1. Create, compile and run a Pthreads "Hello world" program

A. Create a simple Pthreads program that does the following:

- Main program creates several threads, each of which executes a "print hello" thread routine. The argument passed to that routine is their thread ID.
- Thread's "print hello" routine accepts the thread ID argument and prints "hello world from thread #".
- Then thread exit properly to finish its execution.
- Main program calls **pthread_exit** as the last thing it does.

B. Compile the program using: *gcc -pthread -o hello myhello.c*

C. Run your hello executable and notice its output.

2. Thread Scheduling

A. Write a program that will create 32 threads. Introduce a sleep(4) statement to help insure that all threads will be in existence at the same time. Also, each thread runs a loop of 1<= i <=10000 iterations and adds up value *sin(i)* to a local variable *sum*.

B. Compile and run the program. This exercise demonstrates how the OS scheduler behavior determines the order of thread completion. Notice the order in which thread output is displayed. Is it ever in the same order?

3. Argument Passing

1. WAP to pass a single argument like a priority_value in thread function.
2. WAP to pass multiple arguments in thread function through a structure like below:

```
struct thread_data
{
  int          thread_id;
  int priority_value;
  char message;
};
```

Each thread will have its own id, a priority and a message. The entire structure will be passed as argument to thread.

4. Use of Mutexes

1. WAP to decompose processing of computing sum of a array elements by distributing loop iterations among 5 threads. A global sum is maintained and a mutex variable is used to protect its updates. Compare the results with the sum computed using a single thread.

2. WAP to decompose the computation of dot product of two vectors of size 1000 among 4 threads. The main data is made available to all threads through a globally accessible structure as given below to allow the thread function to access its input data and place its output into the structure.

```
typedef struct
 {
  double     *a;    // for 1st input vector
  double     *b;    // for 2nd input vector
  double    sum;    // to store local sums
 } DATA;
```

Each thread works on a different part of data of length size *250* elements. The main thread waits for all the threads to complete their computations, and then it prints the resulting sum. This program requires a mutex to protect the global sum as each thread updates it with their partial sums. Execute the program several times and notice that the order in which threads update the global sum varies.

5. Use of Condition Variables

WAP that creates two threads; one thread waits for a variable v to become 0 and increments v if finds a chance, however another thread waits for v to become 1 and decrements v if gets a chance. Use condition variables.