No E-Mail submissions will be accepted.
Submission formats and file naming:
File name : firstName_lastName_lab_5
File format: pdf or MS Word format
e.g. Donald_Trump_lab_5.pdf

1) Use cocalc.com to run the code given below and answer the following questions:

```
gcc main1.c –lpthread -o main1.out
```

```
./main1.out & pstree -p | grep main1.out
```

main1.c

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
int fd, counter = 0;
void *myMethod(void *arg){
    int * id;
    id = (int *)arg;
    char st[]=" [%c] Hello from thread_%d.\n";
        for(int i=0; i<5; i++ ){
            sprintf(st, st, *id==1? '*': '+', *id);
            write(fd, st, strlen(st));
        }
        sleep(1);
        printf("Thread %d counter = %d \n", *id, ++counter);
    return NULL;
}

int main(int argc, char *argv[]){
    fd = open("myfile.txt",'a');
    if (fd<0){
        printf("If you see this message, delete myfile.txt using rm -f myfile.txt, then run the program again.\n");
        exit(0);
    }
    pthread_t threadMyMethodOne;
    pthread_t threadMyMethodTwo;
    int first_thread = 1, second_thread = 2;
    pthread_create(&threadMyMethodOne, NULL, myMethod, &first_thread);
    pthread_create(&threadMyMethodTwo, NULL, myMethod, &second_thread);
    pthread_join(threadMyMethodOne, NULL);
    pthread_join(threadMyMethodTwo, NULL);
    printf("The file (fd=%d) is  closed.\n", fd);
    close(fd);
    return 0;
}
```

1. Attach a screenshot of your output.

```
~$ ./main1.out & pstree -p | grep main1.out
[1] 1026
                          |           |-main1.out(1026)-+-{main1.out}(1029)
                          |           |                 `-{main1.out}(1030)
~$ Thread 2 counter = 2
Thread 1 counter = 1
The file (fd=3) is closed.
```

2. Based on the output of your code, can you provide an explanation on whether the global variable "counter" is shared between threads 1 and 2?

   The variable counter is shared between the 2 threads since threads running in the same process share the same memory space. We can see this by seing the counter only get incremented 1 time for each thread.

3. After running the program, provide a screenshot showing the contents of myfile.txt. You may need to change the file's permissions using a command like **chmod 444 myfile.txt**.

```
[*] Hello from thread_1.
[+] Hello from thread_2.
[*] Hello from thread_1.
[+] Hello from thread_2.
[*] Hello from thread_1.
[+] Hello from thread_2.
[*] Hello from thread_1.
[+] Hello from thread_2.
[*] Hello from thread_1.
[+] Hello from thread_2.
```

4. From the contents of myfile.txt, is the file myfile.txt shared between the two threads (yes/no)? Explain your answer.

   Yes, files are one of the items shared among threads in the same process. We can see this working by having both threads printing to the file

5. From the output, Obtain the *pid* and *tid* for the process and threads? Attach a screenshot of your output.

```
~$ ./main1.out & pstree -p | grep main1.out
[1] 1026
                          |           |-main1.out(1026)-+-{main1.out}(1029)
                          |           |                 `-{main1.out}(1030)
~$ Thread 2 counter = 2
Thread 1 counter = 1
```

Process ID (PID)	1026
Thread ID 1 (TID)	1029
Thread ID 2 (TID)	1030

6. Change the sleep time from 1 to 100, then recompile your program and run it using

```
./main1.out & pstree -p | grep main1.out
```

In the second terminal use **kill -9 <tid>** to terminate one of the threads. Does the main process remain alive after the thread is killed? Attach a screenshot of the output after terminating the thread.

```
~$ ./main1.out & pstree -p | grep main1.out
[1] 1863
                          |           |-main1.out(1863)-+-{main1.out}(1866)
                          |           |                 `-{main1.out}(1867)
~$
[1]+  Killed                  ./main1.out
```

No, the main process does not stay alive. Killing the thread terminates the entire main process and is output as such in the terminal.

2) Use the online C compiler

https://www.onlinegdb.com/online_c_compiler

to run your code and then answer the following questions.

```c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/wait.h>

int global_var = 0;

int main(){
  int status;
  int pid_t, fork_return;
  fork_return = fork();

  if (fork_return==0){
    global_var++;
    printf("Child  pid  = %d \t global_var = %d\n", getpid(), global_var);
  }else{
    global_var++;
    waitpid(-1, &status, 0);
    printf("Parent pid  = %d \t global_var = %d\n", getpid(), global_var);
  }
  return 0;
}
```

```
Child pid = 4470          global_var = 1
Parent pid = 4466         global_var = 1
```

1. From the output, determine the value of the global_var variable in the parent process.
   1
2. From the output, determine the value of the global_var variable in the child process.
   1
3. Based on your observations, explain whether the global_var variable is shared between the parent and child processes.

   The variable is shared among the children but each process gets the initialized value (0) and increment it once

3. Use the online compiler https://www.onlinegdb.com/online_c_compiler to run user_level.c and then answer the following questions.

a) Run the program and observe the output.

A. What do you notice about the interleaving of "Thread 1" and "Thread 2" messages?

   The threads are taking turns back and forth instead of running together

B. Which thread starts first?

Thread 1

b) Print the thread IDs inside each thread function.

1. Modify thread_1() and thread_2() to print the thread ID using

```
printf("Thread 1.\n"); ~> printf("Thread 1. PID: %d TID: %ld \n", getpid(), syscall(SYS_gettid));

printf("Thread 2.\n"); ~> printf("Thread 2. PID: %d TID: %ld \n", getpid(), syscall(SYS_gettid));
```

2. Does each thread have a different thread ID? Why or why not?

No, they both use the same Id

Since the program uses ucontext_t it is running both threads in the same process

c) Change the sleep(1) calls to sleep(100) in both thread functions.

A. What happens to the execution? Does one sleep call block both threads?

The first thread prints and pauses the program. Both threads are blocked

B. Explain why this happens in the context of user-level threading.

Because the threads are inside a single process so when one is slept the entire process is slept. The kernel treats the threads as one