < unistd.h >
#include < stdlib.h >
#include < stdio.h >

#include < pthread.h >


char *Usage = "usage: avg-1thread count";

#define RAND() (drand48()) /* basic Linux random number generator */

struct arg_struct
{
        int size;
        double *data;
};

struct result_struct
```

```
{
        double sum;
};

void *SumThread(void *arg)
{
        int i;
        double my_sum;
        struct arg_struct *my_args;
        int my_size;
        double *my_data;
        struct result_struct *result;

        printf("sum thread running\n");
        fflush(stdout);

        my_args = (struct arg_struct *)arg;
        result = (struct result_struct *)malloc(sizeof(struct result_struct));

        my_size = my_args->size;
        my_data = my_args->data;

        free(my_args);

        my_sum = 0.0;
        for(i=0; i < my_size; i++) {
                my_sum += my_data[i];
        }

        result->sum = my_sum;
        printf("sum thread done, returning\n");
        fflush(stdout);

        return((void *)result);
}

int main(int argc, char **argv)
{
        int i;
        int n;
        double *data;
        int count;
        struct arg_struct *args;
        pthread_t thread_id;
        struct result_struct *result;
        int err;

        count = atoi(argv[1]);  /* count is first argument */


        data = (double *)malloc(count * sizeof(double));

        for(i=0; i < count; i++) {
                data[i] = RAND();
        }

        args = (struct arg_struct *)malloc(sizeof(struct arg_struct));

        args->size = count;
        args->data = data;

        printf("main thread forking sum thread\n");
        fflush(stdout);

        err = pthread_create(&thread_id, NULL, SumThread, (void *)args);

        printf("main thread running after sum thread created, about to call join\n");
        fflush(stdout);

        err = pthread_join(thread_id,(void **)&result);

        printf("main thread joined with sum thread\n");
        fflush(stdout);

        printf("the average over %d random numbers on (0,1) is %f\n",
                        count, result->sum / (double)count);

        free(result);
        free(data);
```

```
        return(0);
}
```

In pseudocode form, the logic is as follows. The *main()* function does

```
allocate memory for thread arguments
fill in thread arguments (marshal the arguments)
spawn the thread
wait for the thread to complete and get the result (the sum in this example) computed by the thread
print out the average
```

and the thread executes

```
unmarshal the arguments
compute and marshal the sum so it can be returned
return the sum and exit
```

Reading through the Code

The first thing to notice is that the code that computes the sum is performed in a separate C function called *SumThread()*. The pthreads standard specifies that threads begin on function boundaries. That is, each thread starts with some function call which, in this example, is *SumThread()*. This "first" function can call other functions, but it defines the "body" of the thread. We'll call this first function the "entry point" for the thread.

The second thing to notice are the types in the prototype for the thread entry point:

```
void *SumThread(void *arg)
```

The standard as implemented for Linux specifies that

- the entry point funtion take one argument that is of type *(void *)*
- the entry point function return a single argument that if of type *(void *)*

This typing specification can cause some confusion if you are not entirely comfortable with C pointers so it is important to try and understand why it is defined this way. In C, a *(void *)* pointer is that can legally point to **any** data type. The key is that your program can make the decision about what it points to at run time. That is, a a *(void *)* tells the compiler that your program will determine the type at run time under program control.

You can use a *(void *)* pointer to point to any type that is supported by C (e.g. *int*,*double*,*char*, etc.) but it is most useful when it is used to point to a structure.

A structure is a way for you to define your own composite data type. In a pthreads program, the assumption that the API designers make is that you will define your own data type for the input parameters to a thread and also one for the return values. This way you can pass what ever arguments you like to your threads and have them return arbitrary values.

In this example, we define two structures

```
/*
 * data type definition for arguments passed to thread
 */
struct arg_struct
{
        int size;
        double *data;
};

/*
 * data type definition for results passed back from threads
 */
struct result_struct
{
        double sum;
};
```

The argument structure allows the code that spawns the thread to pass it two arguments: the size of the array of values and a pointer to the array. The thread passes back a single value: the sum.

Notice that the thread entry point function converts its one argument to a pointer to the argument data structure. The data type for *my_args* is

```
struct arg_struct *my_args;
```

and the body of the thread assigns the *arg* pointer passed as an argument to *my_args* via a C language cast.

```
my_args = (struct arg_struct *)arg;
```

You can think of the thread as receiving a message with its initial arguments in it as its only parameter. This message comes in a generic "package" with the type *(void *)* and it is the thread's job to unpack the message into a structure that it understands. This process is called "unmarshaling" which refers to the process of translating a set of data types from a generic tranport form to one that can be processed locally. Thus the line shown above in which the *(void *)* is cast to a *(struct arg_struct *)* is the thread "unmarshaling" its arguments.

Similarly, when the thread has finished computing the sum, it needs a data structure to pass back to a thread that is waiting for the result. The code calls *malloc()* to allocate the memory necessary to transmit the results once the thread has completed:

```
result = (struct result_struct *)malloc(sizeof(struct result_struct));
```

and when the sum is computed, the thread loads the sum into the result structure:

```
result->sum = my_sum;
```

The marshaling into a *(void *)* of the *(struct result_struct *)* takes place directly in the return call

```
return((void *)result);
```

From the *my_args* variable, the thread can then access the *size* value and the *data* pointer that points to the array of numbers. Notice that the very next thing the thread does is to call *free()* on the *arg* pointer. In a C program it is essential that you keep track of how memory is allocated an freed. Good practive is to free memory as soon as you know the memory is no longer needed. In this example, the code that creates this thread in the *main()* function calls *malloc* to allocate the memory that is needed to hold the argument structure.

Notice that the thread has called *malloc()* to create a result variable to pass back the sum. It must be the case that the *main()* thread calls *free()* on the result structure it gets back from the thread. Look for the *free()* call in the *main()* routine to see where this takes place.

Creating a thread

The *main()* function creates an argument structure, spawns the thread, waits for it to complete, and uses the result that the thread passes back to print the average.

Creating and marshaling the arguments for the thread:

```
args = (struct arg_struct *)malloc(sizeof(struct arg_struct));
args->size = count;
args->data = data;
```

The thread that computes the sum is created by the *pthread_create()* call in the *main()* function.

```
err = pthread_create(&thread_id, NULL, SumThread, (void *)args);
```

The *pthread_create()* call takes four arguments and returns a single result. The arguments are

- a pointer to a variable of type *pthread_t* (as an out parameter)
- a pointer to a structure indicating how to schedule the thread (NULL means use the default scheduler)
- the name of the entry point function
- a single pointer to the arguments as a *(void *)*

The return value is an error code, with zero indicating success. If the return value is zero, the variable pointed to by the

first argument will contain the thread identifier necessary to interact with the thread (see *pthread_join()* below).

When does the thread run?

Logically, the thread begins executing as soon as the call to *pthread_create()* completes. However it is up to the implementation as to when the thread is actually scheduled. For example, some implementations will allow the spawning thread to continue executing "for a while" before the spawned threads begin running. However, from a logical perspective, the newly created thread and the thread that created it are running "in parallel."

Notice also that the *main()* function is acting like a thread even though it wasn't spawned via *pthread_create()*. Under Linux, the program that begins executing before any threads are spawned is, itself, a thread. The logical abstraction is that Linux "spawned" this first thread for you. That is, when using pthreads, the function *main()* behaves as if it has been spawned by Linux. It is a little different since it takes two arguments, but for thread scheduling purposes, it behaves like a thread otherwise. We'll call this thread "the main thread" from now on to indicate that it is the "first" thread that gets created when the program begins to run.

Thus the main thread in this example spawns a single thread to compute the sum and waits for this thread to complete before proceeding.

Waiting for the result

After the main thread spawns the thread to compute the sum, it immediately calls

```
err = pthread_join(thread_id,(void **)&result);
```

The first argument to this call is the identifier (filled in by the call to *pthread_create()* when the thread was created. The second argument is an out parameter of type *(void **)*. That is, *pthread_join()* takes a pointer to a *(void *)* so that it can return the *(void *)* pointer passed back from the thread on exit.

This "pointer to a pointer" parameter passing method often confuses those new to pthreads. The function *pthread_join()* needs a way to pass back a *(void *)* pointer and it can't use the return value. In C, the way that a function passes back a pointer through an out parameter is to take a pointer to that kind of pointer as a parameter. Notice that the type of *result* is *(struct result_struct *)*. My using the *&* operator, the parameter passed is the address of result (which is a pointer) and that "pointer to a pointer" is cast as a *(void **)*.

Like with *pthread_create()*, *pthread_join()* returns an integer which is zero on success and non zero when an error occurs.

Here is the output

```
./avg-1thread 100000
main thread forking sum thread
main thread running after sum thread created, about to call join
sum thread running
sum thread done, returning
main thread joined with sum thread
the average over 100000 random numbers on (0,1) is 0.499644
```

Notice that the main thread continues to run after the Sum thread is spawned. Then it blocks in *pthread_join()* waiting for the su thread to finish. Then the sum thread runs and finishes. When it exits, the call to *pthread_join()* unblocks and the main thread completes.

Computing the sum in parallel

The previous example is a little contrived in that there is no real advantage (and probably a very small performance penalty) in spawning a single thread to compute the sum. That is, the first non-threaded example does exaxtly what the single threaded example does only without the extra work for marshaling and unmarshaling the arguments and spawninga and joining. You might ask, then, "why use threads at all?"

The answers is that it is possible to compute some things in parallel using threads. In this example, we can modify the sum thread so that it works on a subregion of the array. The main thread can spawn multiple subregions (which are computed in parallel) and then sum the sums that come back to get the full sum. The following example code does this parallel computation of the sums.

```
#include < unistd.h >
#include < stdlib.h >
#include < stdio.h >

#include < pthread.h >

char *Usage = "usage: avg-manythread count threads";

#define RAND() (drand48()) /* basic Linux random number generator */

struct arg_struct
{
        int id;
        int size;
        double *data;
        int starting_i;
};

struct result_struct
{
        double sum;
};

void *SumThread(void *arg)
{
        int i;
        double my_sum;
        struct arg_struct *my_args;
        int my_size;
        double *my_data;
        struct result_struct *result;
        int my_start;
        int my_end;
        int my_id;

        my_args = (struct arg_struct *)arg;
        result = (struct result_struct *)malloc(sizeof(struct result_struct));

        printf("sum thread %d running, starting at %d for %d\n",
                my_args->id,
                my_args->starting_i,
                my_args->size);
        fflush(stdout);

        my_id = my_args->id;
        my_size = my_args->size;
        my_data = my_args->data;
        my_start = my_args->starting_i;

        free(my_args);

        my_end = my_start + my_size;

        my_sum = 0.0;
        for(i=my_start; i < my_end; i++) {
                my_sum += my_data[i];
        }

        result->sum = my_sum;

        printf("sum thread %d returning\n",
                my_id);
        fflush(stdout);

        return((void *)result);
}

int main(int argc, char **argv)
{
        int i;
        int t;
        double sum;
        double *data;
        int count;
        int threads;
        struct arg_struct *args;
        struct result_struct *result;
        int err;
        pthread_t *thread_ids;
```

```
        int range_size;
        int index;


        count = atoi(argv[1]);  /* count is first argument */


        threads = atoi(argv[2]); /* thread count is second arg */

        data = (double *)malloc(count * sizeof(double));

        for(i=0; i < count; i++) {
                data[i] = RAND();
        }


        thread_ids = (pthread_t *)malloc(sizeof(pthread_t)*threads);

        range_size = (count / threads) + 1;

        if(((range_size-1) * threads) == count) {
                range_size -= 1;
        }

        printf("main thread about to create %d sum threads\n",
                        threads);
        fflush(stdout);
        index = 0;
        for(t=0; t < threads; t++) {
                args = (struct arg_struct *)malloc(sizeof(struct arg_struct));
                args->id = (t+1);
                args->size = range_size;
                args->data = data;
                args->starting_i = index;
                if((args->starting_i + args->size) > count) {
                        args->size = count - args->starting_i;
                }
                printf("main thread creating sum thread %d\n",
                        t+1);
                fflush(stdout);
                err = pthread_create(&(thread_ids[t]), NULL, SumThread, (void *)args);
                printf("main thread has created sum thread %d\n",
                        t+1);
                index += range_size;
        }

        sum = 0;
        for(t=0; t < threads; t++) {
                printf("main thread about to join with sum thread %d\n",t+1);
                fflush(stdout);
                err = pthread_join(thread_ids[t],(void **)&result);
                printf("main thread joined with sum thread %d\n",t+1);
                fflush(stdout);
                sum += result->sum;
                free(result);
        }

        printf("the average over %d random numbers on (0,1) is %f\n",
                        count, sum / (double)count);

        free(thread_ids);
        free(data);

        return(0);
}
```

In this example, each thread is given a starting index into the array and a range it needs to cover. It sums the values in that range and returns the partial sum in the result result structure.

The main thread spawns each thread one at a time in a loop, giving each its own argument structure. Notice that the argument structure is filled in with the starting index where that thread is supposed to start and the range of values it needs to cover.

Aftre the main thread spawns all of the sum threads, it goes into another loop and joins with them, one-at-a-time, in the roder that they were spawned.

Here is a a sample output from this multi-threaded program:

```
MossPiglet% ./avg-manythread 100000 5
main thread about to create 5 sum threads
main thread creating sum thread 1
main thread has created sum thread 1
main thread creating sum thread 2
main thread has created sum thread 2
main thread creating sum thread 3
main thread has created sum thread 3
main thread creating sum thread 4
sum thread 1 running, starting at 0 for 20000
main thread has created sum thread 4
sum thread 2 running, starting at 20000 for 20000
sum thread 3 running, starting at 40000 for 20000
main thread creating sum thread 5
sum thread 4 running, starting at 60000 for 20000
sum thread 1 returning
main thread has created sum thread 5
main thread about to join with sum thread 1
sum thread 5 running, starting at 80000 for 20000
sum thread 2 returning
sum thread 3 returning
sum thread 4 returning
main thread joined with sum thread 1
main thread about to join with sum thread 2
main thread joined with sum thread 2
main thread about to join with sum thread 3
sum thread 5 returning
main thread joined with sum thread 3
main thread about to join with sum thread 4
main thread joined with sum thread 4
main thread about to join with sum thread 5
main thread joined with sum thread 5
the average over 100000 random numbers on (0,1) is 0.499644
```

This output is worth studying for a moment. Notice that the main thread starts 5 threads. The it completes the creation of 3 threads. It calls the create for the 4th thread, but before it prints the "created" message sum thread 1 starts running. Then the message saying that 4 was created prints. Then sum thread 2 and 3 start, and then the main thread creates thread 5.

This order of execution is not guaranteed. In fact, there are many leagl orderings of thread execution that are possible. All that matters is that the main thread not try and access the result from a thread until **after** it has joined with that thread.

This point is important All of the threads are independent and they can run in any order once they are created. They interleave their execution with each other and the main thread (or exencute in paralle if multiple cores are available). However a call to *pthread_join()* ensures that the calling thread will not proceed until the thread being joined with completes.

It is in this way that the main thread "knows" when each thread has completed computing its partial sum and successfully returned it.

You might wonder "What happens if a sum thread completes before the main thread calls *pthread_join()*?" In fact, that occurs in this sample execution. Thread 1 returns before the main thread calls *pthread_join()* on thread 1. The semantics of *pthread_join()* are that it will immediately unblock if the thread being joined with has already exited. Thus *pthread_join()*

- blocks is the thread being joined with hasn't yet exited
- unblocks immediately if the thread being joined with has already exited

Either way, the thread calling *pthread_join()* is guaranteed that the thread it is joining with has exited and, thus, any work that thread was doing must have been completed.

Synchronization

The functionality of *pthread_join()* illustrates an important operating systems concept: **synchronization**.

The term synchronization literally means "at the same time." However, in a computer science context it means

"the state of a concurrent program is consistent across two or more concurrent events."

In this example, the main thread "synchronizes" with each of the sum threads using the *pthread_join()* call. After each call to *pthread_join()* you, the programmer, know the state of two threads:

- the main thread, which has received the partial sum from the thread whose id is contained in the variable *thread_ids[t]*
- the sum thread whose id is contained in the variable *thread_ids[t]* and this state is that the thread has exited after sucessfully computing its partial sum.

Thus the main thread and one of the sum threads "synchronize" before the main thread tries to use the partial sum computed by the sum thread.

Synchronization is an important concept when concurrent and/or asynchronous events must be managed. We'll discuss syncronization a great deal through this class as it is a critical function of an operating system.

How much faster is it?

You can experiment with these last two example codes to see how much of an improvement threading and parallelism make in terms of the performance of the code. On my laptop. Running these codes with the Linux time command:

```
time ./avg-1thread 100000000
main thread forking sum thread
main thread running after sum thread created, about to call join
sum thread running
sum thread done, returning
main thread joined with sum thread
the average over 100000000 random numbers on (0,1) is 0.500023

real    0m1.620s
user    0m1.419s
sys     0m0.195s
```

and

```
time ./avg-manythread 100000000 10
main thread about to create 10 sum threads
main thread creating sum thread 1
main thread has created sum thread 1
main thread creating sum thread 2
sum thread 1 running, starting at 0 for 10000000
main thread has created sum thread 2
sum thread 2 running, starting at 10000000 for 10000000
main thread creating sum thread 3
main thread has created sum thread 3
main thread creating sum thread 4
main thread has created sum thread 4
main thread creating sum thread 5
main thread has created sum thread 5
main thread creating sum thread 6
sum thread 4 running, starting at 30000000 for 10000000
sum thread 3 running, starting at 20000000 for 10000000
main thread has created sum thread 6
sum thread 5 running, starting at 40000000 for 10000000
sum thread 6 running, starting at 50000000 for 10000000
main thread creating sum thread 7
main thread has created sum thread 7
sum thread 7 running, starting at 60000000 for 10000000
main thread creating sum thread 8
main thread has created sum thread 8
main thread creating sum thread 9
main thread has created sum thread 9
main thread creating sum thread 10
main thread has created sum thread 10
main thread about to join with sum thread 1
sum thread 8 running, starting at 70000000 for 10000000
sum thread 9 running, starting at 80000000 for 10000000
```

```
sum thread 10 running, starting at 90000000 for 10000000
sum thread 1 returning
main thread joined with sum thread 1
main thread about to join with sum thread 2
sum thread 5 returning
sum thread 3 returning
sum thread 4 returning
sum thread 6 returning
sum thread 8 returning
sum thread 7 returning
sum thread 2 returning
main thread joined with sum thread 2
main thread about to join with sum thread 3
main thread joined with sum thread 3
main thread about to join with sum thread 4
main thread joined with sum thread 4
main thread about to join with sum thread 5
main thread joined with sum thread 5
main thread about to join with sum thread 6
main thread joined with sum thread 6
main thread about to join with sum thread 7
main thread joined with sum thread 7
main thread about to join with sum thread 8
main thread joined with sum thread 8
main thread about to join with sum thread 9
sum thread 9 returning
main thread joined with sum thread 9
main thread about to join with sum thread 10
sum thread 10 returning
main thread joined with sum thread 10
the average over 100000000 random numbers on (0,1) is 0.500023

real    0m1.437s
user    0m1.501s
sys     0m0.207s
```

That's right -- using 10 threads only speeds it up with 0.2 seconds. Can you figure out why?

Experimenting with these examples

Many of the lectures in this class (like this one) include coding examples that are intended to illustrate some of the concepts that the lecture hopes to get across. In this case, the concepts are

- threads as a programming abstraction for dealing with concurrency and parallelism,
- basic thread creation and synchronization in pthreads
- concurrency in the thread runtime

If you don't understand these three concepts yet, that's okay -- there are a couple of ways to proceed.

First, you should go back an reread the lecture notes from beginning to end. The notes that I provide are intended to be read sequentially and not skimmed when you are trying to learn the concepts. This type of writing differs from other forms with which you might be familiar. It is, however, important to understand that it is by design. Each section of the notes leads to the next thus simply dropping into the middle or (more probably) looking at the bullet points and the examples won't likely yield a satisfactory explanation.

Secondly, the code exmaples are intended to serve as a vehicle for your own personal experimentation. That is, you can build, modify, and run these codes as a way of fiamilarizing yourself with some of the details that you might not have understood from the lecture notes.

For example, you might suspect that the reason the speed up using 10 threads over 1 thread in the previous examples is due to the *printf()* statements. Both of the last two codes include print statements to show how the threads interleave their execution (the third concept the lecture covers). You might suspect that the small difference in time is because the programs spend time printing messages and the message printing is much slower than the computation time.

To test this theory, you can make copies of these programs, comment out or remove the print statements, and rerun the timing experiments. Does it make the program faster? By how much?

Building the examples

In the notes, I include code fragments that won't necessarily execute properly if you cut and paste them directly from the test. That's because the working code is often too long to display properly in a classroom lecture format. Instead, I've provided commented, working versions of the code at

http://www.cs.ucsb.edu/~rich/class/cs170/notes/IntroThreads/example

In this directory you will find

- one or more C program files
- possibly one or more C head header files
- a makefile
- a README.md file

You can copy the files from this location.

I've also made the examples available on github. If you haven't used github before, it is a public code repository that promotes code sharing using the git source code control system. Git has many interesting features that are designed to allow distributed sets of developers to collaborate. One such feature allows you to "clone" a repository so that you can work with it on your own. To get a copy fo the examples from this class, log into a CSIL machine and type

```
git clone https://github.com/richwolski/cs170-lecture-examples.git
```

This command will create a subdirectory called cs170-lecture-examples. In it you'll see several subdirectories. For this lecture, the code is in the "IntroThreads" subdirectory.

To build the programs type

```
cd cs170-lecture-examples/IntroThreads
make
```

If all has gone well, you'll build the three programs this lecture discusses. You should look at the code in these progras as well. In the lecture notes, I've removed many of the error checks that good C and Linux programs should have.

You can make modifications to these programs. To rebuild them simply run the "make" command again and it will invoke the C compiler for you. The use of the make command is trivial for these specific examples (you can run gcc manually with little trouble). When we get to some of the assignments, however, it will be important to use make since the build environment is substantially more complex.