Royal Commission for Jubail and Yanbu
Yanbu University College
Department of Computer Science and Engineering

CS401- Computer Architecture and Parallel Computing
Project 1: Threads

This is an individual assignment

Due date: Check the course website on e-learning.

NOTE:

1) For all other questions, your only reference is your textbook.
2) **Any trace of copying answers from other students in current or previous semesters will cause both parties (copier and the one copied from) to receive 0 marks in the whole Assignment/Project, irrespective if one or more answers to the questions have been copied.**

3) Submission should be in electronic format and saved as in the following format:
asg2_prj1_YourName.txt , source code files in .c format , unless otherwise stated.
4) Due date: Check the assignment's due date/time on course website.

To be clear on points 1 and 2 above, here are two terms that will be of help to you to sort any misconceptions for that regard.

Plagiarism is stealing or passing off the ideas or words of another as one's own – using material without crediting the source. This is prohibited behaviour and will not be tolerated. Take the time to properly cite material written by someone else -- include references, put verbatim quotes in quotation marks, and do not paraphrase excessively. If you have questions about this, ask me.

Citing means to identify where in the body of the report you used the information that is provided in the reference. For Internet web sites, include both the URL and some facts about the web site itself. If the facts come from a paper identified by the web site, identify the author of the paper, if possible; the company that sponsored the paper, if possible; or the identity of the site from which you obtained the information. This kind of reference enables the reader to better judge the value of the reference and enable the reader to locate other information produced by the same person, company, or sponsor.

Threads:

A thread is a semi-process, that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).

In goal of this assignment is to become familiar with basic thread life-cycle operations. Particularly:

- Creating Threads
- Terminating Thread Execution
- Passing Arguments To Threads
- Thread Identifiers
- Joining Threads
- Detaching / Attaching Threads

pthreads: the POSIX threading library

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or *pthreads*. Most hardware vendors now offer pthreads in addition to their proprietary API's.

pthreads are defined as a set of C language programming types and procedure calls. Vendors usually provide a pthreads implementation in the form of a header/include file and a library, which you link with your program

***pthread* API:**

The subroutines in the pthread library can be grouped into three major classes:

- Thread management: The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (join-able, scheduling etc.)

- Mutexes: The second class of functions deal with a coarse type of synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

- Condition variables: The third class of functions deal with a finer type of synchronization – based on programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Naming conventions:

All identifiers in the threads library begin with *pthread_*

pthread_	Threads themselves and miscellaneous subroutines
pthread_t	Thread objects
pthread_attr	Thread attributes objects
pthread_mutex	Mutexes
pthread_mutexattr	Mutex attributes objects.
pthread_cond	Condition variables
pthread_condattr	Condition attributes objects
pthread_key	Thread-specific data keys

Thread Management Functions:

Function:
```
int pthread_create (
    pthread_t * threadhandle, /* Thread handle returned by reference */
   pthread_attr_t *attribute, /* thread attributes, use NULL for defaults*/
   void *(*start_routine)(void *), /* Starting function which the thread executes */
   void *arg /* An extra argument passed as a pointer */
);
```

Description: Request the PThread library for creation of a new thread. The return value is 0 on success. The return value is negative on failure. The pthread_t is an abstract data type that is used as a handle to reference the thread.

Function
```
void pthread_exit
(
void *retval /* return value passed as a pointer */
);
```

Description: This Function is used by a thread to terminate. The return value is passed as a pointer. This pointer value can be anything so long as it does not exceed the size of (void *). Be careful, this is system dependent. You may wish to return an address of a structure, if the returned data is very large.

Function:
```
int pthread_join
(
pthread_t threadhandle, /* Pass threadhandle */
void **returnvalue /* Return value is returned by ref. */
);
```

Description: Return 0 on success, a
nd negative on failure. The returned value is a pointer
returned by reference. If you do not care about the return value, you can pass NULL for the second argument.

Creating threads:

Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.

pthread_create creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code .

Note: To compile any of the pthread library programs, you must explicitly link to the library, by adding:
***(-lpthread)* to the compilation command. For example, if we are to compile a mythread.c program, here is an example command line to compile the program:**

```
$ gcc -o mythead mythread.c -lpthread
```

Thread-termination:

There are several ways in which a pthread may be terminated:

- The thread returns from its starting routine. By default, the pthreads library will reclaim any system resources used by the thread. This is similar to a **process** terminating when it reaches the end of main.
- The thread makes a call to the ***pthread_exit*** subroutine, after it has finished its work.
- The thread is cancelled by another thread via the ***pthread_cancel*** routine.
- The entire process is terminated due to a call to either the exec or exit subroutines.
- If main() finishes first, without calling `pthread_exit` explicitly itself

The `pthread_exit()` routine allows the programmer to specify an optional termination *status* parameter. This optional parameter is typically returned to threads "joining" the terminated thread.

In subroutines that execute to completion normally, you can often dispense with calling `pthread_exit()` - unless, of course, you want to pass the optional status code back.

Cleanup: the `pthread_exit()` routine does not close files; any files opened inside the thread will remain open after the thread is terminated.

Discussion on calling `pthread_exit()` from main():

- There is a definite problem if main() finishes before the threads it spawned if you don't call `pthread_exit()` explicitly. All of the threads it created will terminate because main() is done and no longer exists to support the threads.
- By having main() explicitly call `pthread_exit()` as the last thing it does, main() will block and be kept alive to support the threads it created until they are done.

//Example 7.1

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

int glob_data = 5 ;

void *th_func(void *p)
{

 printf ("Thread here. Global data was %d.\n", glob_data) ;
```

```
 glob_data = 15 ;

 printf ("Thread Again. Global data was now %d.\n", glob_data) ;
}

void main ( ) {

 pthread_t t1 ;

 pthread_create (&t1, NULL, th_func, NULL) ;

 printf ("Main here. Global data = %d\n", glob_data) ;

 glob_data = 10 ;

 pthread_join (t1, NULL) ;

 printf ("End of program. Global data = %d\n", glob_data) ;
}
```

Passing arguments to threads:

The pthread_create() routine provides the programmer with a facility to pass one argument to the thread start routine. All arguments must be passed by reference and cast to (void *). In case where multiple arguments must be passed, this limitation can be easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the pthread_create() routine.

//Example 7.2

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

int sum; /*This data is shared by the thread(s) */

void *runner(void *param); /* the thread */

void main(int argc, char *argv[])
{
 pthread_t tid; /* the thread identifier */
 pthread_attr_t attr; /* set of thread attributes */

 if(argc != 2){
  fprintf(stderr,"usage: a.out <integer value>\n");
  exit();
 }

 if(atoi(argv[1]) < 0) {
  fprintf(stderr, "%d must be >= 0 \n", atoi(argv[1]));
  exit();
 }

 /* get the default attributes */
 pthread_attr_init(&attr);

 /*create the thread */
 pthread_create(&tid,&attr,runner,argv[1]);
```

```
 /* Now wait for the thread to exit */
 pthread_join(tid,NULL);

 printf("sum = %d\n",sum);
}


/*The thread will begin control in this function */
void *runner(void *param)
{
 int upper = atoi(param);
 int i;
 sum=0;
 if(upper > 0)
 {
  for(i=1; i <= upper;i++)
   sum += i;
 }
 pthread_exit(0);
}
```

Joining threads:

"Joining" is one way to accomplish synchronization between threads. The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates. The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit(). A joining thread can match one pthread_join() call. It is a logical error to attempt multiple joins on the same thread .

Example:

The following example illustrates passing a structure to the thread, hence overcoming single-valued arguments to threads, as well as, using thread joins

```
//Example 7.3

#include <pthread.h>
#include <stdio.h>


struct char_print
{
     char character;
     int count;
};

void *print (void *parameters)
{
     struct char_print *p;
     p = (struct char_print *) parameters;
     int i;
     for (i=0; i< p->count ; ++i)
          fputc(p->character, stderr);
```

```
     pthread_exit(NULL);
}

int main()
{
     pthread_t t1_id;
     pthread_t t2_id;

     struct char_print t1_args;
     struct char_print t2_args;



     t1_args.character='1';
     t1_args.count = 60;
     pthread_create(&t1_id, NULL, print, (void *) &t1_args);

     t2_args.character='0';
     t2_args.count = 100;
     pthread_create(&t2_id, NULL, print, (void *) &t2_args);

     pthread_join(t1_id,NULL);

     pthread_join(t2_id,NULL);

}
```

Thread Attributes:

Threads have a number of attributes that may be set at creation time. This is done by filling a thread attribute object attr of type *pthread_attr_t*, then passing it as second argument to *pthread_create*. Passing NULL is equivalent to passing a thread attribute object with all attributes set to their default values.

Examples of thread attributes are:

Attribute	**Description**
`detachstate'	Choose whether the thread is created in the joinable state (PTHREAD_CREATE_JOINABLE) or in the detached state (PTHREAD_CREATE_DETACHED). The default is PTHREAD_CREATE_JOINABLE. In the joinable state, another thread can synchronize on the thread termination and recover its termination code using pthread_join, but some of the thread resources are kept allocated after the thread terminates, and reclaimed only when another thread performs pthread_join on that thread. In the detached state, the thread resources are immediately freed when it terminates, but pthread_join cannot be used to synchronize on the thread termination. A thread created in the joinable state can later be put in the detached thread using pthread_detach.
`schedpolicy'	Select the scheduling policy for the thread: one of SCHED_OTHER (regular,

	non-realtime scheduling), SCHED_RR (realtime, round-robin) or SCHED_FIFO (realtime, first-in first-out). The default is SCHED_OTHER. The realtime scheduling policies SCHED_RR and SCHED_FIFO are available only to processes with superuser privileges (root user). pthread_attr_setschedparam will fail and return ENOTSUP if you try to set a realtime policy when you are unprivileged. The scheduling policy of a thread can be changed after creation with pthread_setschedparam.
schedparam'	Change the scheduling parameter (the scheduling priority) for the thread. The default is 0. This attribute is not significant if the scheduling policy is SCHED_OTHER; it only matters for the realtime policies SCHED_RR and SCHED_FIFO. The scheduling priority of a thread can be changed after creation with pthread_setschedparam.

Attribute objects are consulted only when creating a new thread. The same attribute object can be used for creating several threads. Modifying an attribute object after a call to *pthread_create* does not change the attributes of the thread previously created. unless you're you intend to modify the thread attributes as part of your program, you can use NULL as a value for the attribute. The following table summarizes the thread attribute related functions.

Function	**Description**
`int pthread_attr_init ( pthread_attr_t * attr)`	pthread_attr_init initializes the thread attribute object attr and fills it with default values for the attributes. Each attribute attrname can be individually set using the function pthread_attr_setattrname and retrieved using the function pthread_attr_getattrname.
`int pthread_attr_destroy ( pthread_attr_t * attr)`	pthread_attr_destroy destroys the attribute object pointed to by attr releasing any resources associated with it. attr is left in an undefined state, and you must not use it again in a call to any pthreads function until it has been reinitialized.
`int pthread_attr_setattr ( pthread_attr_t *obj, int value)`	Set attribute attr to value in the attribute object pointed to by obj. See below for a list of possible attributes and the values they can take. On success, these functions return 0.
`int pthread_attr_getattr ( const pthread_attr_t *obj, int * value)`	Store the current setting of attr in obj into the variable pointed to by value. These functions always return 0.

`Thread Identifiers:`

The following functions are helper functions provided by the pthread library, and come very handy while programming with threads.

Function	**Description**
pthread_self ( )	Returns the unique thread ID of the calling thread. The returned data object is opaque cannot be easily inspected.

pthread_equal (thread1, thread2)	Compares two thread IDs: If the two IDs are different 0 is returned, otherwise a non-zero value is returned. Because thread IDs are opaque objects, the C language equivalence operator == should not be used to compare two thread IDs

Project 1 (10 marks):

5- (6 marks) Write a program that uses 6 threads to compute the sum of an array of 3000 integers. The program should divide the set to 6 parts and give the 1st thread the first 500 integers , the 2nd thread the next 500 integers and so on. The 6 sums from these parts should then be added to give a total sum.
The program should display this total sum.

The program should read the dataset of a 3000 integers from a file. This file name should be given in the command line (argc, argv).

In order to aid you in this question some parts of the program have already been provided.

The data set of a 3000 integers is already provided in a file called **"input.txt"** . In this file every integer is written on a single separate line.

Skeleton code to start has already been provided in the file **"project1q5s.c".** In this file a function to get the filename from the command line and read the file has already been written. The function readfile() gets the filename, opens the file in read mode and copies it's contents into an array of integers of size 3000. This array has been defined and set as a global variable.

A Debugging and testing has also been provided called testSum(). This function sequentially computes the sum of the 3000 integers in the array and displays the total sum. This function is present to test against the answer provided by using the 6 threads. **Note :** This function is only for testing , please do not use this function in the final output of the program.

Your job is now to add to this existing code the implementation using 6 pthreads.

Output of the program : The program should display the 6 sums computed by the threads and the total sum when these 6 sums are added together.

e.g of the run of the program and how the output should look like:

```
Before Parallelization,running the original sequential program,
output is:

cse480@cse480-desktop:~$ ./project1q5s input.txt
Reading file: input.txt
Reading file Complete, integers stored in array.

Testing without threads, Sum is : 84295

After Parallelization with Pthreads , output should be :

Thread 1 Sum:  11798
Thread 2 Sum:  10426
Thread 3 Sum:  11757
```

```
Thread 4 Sum:  11387
Thread 5 Sum:  19166
Thread 6 Sum:  19761

Total Sum is:  84295
```

Files for the program : input.txt and project1q5s.c are available on the elearning page as "Project 1 files" with the Project 1 Instructions.

6- **(4 marks)** Write a program that does as follows : Create an array of 100 integers (You are free to create this array however you like, hard code it in the code, read from a file , take it from the command line). Create two threads that read this array. The 1st Thread should compute the sum of the 100 integers. The 2nd Thread should compute the Average of the 100 integers. Display the **Sum** and **Average** in the end.

Notes for all Programs :
-Provide the Source code ".c" files of all your programs
-Provide the output of the program in a text format.

- No credit will be awarded for programs that will not compile successfully (compiles with errors). So, make sure your program compiles successfully, and test it multiple times before submitting.

- Your submitted answer file must be in a text format only (not in Microsoft word, or image format for example)