Lab 12: Multithreading

For this lab, create a new directory named `lab12` under your `cs449` directory and create your program there:

```
mkdir lab12
cd lab12
```

For this lab, you can get the starter file (`bbSort.c`) to your directory using the following command:

```
cp /afs/cs.pitt.edu/usr0/tkosiyat/public/cs0449/bbSort.c .
```

For those who works on Ubuntu on your own computer, the starter file is also available on the CourseWeb.

Introduction to Multithreading

First, let's look at the program `bbSort.c` where its `main()` function is shown below:

```
int main(void)
{
        int *array1;
        int *array2;
        int i;
        struct timeval startTime, endTime, elapsedTime;

        // Seed the random number generator

        srand(time(0));

        // Generate two integer array filled with random numbers

        array1 = genRandomIntArray(NUMEL,10000);
        array2 = genRandomIntArray(NUMEL,10000);

        // Sort two arrays

        gettimeofday(&startTime, NULL);
        bubbleSort(array1, NUMEL);
        bubbleSort(array2, NUMEL);
        gettimeofday(&endTime, NULL);

        // Print the first 100 elements of each array

        printf("array1: ");
        printArray(array1, 100);
        printf("array2: ");
        printArray(array2, 100);

        // Show how long does it takes to sort two arrays
        elapsedTime = getElapsedTime(startTime, endTime);

        printf("Sorting Time: %li.%.6li seconds\n", elapsedTime.tv_sec, elapsedTime.tv_usec);

        return 0;
}
```

From top to bottom, the above program perform the following:

1. Generates two integer arrays and fills them with random numbers,
2. Sorts `array1` follows by `array2`,
3. Display the first 100 elements of both arrays, and
4. Displays how long does it takes to sort two arrays.

To compile and run the above program, use the following command:

```
gcc -m32 -o bbSort bbSort.c
./bbSort
```

The last line of output should show `Sorting Time:`. Note that the sorting algorithm that we use is called bubble sort where its algorithm is Big-Oh of n^2. As a result, the sorting time grows exponentially with the size of array. In the program `bbSort.c`, there is a `define` statement as shown below:

```
#define NUMEL 10000
```

The above statement set the number of elements of both arrays. First adjust this number such that the sorting time is not too fast or too slow (somewhere between 2.0 to 5.0 seconds). Do not forget that once you modify the above line, you have to recompile it before you run the program.

Note that the above program will not sort the second array until the first array is sorted. Ideally, the above program can run faster if it can sort both arrays at the same time. This can be done in two different scenarios:

- Sort each array using two processes. In other words, use two programs where each program sorts its own array.
- Use the same process but sort each array using two threads.

The first option results in two process address spaces which is fine as long as two processes does not need to share the same set of data. The second option allows both threads to share the same set of data which is very useful in many situation. So, we will focus on the second option.

Lab 12: Multithreading

Creating New Threads

By simply call a function such as `bubbleSort()` in `bbSort.c` will not result in a new thread. When you run a C program, it is a single thread program. That is why when you call a function, the program cannot do anything other than wait until the function is done. However, if a function is called and it was created as a new thread, the `main()` thread will not have to wait. It can continue doing something useful.

To call a function as a new thread, you need help from `pthread` library. So, first, you need to include the `pthread` library as shown below:

```
#include <pthread.h>
```

To create a thread you need to perform three simple steps:

1. Declare a variable of type `pthread_t`. For example

   ```
   pthread_t thread;
   ```

2. Call `pthread_create()` function to create a new thread. The signature of this function is as follows:

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

   The first argument is the address of the variable of type `pthread_t` created in step 1. The second argument simply set to `NULL` for this lab. The third argument is the function to be called as a new thread. Note that it is a function pointer. Thus, the function that you want to call as a new thread must take one argument of type `void` pointer and return a value of type `void` pointer. The last argument is the argument in the form of `void` pointer that you want to pass to the `start_routine()` function.

3. Call `pthread_join()` function to wait for the new thread to finish. This function takes two arguments. The first argument is a variable of type `pthread_t` created in step 1 and used in step 2. For the second argument, simply set it to `NULL` for now.

Note that calling `pthread_join()` is optional as long as your `main()` thread has something else to do. If your `main()` thread does not have anything else to to, it will finish its own `main()` function which results in program being terminated. If the `main()` thread is terminated, any other threads created by the `main()` thread will be terminated as well.

So, ideally, the structure of your program should look somewhat like the following:

```
:
#include <pthread.h>
:
void *foo(void *) {...}
:
int main(void)
```

```
{
    pthread_t thread;
    aType args;
    :
    if(pthread_create(&thread, NULL, foo, &args))
    {
        printf("Error creating thread\n");
        return -1;
    }
    :
    if(pthread_join(thread, NULL))
    {
        printf("Error joining thread\n");
        return -1;
    }
    :
    return 0;
}
```

Note that the type of variable `args` is simply `aType`. It really depends on how many argument your function needs. Recall that the function that will be run as a new thread can only receive one argument of type `void` pointer. If you original function needs three arguments, you have to pack them into one and pass them to the function `pthread_create()` as a pointer. There are various way to do this and one way is to use structure. For example, suppose the original function named `foo()` has the following signature:

```
int foo(int x, int *y, float z)
```

and you need to convert it into a function that can be called by `pthread_create()` which must have the following signature:

```
void *foo(void *args)
```

simply create a structure as follows:

```
struct fooArgs
{
    int x;
    int *y;
    float z;
};
```

and modify the function `foo()` as follows:

```
void *foo(void *args)
{
    struct fooArgs *temp = (struct fooArgs *) args;
    int x = fooArgs->x;
    int *y = fooArgs->y;
    float z = fooArgs->z;
    :
}
```

Now, you can use `pthread_create()` function to call as follows:

```
    struct fooArgs someArgs;
    pthread_t thread;
    :
    someArgs.x = 20;
    someArgs.y = anArray;
    someArgs.z = 12.345;
    :
    pthread_create(&thread, NULL, foo, &someArgs);
    :
```

How to Compile?

To compile a multithread program, you need to add `-lpthread` flag. For example, suppose you have a multithread program named `bbSortThreads.c`, to compile this program, use the following command:

```
gcc -m32 -o bbSortThreads bbSortThreads.c -lpthread
```

What to do?

Your goal is to create a new program named `bbSortThreads.c` and give it the ability to sort two arrays at the same time using different threads. For simplicity, start from `bbSort.c` using the following command:

```
cp bbSort.c bbSortThreads.c
```

and start by modifying `bbSortThreads.c`. Use the previous section as a guideline to modify the `main()` function. Note that you must comment out the two lines that calls `bubbleSort()` which are located between two calls to the function `gettimeofday()`. Then insert four new statements as shown below (variable names may be different):

```
        :
        gettimeofday(&startTime, NULL);
        //bubbleSort(array1, NUMEL);
        //bubbleSort(array2, NUMEL);
        pthread_create(&thread1, NULL, bubbleSort, &args1);
        pthread_create(&thread2, NULL, bubbleSort, &args2);
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
        gettimeofday(&endTime, NULL);
        :
```

If all went well, the time to sort two arrays should be cut in half (approximately).

What to Hand In

First, let us go back up to our cs449 directory:

```
cd ..
```

Now, let us first make the archive. Type your username for the USERNAME part of the filename:

```
tar cvf USERNAME_lab12.tar lab12
```

And then we can compress it:

```
gzip USERNAME_lab12.tar
```

Which will produce a `USERNAME_lab12.tar.gz` file.

If you work on `cs449.cs.pitt.edu` (`thoth`) you can skip to the next section. **If you use your own machine, you need to transfer the file to `cs449.cs.pitt.edu` first**. This can simply be done by a command line. For example, assume that your username is `abc123` and you are in the same directory as the file `abc123_lab12.tar.gz`. To transfer the file to `cs449.cs.pitt.edu` use the following command:

```
scp abc123_lab12.tar.gz abc123@cs449.cs.pitt.edu:.
```

The above command will copy the file to your home directory in `cs449.cs.pitt.edu`. If you want to copy it to your `private` directory, use the following command:

```
scp abc123_lab12.tar.gz abc123@cs449.cs.pitt.edu:./private/.
```

Copy File to Submission Directory

We will then submit that file to the submission directory:

```
cp USERNAME_lab12.tar.gz /afs/cs.pitt.edu/public/incoming/CS0449/tkosiyat/sec1
```

Once a file is copied into that directory, you cannot change it, rename it, or delete it. If you make a mistake, resubmit a new file with slightly different name, being sure to include your username. For example `USERNAME_lab12_2.tar.gz`. **Check the due date of this lab in our CourseWeb under Labs/Recitations**.