1 Instructions

You may work in **a group** on this project if you wish or you may work **alone**, although I **strongly encourage** you to work with a group. If you do work with a group, only **one** of you should **submit** the project to Blackboard. Make sure to put **both** of your names in the **source code** files and **name** the **tarball** following the instructions of **Section 5**. You will each **earn** the **same** number of points. **Section 5** describes **what** to submit and by **when**; **read it now**.

2 Lab Project Objectives

1. To be able to the concepts of **concurrent** and **parallel programming**.
2. To be able to explain what a GNU/Linux **process** is and how it differs from a **program**.
3. To be able to perform rudimentary GNU/Linux process management using the commands: **fg**, **jobs**, **ps**, and **kill**, and to be able to start processes in the background with **&**.
4. To be able to explain what a **thread** is.
5. To be able to use the POSIX **pthreads** library to call **pthread_create()** to start a thread and **pthread_exit()** to terminate a thread.
6. To be able to **pass parameters** to a thread and to **return values** from a thread.
7. To be able to write a **working makefile** to build a project.
8. To be able to write a basic **bash shell script** to test a project.

3 Concurrent Programming

Concurrent computing is a form of programming in which the program operates as a group of interacting computation processes or tasks [1]. The tasks themselves may be executed simultaneously or in an interleaved fashion. For example, suppose we have two concurrent tasks T1 and T2. These possibilities exist,

T1 and T2 are executed at the same time.
T1 is executed for a short while, then T2, then T1, then T2, etc.
T1 may start first, then T2 is started and runs during the time T1 is also running, and then T1 may finish before T2.

On a single-processor system, the separate processes are executed in an interleaved fashion with the microprocessor being switched among the processes at such a rate that computation appears to be happening simultaneously. There are various **scheduling algorithms** that may be used to distribute the processor's time, e.g., in a **round robin scheduling** algorithm, if there are four tasks T1, T2, T3, and T4, the sequence of execution is T1, T2, T3, T4, T1, T2, T3, T4, T1, ... In a **priority scheduling** algorithm, some tasks may have higher priority than other tasks. For example, suppose we have three tasks, T1, T2, and T3 and the order of priority from highest to lowest is T1, T2, T3. If T2 is executing, and T1 *needs* to execute, then T2 will be **suspended** and the processor will be allocated to T1 which will execute to completion, at which point T2 will be resumed.

In a multiprocessor or multicore system, multiple processes may be executed **in parallel** by assigning them to processors. For example, if there are two tasks T1 and T2 and two processors P1 and P2, then T1 may be assigned to execute on P1 and T2 on P2. *Both* processes are executing at the same time; this is **true parallelism** as opposed to the apparent parallelism of a uniprocessor system.

3.1 Processes

In GNU/Linux, a **process** is an instance of a program which is being executed. There is a many-to-one mapping between processes and programs. If two users are both running **vi** (a bad, bad idea) at the same time, then **one** program is running but there are **two** processes[1]. When you start a program, **bash** (or whatever shell you are running; which, by the way, is just another process) is **suspended** (basically, put to sleep) until the newly launched process terminates. In GNU/Linux, every process is assigned a unique **process identifier** (**pid**) by the kernel. You can see the running processes on a system by typing the **ps** command.

Exercise 1. Log in to **general.asu.edu** or some other GNU/Linux system. Type **ps** to display your running processes (note: type **ps -e** to see *all* the processes running on the sytem; this could be a lengthy list, you might try **ps -e | less**).

```
$ ps
  PID TTY          TIME CMD
22164 pts/0    00:00:00 bash
22256 pts/0    00:00:00 ps
```

[1] If you think of a program as being like a "class" than a process is like an "object".

From the **ps** output, you can see that I am running two processes with pid's 22164 (**bash**, or whatever shell you are running, is launched automatically when you log in) and 22256 (a process started by performing the **ps** command). The TTY column displays the "terminal" I am logged in to (don't worry about it) and the TIME column shows the amount of CPU time the process has consumed (this is not the same thing as "wall time").

Exercise 2. A process can be terminated by issuing the **kill** command. Using your favorite text editor, type this C program (which implements an infinite loop), compile it, and run it.

```
/* infinite.c */
int main()
{
    while (1)
        ;
    return 0;
}
```

```
$ gcc infinite.c -o infinite
$ ./infinite
```

When you run the program, it will appear to hang up. To terminate the process, type **Ctrl+C** which sends a SIGINT **signal** [2] to the process requesting it to terminate (SIGINT is the **int**errupt **sig**nal). A process can be written to ignore SIGINT, but the default behavior is for the process to terminate.

Exercise 3. When you run a process, e.g., by typing **./infinite**, the process runs in the **foreground**. Only one of your processes can be running in the foreground at a time. But, you can have multiple processes running in the **background**. To run a process in the background, you append **&** to the command. Try running **infinite** in the background,

```
$ ./infinite&
[1] 23897
$
```

The process is now running in the background with pid 23897 and this is background **job** 1 (the number in brackets; a **job** is a process running in the background; you can have mulitple running jobs). If you type **ps** at this point, you will see that **infinite** is running,

```
$ ps
  PID TTY          TIME CMD
22164 pts/0    00:00:00 bash
23897 pts/0    00:00:00 infinite
23912 pts/0    00:00:00 ps
$
```

So at this time, what is the foreground process? **Bash** of course. We can, however, bring **infinite** to the foreground with the **fg** command. The argument to **fg** is the job number, so we type,

```
$ fg 1
```

Now, **bash** is suspended and **infinite** is running in the foreground. What if you don't know the job number of the process that you want to bring to the foreground? The command **jobs** will display the job numbers of all of your background processes,

```
$ fg 1                                  Terminate infinite
Ctrl+C
$ ./infinite&                           Launch infinite in the background
[1] 24900
$ ./infinite&                           Launch another infinite in the background
[2] 24963
$ jobs -l
[1]-  24900 Running    ./infinite &
[2]+  24963 Running    ./infinite &
$
```

Let's terminate the first of these processes,

```
$ kill 24900
```

```
$ jobs -l                               That's an ell not a one
[2]+  24963 Running    ./infinite &
```

Let's bring **infinite** to the foreground and then send it back to the background,

```
$ fg 2
./infinite
Ctrl+Z                                  Ctrl+Z suspends a process

[2]+  Stopped  ./infinite &
$ bg 2                                  Sends infinite to the background
[2]+ ./infinite &
$ ps
  PID TTY          TIME CMD
22164 pts/0    00:00:00 bash
24963 pts/0    00:05:37 infinite
25501 pts/0    00:00:00 ps
```

3.2 Threads

A **thread** is an independent stream of instructions (independent of the other instructions of the program and other threads) that can be scheduled to run by the operating system [3]. You may think of a thread as a "function" that is called and executed independent of the main process (and other threads). The thread will run to completion and will then terminate. While the thread is running, the main process which spawned the thread may or may not continue executing, e.g., it could **block** on the thread, waiting for the thread to terminate before the main process continues executing.

In the GNU/Linux world, concurrent programming is often done using the **POSIX**[2] **Threads** (**pthreads**) programming library [4]. In this project we will explore simple threaded programming with pthreads. Some good online tutorials can be found at [3, 5, 6].

3.2.1 Creating a Thread

A thread is created with the **pthread_create()** function call,

```
pthread_create
      (
      pthread_t              *thread_id,
    const pthread_attr_t    *thread_attr,
    void *                 (*start_routine)(void*),
    void                    *thread_arg
    )
```

The first parameter **thread_id** is the address in memory of a variable of type **pthread_t**; this variable is an identification number that uniquely identifiers the thread, i.e., each thread is assigned a unique id number and **pthread_create()** returns the id number via this variable.

The second parameter is a pointer to a variable of type **const pthread_attr_t**. This parameter specifies "attributes" for the thread, some of which can be changed from their default values. Attributes include things such as: the size of the thread's run time stack, the address of the stack, the thread's scheduling policy (e.g., round-robin, first-in-first-out) and whether the thread is to run as **detached** or **joinable**[3].

---

[2]**POSIX** is an acronum for **P**ortable **O**perating **S**ystem **I**nterface. POSIX is an IEEE standard (IEEE 1003) which was written to define a standard for maintaining compatibility among operating systems, in particular Unix-like operating sytstems. The latest POSIX specification is IEEE 1003.1-2008 and can be found at http://pubs.opengroup.org/onlinepubs/9699919799 .

[3]A **joinable** thread is one that will run whilc the the starting process/thread "blocks" (waits) on it. When the thread terminates, the starting process/thread resumes execution. After starting a **detached** thread, the starting process/thread continues execution simultaneously with the thread.

The third parameter is the address of the function to be called to start the thead[4], i.e., it is a **function pointer**. The C syntax for declaring a function pointer is nasty,

```
return-data-type (*function-name)(function-params)
```

For example, consider this function,

```
int *AllocIntArray
    (
    int pSize  // Number of elements in the array to be allocated
    )
{
    int *array = (int *)malloc(pSize * sizeof(int));
    return array;
}
```

Suppose we want to pass **AllocIntArray()** as a paramter to another function (yes, you can do that in C, although you don't really pass the function, but rather you pass the address of the function),

```
// The parameter to SomeFunction() is a pointer (the address) to a function
which return an int * and has an int
// as the input parameter. pFunction is the address of AllocIntArray().
void SomeFunction
    (
    int* (*pFunction)(int)  // A pointer to a function
    )
{
    // Call pFunction and pass an int as the parameter. This calls
AllocIntArray(10).
    int *anArray = pFunction(10);
}
```

[4]You can think of this as the thread's **main()** function.

```
void SomeOtherFunction
    (
    )
{
    // Call SomeFunction() and pass the address of AllocIntArray() as the
parameter.
    SomeFunction(AllocIntArray);
}
```

You do not need to write **&AllocIntArray** to pass the address of **AllocIntArray()**. Functions are like arrays in this case, i.e., remember the name of an array evaluates to the address of the array, and the name of a function evaluates to the address of the function.

The fourth parameter **thread_arg** is of type void * and is a pointer to the input parameter to the thread's starting function, e.g.,

```
// The starting function for a thread always has an input parameter which is
a void * and always returns a
// void *.
void *ThreadStartFunction
    (
    void *pArg
    )
{
    // Retrieve the integer input paramter. First, we know that pArg is
really a pointer to an int (the
    // compiler doesn't; it thinks pArg is a void pointer). So, type cast
pArg to be a pointer to an int.
    // Then, we dereference arg to retrieve the integer.
    int *arg = static_cast<int *>(pArg);
    int n = *arg;
    ...
}

void StartThread
    (
    )
{
    pthread_t ThreadId;
    int n = 10;
    // Start a new thread with ThreadStartFunction() as the starting function
and pass n as the paramter.
    // Note, that the data type of the parameter must be type casted to be
a void pointer. Also note that
    // if you do not want to change the default thread attributes, then pass
0 (or NULL) as the second
    // parameter.
    pthread_create(&ThreadId,   0,   ThreadStartFunction,   static_cast<void
*>(&n));
    ...   StartThread()   continues   executing   simultaneously   with
ThreadStartFunction().
```

```
    // Note that ThreadId contains the id number for the thread that was
just started.
}
```

How do you pass more than one parameter to a thread's starting function? In C, as a pointer to a struct, and in C++, a pointer to a struct or a pointer to an object. For example,

```
// Define a structure for the thread's parameters. We assume the thread has
three parameters: an int, a double
// and a C string.
typedef struct {
    int     mInt;
    double  mDouble;
    char   *mString;
} ThreadParam;

// pArg is a defined as a void pointer, but it is really a pointer to a
ThreadParam.
void *ThreadStartFunction
    (
    void *pArg
    )
{
    // Type cast pArg to be the correct type.
    int *arg = static_cast<ThreadParam *>(pArg);
    // We can now access the parameters.
    cout << arg->mString;
    double z = arg->mInt + arg->mDouble;
    ...
}
```

```
void StartThread
     (
     )
{
     pthread_t ThreadId;
     ThreadParam ThreadParams = { 100, 3.13, "Wilma" };
     pthread_create(&ThreadId,   0,   ThreadStartFunction,   static_cast<void
*>(&ThreadParams));
     ...
}
```

If the thread does not have any parameters, then pass 0 (or NULL) for the fourth parameter.

3.2.2 Terminating a Thread

A thread may terminate in several ways, including: (1) by returning normally fromt the starting function; (2) by calling **pthread_exit()**; (3) when the main process terminates by calling **exit()**; or (4) if **main()** returns normally without calling **pthread_exit()**.

3.2.2.1 Method 1

A thread will terminate when the starting function returns, e.g.,

```
void *ThreadStartFunction
     (
     void *pArg
     )
{
     ... do some stuff
     return 0;  // Returns 0 (or NULL).
}
```

3.2.2.2 Method 2

A thread will terminate when it calls **pthread_exit()**. The function declaration for **pthread_exit()** is,

```
void pthread_exit
     (
     void *pValue
   )
```

The parameter to **pthread_exit()** is a void pointer to the thread's return value (if the thread does not return a value, pass 0 or NULL).

This example shows how a thread can return an **int**. Note that in order for the returned value to be "received" by someone, then the thread must be started as **joinable** (the normal thread behavior is to start as **detached**). So, who receives the return value? The process.thread which **joins** with the thread returning the value. Study **StartThread()** to see how to start a thread as joinable, and how to join with the thread and retrieve the return value.

```
void *ThreadStartFunction(void *pArg)
{
     ... do some stuff
     // Return the integer value 42.
     int *returnValue = new int;
     *returnValue = 42;
```

```
    pthread_exit(static_cast<void *>(returnValue));
}

void StartThread
    (
    )
{
    pthread_t ThreadId;

    // Threads are started in detached mode by default. If you want to start
a thread to be joinable then
    // create a variable of the pthread_attr_t type and configure it this
way.
    pthread_attr_t ThreadAttributes;
    pthread_attr_init(&ThreadAttributes);
    pthread_attr_setdetachstate(&ThreadAttributes,
PTHREAD_CREATE_JOINABLE);

    ThreadParam ThreadParams = { 100, 3.13, "Wilma" };

    // Notice that the address of ThreadAttributes is passed to the create
function.
    pthread_create(&ThreadId,    &ThreadAttributes,    ThreadStartFunction,
static_cast<void *>(&ThreadParams));

    // The ThreadAttributes has to be destroyed.
    pthread_attr_destroy(&ThreadAttributes);

    // Now this process will block (wait) until the thread terminates by
calling pthread_exit(). We block
    // by calling pthread_join() and passing the thread id as the first
parameter, and a void ** as the
    // second parameter, i.e., vReturnValue is a pointer to a void pointer.
    void *vReturnValue;
    pthread_join(ThreadId, &vReturnValue);

    // After the thread terminates, execution will continue. We need to
retrieve the integer that was
    // returned. First, we type cast the vReturnValue pointer (which is a
void pointer) to be an int *.
    // Then we dereference iReturnValue to retrieve the integer.
    int *iReturnValue = static_cast<int *>(vReturnValue);
    int returnValue = *iReturnValue;

    // Note that returnValue in ThreadStartFunction() was dynamically
allocated with new. Therefore, we
    // have to deallocate (delete) that block of memory.
    delete iReturnValue;
    ...
}
```

It is **very important** to understand that the thread cannot pass the address of a local variable to **pthread_exit()**, e.g.,

```
void *ThreadStartFunction(void *pArg)
{
    ... do some stuff
    // Return the integer value 42.
    int returnValue = 42;
    pthread_exit(static_cast<void *>(&returnValue));
}
```

Why is this bad? Well, *returnValue* is an **int** and it is a local variable, so that means it is allocated on the run time stack. And what happens to local variables when a function returns? They are deallocated. So, in this case we would return the address of a variable which is not going to exist and in the calling function if we attempt to access that memory location to retrieve the value, we will: (1) perhaps get a segmentation fault; or (2) retrieve some "weird" value which is not the one we expect to get; or (3) get the correct value, but just by luck. The last of these is the least fortuitous because what you will notice is that some times the program will work correctly (you will get the value you expect), and other times it won't (you will get some weird value). These sorts of bugs can be maddeningly difficult to find. If you're lucky, you will get a simple seg fault; *those* are easy to find.

3.2.2.3 Method 3

If the main process terminates by calling **exit()** then all threads started by that process will also be terminated, e.g.,

```
void *ThreadStartFunction(void *pArg)
{
    ... do some stuff
}

void StartThread
    (
    )
{
    ... code here to start a thread by calling ThreadStartFunction()
    ...
    if (someErrorCondition == true) {
        cerr << "Fatal error! Terminating." << endl;
        exit(-1);  // All threads will be terminated
    }
    ...
}
```

3.2.2.4 Method 4

If the main function **main()** terminates first by returning, then all threads started by the process will be terminated.

```
void *ThreadStartFunction(void *pArg)
{
    ... do some stuff
```

```
}

void StartThread
     (
     )
{
     ... code here to start a thread as detached by calling
ThreadStartFunction()
}

int main
     (
     )
{
     ...
     StartThread();
     ...
     return 0;  // If main() returns here before the thread terminates, the
thread will be teminated
}
```

4 Lab Programming Project

Download the **Extra Credit Project tarball** from the course website and extract it to a working directory. You will find several source code files, listed and briefly described below. The *italicized file names* contain ??? symbols in the places where you must complete the code; the remaining files are complete.

Makefile A template for the project make file. This must be completed and submitted for grading.
Amicable.hpp Contains function declarations for global functions in Amicable.cpp.
Amicable.cpp Contains function definitions for the amicable numbers test.
Main.hpp Contains function declarations for global functions in Main.cpp.
Main.cpp Contains the *main*() function and other miscellaneous functions definitions.
Prime.hpp Contains function declarations for global functions in Prime.cpp.
Prime.cpp Contains function definitions for the prime numbers test.
Test.sh A template for the Bash shell script I demonstrated in class. This must be completed and submitted for grading.
Thread.hpp Contains function declarations for global functions in Thread.cpp.
Thread.cpp Contains a function to start a thread.
Types.h Contains useful typedefs.

4.1 Keith Numbers

A **Keith number** [7] is a number K, $K \geq 10$, such that K appears in a specific **sequence**. Let n be the number of digits in K, e.g., if $K = 197$, then $n = 3$. Let the digits of K be d_n, d_{n-1}, ..., d_1 with d_n being the most significant digit of K and d_1 being the least significant digit. The sequence starts with the digits of K,

starting sequence = $d_n\ d_{n-1}\ d_1$

and the next number in the sequence is always the sum of the previous n numbers (this sequence is similar to the Fibonacci sequence where $n = 2$ and the first two numbers are 0 and 1). K is a Keith number if K appears in the sequence. For example, let $K = 197$ and $n = 3$. We have,

starting sequence = 1 9 7

The next number in the sequence is 1 + 9 + 7 = 17, so now the sequence is,

sequence = 1 9 7 17

The next number in the sequence is 9 + 7 + 17 = 33, so now the sequence is,

sequence = 1 9 7 17 33

Continuing,

sequence = 1 9 7 17 33 57 107 **197**

Since *K* = 197 appears in the sequence, 197 is a Keith number. Keith numbers are very rare with approximately 100 being known.

4.2 Lab Requirements

The program, as is, will test all the numbers from 2 to some upper limit for primality and all the pairs of numbers from 2 to some upper limit to see if they are amicable. You are to augment this program by adding two new source code files,

Keith.hpp Contains function declarations for global functions in Keith.cpp.
Keith.cpp Contains function definitions for the Keith numbers test.

Keith.cpp shall provide three global functions, declared in **Keith.hpp** as,

```
extern void FindKeiths
     (
     ulong pLimit
     );

extern void *FindKeithsThread
     (
     void *pState
     );

extern bool IsKeith
     (
     ulong pNum
     );
```

FindKeiths() shall find all Keith numbers from 10 up to and including **pLimit**. When a Keith number is found, we shall do nothing with the Keith number (to see what I mean by "nothing" look at **FindPrime()** in **Prime.cpp**). **FindKeithsThread()** is the starting function for a thread which finds all Keith numbers from 10 up to and including **pLimit**. It shall do this by calling **FindKeiths()**. *Hint: See FindPrimes() in Prime.cpp*. **IsKeith()** shall return true if

pNum is a Keith number and false if it is not. We will test that your code in **Keith.cpp** successfully finds Keith numbers by calling this function against a test driver routine.

You shall complete the lab project by complete the missing code (indicated by ??? in the source code files) so the program makes successfully using your **Makefile**.

The **Test.sh** shell script shall be used to test the program and the output from the shell script shall be similar to this,

```
~/cse220/src/thread$ ./Test.sh
Testing parallel version...
FindPrimesThread() Begin
FindPrimes() Begin
FindKeithsThread() Begin
FindKeiths() Begin
FindAmicableThread() Begin
FindAmicable() Begin
FindPrimes() End
FindPrimesThread() End
FindAmicable() End
FindAmicableThread() End
FindKeiths() End
FindKeithsThread() End
16 seconds.
Testing serial version...
FindPrimes() Begin
FindPrimes() End
FindAmicable() Begin
FindAmicable() End
FindKeiths() Begin
FindKeiths() End
34 seconds.
... repeats the test 9 more times
~/cse220/src/thread$
```

5 What to Submit for Grading and by When

When your programs are complete, type the following commands to "clean" the source code directories and create a bzipped tarball named **cse220-ec-*lastname*.tar.bz2** (note, the thirteen files you see below should be the only files in the tarball you submit for grading).

```
~ $ cd cse220-ec/thread
~/cse220-p05/thread $ make clean
~/cse220-p05/thread $ ls
Amicable.cpp  Keith.cpp  Main.cpp  Makefile   Prime.hpp   Thread.hpp
Amicable.hpp  Keith.hpp  Main.hpp  Prime.cpp  Thread.cpp  Types.hpp
~/cse220-p05/thread $ cd ..
~/cse220-p05 $ tar cvjf cse220-ec-lastname.tar.bz2 cse220-p05
```

Where *lastname* is your surname (or your first name if you do not have a surname). If you work with a group, be sure to put all of your names in the source code files (in the AUTHORS section of the header comment blocks) and name your tarball **cse220-ec-*last name1-lastname2-etc*.tar.bz2**. Submit this tarball to Blackboard using the lab project submission link before the deadline. The **deadline is 11:59pm Friday 9 May 2014 This is a hard deadline; no projects will be accepted after the deadline for any reason. Furthermore, since this is a bonus lab project, no bonus points will be assigned for an early submission. Consult the online syllabus for the late and academic integrity policies.**

6 References

1. http://en.wikipedia.org/wiki/Concurrent_computing
2. http://en.wikipedia.org/wiki/Unix_signal
3. https://computing.llnl.gov/tutorials/pthreads
4. http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html

5. http://www.multicoreinfo.com/research/misc/Pthread-Tutorial-Peter.pdf
6. http://www.multicoreinfo.com/research/misc/MultiThreaded-Programming-With-POSIX.pdf
7. http://en.wikipedia.org/wiki/Keith_number