Exercise 1 (1/3)

- In this exercise, you will work on implementing a simple publish/subscribe messaging system via pipes and named pipes. This exercise consists of two parts (I and II).

Part I:

- write a program *channel.c* which runs two processes (publisher and subscriber) and one (unnamed) pipe between them.
- The publisher process reads a message from stdin and sends it to the subscriber who prints the message to stdout.
- The maximum size of a single message is 1024 bytes.
- Submit **channel.c**.

Part II:

- Write a program *publisher.c* which forks n child process which in turn open n named pipes in the path (/tmp/ex1). The parent process reads messages from stdin and the child processes publish the messages to the subscribers.

Exercise 1 (2/3)

- The publisher program accepts n (command line argument) as the maximum number of subscribers and creates n named pipes in the path ($/tmp/ex1/s\{i\}$) where $i \in [1, n]$. For example, the publisher creates the named pipe ($/tmp/ex1/s1$) for the first subscriber.
- Write another program *subscriber.c* which accepts a number *id* (command line argument) indicates the subscriber's *index* and opens the named pipe ($/tmp/ex1/s\{id\}$), reads the messages and prints to its stdout.
- Write a script **ex1.sh** to run one publisher and **n** subscriber where **n** is read as a command line argument for the script where $(0 < n < 4)$. Run all subscribers and the publisher in separate shell windows (https://askubuntu.com/a/46630). Look at the output of all running subscribers and check whether they all received the message or some of them. Why we need to create n named pipes for n subscribers? Add the answer to the file **ex1.txt**.

Exercise 1 (3/3)

- Assume that all subscribers had subscribed to the channel. Do not create threads here, but only processes.
- The maximum size of a single message is 1024 bytes.
- The publisher/subscriber should always be ready to read/write messages from/to stdin/stdout. We can terminate them by sending termination signals such as **SIGINT** (Ctrl+C).
- Add your explanation to **ex1.txt** about your findings in this exercise with respect to inter-process communication via pipes and fifo files.
- Submit **publisher.c**, **subscriber.c**, **ex1.txt** and **ex1.sh**.
- **Note:** Ensure that the subscriber opens the pipe before you start writing to it, otherwise you would get **SIGPIPE** error.

Exercise 2

- Create a struct **Thread** which contains three fields *id* which holds the thread id, an integer number i and *message* as a string of size 256.
- Write a program **ex2.c** that creates an array of n threads using the struct **Thread**. The field i should store the index of the created thread (0 for the first thread and so on) whereas the field *id* contains the thread id and the field *message* contains the message "Hello from thread i" where i is the value of the field i. Each thread should output its *id* and *message*, then exit. Main program should inform about thread creation by displaying the message "Thread i is created".
- Do the threads print messages in the same order you created?
- Fix the program to force the order to be strictly Thread 1 is created, Thread 1 prints message, Thread 1 exits and so on.
- **Hint:** use gcc -pthread **ex2.c** to compile.
- Submit **ex2.c** and **ex2.sh** which runs the program.

Exercise 3 (1/3)

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
// primality test
bool is_prime(int n){
    if (n <= 1) return false;
    int i = 2;
    for (; i * i <= n; i++){
        if (n % i == 0)
            return false;
    }
    return true;
}

// Primes counter in [a, b)
int primes_count(int a, int b)
{
    int ret = 0;
    for (int i = a; i < b; i++)
        if (is_prime(i) != 0)
            ret++;
    return ret;
}
```

```c
// argument to the start_routine
// of the thread
typedef struct prime_request
{
    int a, b;
} prime_request;


// start_routine of the thread
void * prime_counter(void *arg)
{
    // get the request from arg
    prime_request req = ...

    // perform the request
    int *count = ...

    return ((void *)count);
}
```

Exercise 3 (2/3)

- Given the code snippets in the previous slide, write a multi threaded C program **ex3.c** that accepts two integers as command line arguments: n and m. The program prints the number of primes in the range $[0, n)$, distributing the computation *equally* onto m threads.
- Divide the interval $[0, n)$ into m equal sub intervals (except maybe for the last one), spawn a separate thread to count the number of primes in each of those subintervals, and sum their individual results to get the final answer. The final answer will be the number of primes in the range [0, n).
- In addition to **ex3.c**, write a shell script **ex3.sh** that compiles the program and calculates its execution time with $n = 10,000,000$ and $m \in \{1, 2, 4, 10, 100\}$. Store the 5 timing results in **ex3_res.txt**. Finally, add a brief explanation of the findings in **ex3_exp.txt**.

Exercise 3 (3/3)

- Submit **ex3.c**, **ex3.sh**, **ex3_exp.txt** and **ex3_res.txt**.
- **Hint:** use `pthread_join` and **time** shell command.
- **Note:** Your main tasks in this exercise are as follows:
  1. Complete the start routine of the threads.
  2. Set up the threads.
  3. Set up the threads arguments (prime requests).
  4. Pass requests to the threads.
  5. Collect prime count from each thread.
  6. Aggregate the prime counts.
  7. Print the final result.
  8. Clean up!

Exercise 4 (1/4)

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <pthread.h>

// primality test (from ex3)
bool is_prime(int n);

// The mutex
pthread_mutex_t global_lock =
    PTHREAD_MUTEX_INITIALIZER;

// Do not modify these variables
    directly, use the functions
    on the right side.
int k = 0;
int c = 0;

// input from command line
int n = 0;
```

```c
// get next prime candidate
int get_number_to_check()
{
    int ret = k;
    if (k != n)
        k++;
    return ret;
}

// increase prime counter
void increment_primes()
{
    c++;
}

// start_routine
void *check_primes(void *arg)
{
    while (1){
        // TODO
    }
}
```

Exercise 4 (2/4)

- In this exercise we are going to solve the problem from ex3 in a different way. We will distribute the primality checking computation of integers in $[0, n)$ between the m threads in a different manner.
- We will have a global state for the next number to check for primality, k, and the number of primes discovered thus far, c.
- Each of the m threads reads the global k, checks it for primality, and increments it. If it turned out to be prime, it increments the global c.
- It must be the case that no two threads check the same integer for primality, and the total of the thread checks must account for the interval $[0, n)$. Also, we want the thread execution to be as parallel as possible.

Exercise 4 (3/4)

You are given the function that checks an integer for primality, the functions for reading and incrementing the global k and c, and a mutex for handling k and c. Your task is as follows.

- Write the threads start_routine.
- Utilize the mutexes so that:
  - No two threads check the same integer for primality.
  - No integer is left unchecked by any of the threads.
  - The execution is as parallel as possible.
- Set up the threads and join them.
- Print the final result.
- Clean up!

Exercise 4 (4/4)

Note that you should not directly manipulate the global k and c. You are already given function to handle that for you. It is however your responsibility that the mutexes are set so that to prevent race conditions.

Write the functionality in `ex4.c`. Also, write a shell script `ex4.sh` that compiles the program and times its execution with $n = 10,000,000$ and $m \in \{1, 2, 4, 10, 100\}$. Store the 5 timing results in `ex4_res.txt`. Finally, provide a brief explanation of the results in `ex4_exp.txt` and compare them to the results from exercise 3.

Submit **ex4.c**, **ex4.sh**, **ex4_exp.txt** and **ex4_res.txt**

Useful tools:

- `pthread_mutex_t`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`