Programming Assignment 1

Program the following two assignments in C. You may use the C programming environment in computer science department UNIX/Linux server, which is "xlogin.cs.ecu.edu". You could log in the server using:

$ ssh username@xlogin.cs.ecu.edu If you are off ECU campus, you need connect ECU via VPN, check the link on how to

connect via VPN: http://www.ecu.edu/cs-itcs/connect/studentVPN.cfm). Link of a tutorial for UNIX and C: http://heather.cs.ucdavis.edu/~matloff/unix.html

Part 1: UNIX Processes

Use the thin clients in Austin Room No. 208 or the PCs in Room No. 207 or your own computer to get onto ***xlogin.cs.ecu.edu*** and work on your programming assignments. The ***xlogin*** server runs SUSE LINUX. Create a subdirectory called ***cs4110*** in your home directory. Create a subdirectory called ***assign1*** in your cs4110 directory. Use that subdirectory to store all the files concerning this assignment and nothing else. You need to follow these general guidelines for all your future assignments as well. Name the two source files ***worker.c*** and ***coordinator.c***. The code for worker process should be compiled separately and its executable be called worker. It should be possible to execute the worker program independently of the coordinator. The executable for the coordinator process should be called coordinator. If you are not using **makefile**, please include the name of the compiler you are using and any special options needed as comments to your source code.

The goal of this homework is to get you familiar with concurrent processing in the UNIX operating system. You will write a program that uses multiple processes to compute the product of two matrices A and B. Let A be of order $m \times p$ and B be of order $p \times n$. Then, the product matrix C will be of order $m \times n$. Each element of the product matrix C is given by

$$C_{ij} = \sum_{k=1}^{p} a_{ik} \times b_{kj} \tag{1}$$

There are two types of processes for this homework:

- A set of "worker" processes: Each worker process reads two integer vectors from its `argv`, computes the result using expression 1 and returns the result using the `exit` system call. So, a worker process is created for every element of the product matrix C. Note that the worker processes can work parallely. In a real distributed system with multiple processors, this will result in speed up of the computation.
- A "coordinator" process: It is responsible for creating the "worker" processes, and coordinating the computation. Note that all the computation is done by the "worker" processes. The input vectors to the "worker" processes are provided in the command line (`argv`).

The two matrices to be multiplied would be kept in separate files whose names are to be provided as command line parameters. Your program should place the resulting matrix in a file whose name is specified as the last command line parameter.Thus the program will be executed by giving the command

coordinator A.mat B.mat C.mat

The format of each matrix file with s rows and t columns is as follows:

$$\begin{array}{ccccc} \text{s} & \text{t} & & & \\ a_{11} & a_{12} & a_{13} & \cdots & a_{1t} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2t} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3t} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{s1} & a_{s2} & a_{s3} & \cdots & a_{st} \end{array}$$

Since the results are passed around by *exit* keep the numbers small. Each worker process should print its *process id*, the two input vectors and the final result computed by it. Each time the coordinator gets a result from a worker, it should print the pid of the worker and the result received on the terminal.

Achieve maximum parallelism in your implementation. Although your program should be general, do not test it on large matrices. At any time, do not have more than fifty processes in the system concurrently.

System Calls The primary system calls that you need to be familiar with for doing this assignment are the following: `fork`, `execvp`, `exit`, `wait`, `getpid` and `perror`. You can read about these system calls from your favorite book on UNIX Operating Systems. Couple of them are kept in the library as reserve material. You can also learn about them by using the manual pages in the system itself.

Part 2: POSIX threads

Use the thin clients in Austin Room No. 208 or the PCs in Room No. 207 or your own computer to get onto ***xlogin.cs.ecu.edu*** and work on your programming assignments. The ***xlogin*** server runs SUSE LINUX. Create a subdirectory called ***cs4110*** in your home directory. Create a subdirectory called ***assign2*** in your cs4110 directory. Use that subdirectory to store all the files concerning this assignment and nothing else. You need to follow these general guidelines for all your future assignments as well. Name your source file as ***prefix.c*** If you are not using **makefile**, please include the name of the compiler you are using and any special options needed as comments to your source code.

Objective: The goal of this homework is to get you familiar with multithreaded programming using **POSIX threads**. You will write a multithreaded program to compute in parallel the sums of all prefixes of an array A of n integers. This technique is called **parallel prefix computation** and is explained in detail in the next section. The beauty of this technique is that it can be used for any associative binary operation (and not just for addition) and consequently it is useful in many different applications. While implementing this technique, you will need to synchronize the threads using **barrier synchronization** mechanism.

Parallel Prefix Computation: Assume that the array `A` is indexed from 1 to n and consists of n integers. The goal is create another array, called `Sum` of the same size and such that `Sum[i]` contains the sum of the first i elements of `A`. Of course, this can be easily achieved by the following simple program fragment.

```
Sum[1] = A[1];
for (i=2; i<=n; i++)
    Sum[i] = Sum[i-1] + A[i];
```

This will be of complexity $O(n)$ and we cannot do any better by means of a sequential program. However, if we can work parallely we can do it in $O(\log n)$ time using the technique of parallel prefix computation.

First, set all the elements `Sum[i]` to be the same as `A[i]` parallely. Then, in parallel add `Sum[i-1]` to `Sum[i]`, for all $i \geq 2$. In particular, add elements that are distance 1 away. Now double the distance and add `Sum[i-2]` to `Sum[i]`, in this case for all $i \geq 3$. If you continue to double the distance, then after $\lceil \log_2 n \rceil$ rounds you will have computed all partial sums (Can you prove it?)

Example: An illustration of parallel prefix computation on an array of size 6 is shown below. The first row is the initial data in the array.

Array Index	1	2	3	4	5	6
Initial Array	5	6	9	4	7	3
Sum after distance 1	5	11	15	13	11	10
Sum after distance 2	5	11	20	24	26	23
Sum after distance 4	5	11	20	24	31	34

Note that the Sum array now has the desired values in it. Note that in each step, each position of the array can be independently computed. Hence, if we use one thread for each position of the array, each step can be computed in unit time. As the number of steps is [log2 n], we have obtained an order of magnitude speed up (O(log n) as compared to the O(n) complexity for the sequential algorithm).

Needless to say, care must be taken to ensure that the threads are synchronized properly. Basically, each thread can move onto the next step only after all the threads have completed the previous step. This type of synchronization is called **barrier synchronization**. The barrier synchronization comes up in very many practical applications that **Pthreads** provide direct calls for that. I will describe those calls in this handout. However, we can also implement it ourselves, using *mutex* lock and condition variables of **Pthreads**.

The input data should be taken from a file. The file will simply be a sequence of integers. Thus the program should be executed by

prefix data

Your output should be the inital state of the array `Sum` and after each round Make sure to achieve parallelism in your implementation. Note that as you are using only threads (not full blown processes), all the threads will have the arrays available to work with (you need to make them global variables). Also make sure that all your threads run the same function.

Thread Primer: Pthreads is just a short form for POSIX threads. POSIX stands for Portable Operating System Interface. Pthreads library is now widely available on various flavours of the UNIX Operating system.

Using Pthreads with a C Program involves four steps. First, include the standard header for the Pthreads library:

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

Second, declare variables for one thread attributes descriptor and one or more thread identifiers, as in

```
pthread_attr_t tattr;  /* thread attributes */
pthread_t tid;         /* thread identifier */
```

Third, initialize the attributes by executing

```
pthread_attr_init (&tattr);
pthread_attr_setscope (&tattr, PTHREAD_SCOPE_SYSTEM);
```

Finally, create the threads, as described below.

The initial attributes of a thread are set before the thread is created; many can later be altered by means of thread-management functions. Thread attributes include the size of the thread's stack, its scheduling priority, and its scheduling scope (local or global). The default attribute values are often sufficient, with the exception of scheduling scope. A programmer usually wants a thread to compete with all other threads for processor time and not just with other threads created within the same process. The second call above accomplishes this.

A new thread is created by calling the `pthread_create` function, as in

```
pthread_create (&tid, &tattr, start_func, arg);
```

The first argument is the address of a thread identifier that is filled in if creation is successful. The second is the address of a previously initialized thread attributes descriptor. The new thread begins execution by calling `start_func` with a single argument, `arg`. If thread creation is successufl, `pthread_create` returns a zero; a non-zero return value indicates an error.

A thread terminates its own execution by calling

```
pthread_exit (value);
```

The `value` is a single return value (or NULL). The `exit` routine is called implicitly if the thread returns from the function that it started executing. However, it is a good programming practice to call it explicitly.

A parent thread can wait for a child to terminate by executing

```
pthread_join(tid, value_ptr);
```

where `tid` is a child's descriptor and `value_ptr` is the address of a location for the return value. The return value is filled in when the child calls `exit`.

Pthreads can participate in a barrier to synchronize to some point in time. Before a barrier can be called, a pthread barrier object must be declared as follows. Do this declaration globally (at the top).

```
pthread_barrier_t barrier;
```

Then, it needs to be initialized. Do the initialization in the main program as follows

```
pthread_barrier_init (&barrier, NULL, count);
```

The count variable defines the number of threads that must join the barrier to reach completion and unblock all threads waiting at the barrier.

The actual barrier call will look as follows.

```
pthread_barrier_wait(&barrier);
```

This function call would be inside the thread code at the barrier point. Once `count` number of threads have called the function, then the barrier condition is met and all the threads are unblocked and allowed to proceed.

Finally, note that you need to give `-lpthread` option at the time of compiling so that the pthread run time library could be linked to your object program.