Systems Programming

Week 6 – Lab 11

Shared data

pthread_mutex_init(3p) - Linux manual page - man7.org

pthread_mutex_lock(3p) - Linux manual page - man7.org

pthread_mutex_destroy(3p) - Linux manual page - man7.org

The Linux Programming Interface – Sections 30.1

In this laboratory students will start to use locks/mutex to guarantee that the concurrent access to the same variable by multiple threads renders always a correct value.

The supplied code has two threads that loops over an array with random numbers and counts how many of them are prime. In this simplistic version, each thread iterates over the whole set of values and just increments a local variable. The main waits for the termination of every thread before existing.

In the following exercises students will implement different versions of this solution where the threads need to access and update shared shared data.

1 Exercise 1 - Sequential retrieving of values

Modify the provided program so that each thread verifies his own set of random numbers. Guarantee that each values is not verifies by two threads.

This work division is not defined by an expression or specific division algorithm, but by the order of access to the global array: each thread will access and process one of the unprocessed numbers on the array (the one in the lowest index).

The program should have a new variable that stores the index of the next number to be processed (**next_random_index**) and all the treads: access the value of **next_random_index** , and process the corresponding random number from the array, and increment **next_random_index**.

After each thread increment this index (**next_random_index)** it is guaranteed that other threads will not process the same number, but the next one in the array.

When the **next_random_index** reaches **LENGTH_ARRAY** every thread should exit.

2 Exercise 2 - Race condition

Experiment running the code with different data configuration, array length and number of thread:

- delete the comment on line **//rand_num_array[i] = i;**
- increase the array length to **100000**
- increase number of threads to 4 or 5

Try to find an execution where the sum of the numbers processed by all threads is not correct, as in the next example with an array of of integers from 0 to 100000:

```
Thread 3 found 2300 primes on 24518 numbers
Thread 0 found 2671 primes on 26882 numbers
Thread 1 found 2399 primes on 24878 numbers
Thread 2 found 2300 primes on 24329 numbers
```

3 Exercise 2 - Critical region / Mutual exclusion

Correct the previous code by defining a critical region for the **next_random_index** global variable.

To guarantee that the access to the critical region does not affect the final result of the program execution, implement mutual exclusion using **pthread_mutex**:

- **pthread_mutex_t** – data type corresponding a mutex
- **pthread_mutex_init** – creation and initialization of a mutex
- **pthread_mutex_lock** – function to call before accessing the critical region
- **pthread_mutex_unlock** – function after leaving the critical region
- **pthread_mutex_destroy** – to destroy the mutex before exiting the program

4 Exercise 4 - Storage of results

Modify the previous program so that each thread stores the prime numbers in a shared array:

- declare a global variable (called **prime_array**) of the same length of **rand_num_array**
- every time a prime number is found, store it in the new variable and increment the global counter of prime numbers.

After the **pthread_join()**, the main should print the number of prime numbers stored in the **prime_array**.

Use all the necessary mutexes to guarantee that all prime number are correctly stored.

5 Threads and zeroMQ

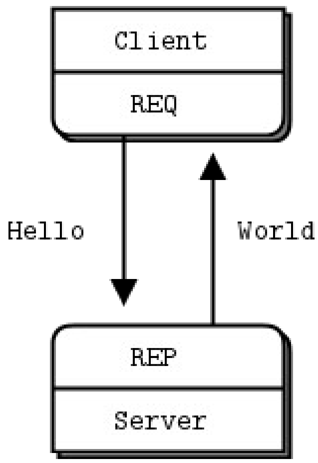
The only thread safe data structure in ZEROmq is the **context**, i.e. only the context can be created as a global variable and accessed by multiple threads. If a socket is used by multiple threads the communication (sending of receiving of messages) will work as expected.

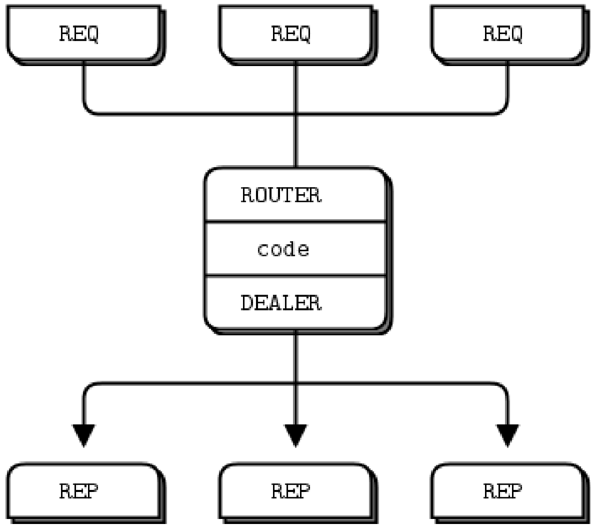
In a single thread server the various ZMQ_REQ sockets connects to a well know ZMQ_REP socket on the server:

`responder = zmq_socket (context, ZMQ_REP);` `zmq_bind (responder, addr);`	`requester = zmq_socket (context, ZMQ_REQ);` `zmq_connect (requester, addr);`

The server does a bind, while the client connects to that address. If various clients bind to the same address, the server will be able to read, process and reply sequentially to each one of the request from the various clients.

In order to have various threads reading from the same ZMQ_REP sockets and process concurrently the requests, it is necessary to use a Proxy that intermediates the processing of messages between the ZMQ_REQ and ZMQ_REP:





https://libzmq.readthedocs.io/en/latest/zmq_proxy.html

In order to have on the server multiple threads reading and answering to requests on the same server it is necessary to follow the next steps:

- **on the server main**
    - create a socket of type ZMQ_ROUTER (that will be accessed by the clients)
    - bind an address to the socket of type ZMQ_ROUTER
    - create a socket of type ZMQ_DEALER (that will be access by the various threads)
    - bind an address to the socket of type ZMQ_DEALER
- **on each server thread:**
    - create a socket of type ZMQ_REP
    - connect the socket to the address of the ZMQ_DEALER
- **on the server main**

- on the server (continued)
   - create a proxy that links the ZMQ_ROUTER and ZMQ_DEALER sockets
- **on the clients**
   - create a socket of type ZMQ_REQ
   - connect to the address of the ZMQ_ROUTER socket

After following this steps any request sent by client to the ZMQ_ROUTER will be forwarded to one of the threads connected to the ZMQ_DEALER. The replies will be forwarded to the correct client.

The sequence of C instructions are as follows:

SERVER - Global	
`void *  context;`	
SERVER - Main	**SERVER - Threads**

`context = zmq_ctx_new ();`
`void* frontend = zmq_socket(context, ZMQ_ROUTER);`
`zmq_bind (frontend, "ipc://fifo-pipe-front-end");`
 	
`void *backend = zmq_socket (context, ZMQ_DEALER);`
`zmq_bind (backend, "inproc://back-end");`	

`Create threads`	`void *responder = zmq_socket (context, ZMQ_REP);` `zmq_connect(resp, "inproc://back-end");`
`zmq_proxy (frontend, backend, NULL);`	
CLIENT	
`void *requester = zmq_socket (context, ZMQ_REQ);` `zmq_connect (requester, "ipc://fifo-pipe-front-end");`	

The client should continue to create a ZMQ_REQ socket, but now it must be connected to the address of the ZMQ_ROUTER.

For the ZMQ_ROUTER and ZMQ_REQ sockets any suitable socket should be used:

- **ipc** for communication in the same computer
- **tcp** for internet based communication.

For the ZMQ_DEALER and ZMQ_REP, besides **ipc** and **tcp** the **inproc** type of sockets can also be used. These sockets only work inside the same process and the address should be a single word:

https://libzmq.readthedocs.io/en/latest/zmq_inproc.html

6 Exercise 5

Modify the supplied code in **zmq-REP-mt** so that every number sent by the clients is verified whether it prime by one of the threads. Each thread will receive the requests from the clients (mediated by the proxy) verify if the received number is prime and return true(1) or false (0) wether the number is prime or not.

Every time a prime number is found, it should be stored in a array.

7 Fan-in

Some times multiple threads need to send data to a single thread, without needing for a reply. This can be accomplished with the ZMQ_PUSH and ZMQ_PULL sockets.

The thread that will receive the messages will create a socket of type ZMQ_PULL and bind to it an address. Any thread that need to send data to this sockets will need to:

- create a socket of type ZMQ_PUSH
- connect to the address of the ZMQ_PULL address

An example of this type of sockets is available in the **zmq-fan-in** folder.

8 Exercise 6

Modify the previous program so that another thread from the server prints the prime numbers as they are found.

9 Exercise 7

Modify the previous program so that when the user type **exit** the program print all the found prime numbers and terminates.