Politechnika Świętokrzyska
Kielce University of Technology

Operating systems 1

POSIX threads - pthreads

Tanmay Nandanikar | Lab report 7 | 11/27/20

dr inż. Karol Tomaszewski

INTRODUCTION

This lab primarily focused on multithreading in linux using POSIX threads or pthreads. Mutexes, POSIX semaphores, and conditional variables were also introduced in this lab.

1 BRIEF OVERVIEW OF THEORY

1.1 PTHREADS

Thread-based multitasking deals with the concurrent execution of pieces of the same program. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. C natively does not support multi-threading as it is provided by the Operating System. Pthreads provides an API for multithreading using C. Some important functions are listed below:

1. `pthread.h` : Library that needs to be included to use the pthreads API
2. `pthread_create (thread, attr, start_routine, arg):` Used to create a POSIX thread
3. `pthread_exit (status):` Used to exit a thread

1.2 MUTEXES

Mutexs are very similar to binary semaphores in form and function. The following functions are used to handle mutexes:

• The `pthread_mutex_init()` function is used to initialize mutexes. It accepts two arguments. The first is the mutex pointer, and the second is the attribute structure of type pthread_mutexattr_t containing mutex attributes. This function always returns zero. Mutex can be initialized directly, e.g. it can be assigned the value of pthread_mutex_initializer.

• The `pthread_mutex_lock()` function acquires a mutex or suspends current thread execution if the mutex has already been acquired. As an argument, it takes a pointer to mutex. Returns zero if successful or a non-zero value otherwise.

• The `pthread_mutex_unlock()` function releases the mutex. It takes mutex pointer as an argument. Returns values according to the same scheme as pthread_mutex_lock().

• The `pthread_mutex_trylock()` function works same as pthread_mutex_lock(), but if the mutex is busy, it doesn't block the current thread. In such case the function only returns an ebusy error.

1.3 POSIX SEMAPHORES

This is another implementation of semaphores, which can be used for both processes and threads unlike the System V implementation. The following functions are used to handle posix semaphores:

- The `sem_init()` function is used to initialize a semaphore. It accepts three arguments to invoke. The first argument is the semaphore pointer. The second is the flag indicating whether the semaphore will be available for threads or processes. In the latter case, the value of this argument must be zero. The third argument is the initial value of the semaphore (of type int). The function returns zero if executed correctly or -1 otherwise. Then it also sets the value of the variable errno.

- The `sem_post()` function is used to release a semaphore by increasing its value by one. If the resulting value is greater than zero and another thread has been put to sleep on the semaphore, it will be awakened. The function takes the semaphore pointer as an argument. It returns values according to the scheme described above.

- The `sem_wait()` function acquires a semaphore. As an argument, it takes a semaphore pointer. It puts the thread to sleep if the semaphore value is less or equal zero. Returns values in the same way as previously described functions.

- The `sem_trywait()` function acquires a semaphore, but does not put the thread to sleep when the semaphore value is zero. In such case it only returns an error (sets the variable errno value to eagain).

- The `sem_getvalue()` function returns the semaphore value by storing it in a variable of type int, to which the pointer is passed to it as the second argument of the call. As a first argument, the pointer to the semaphore is passed. The values are returned by this function according to the scheme described above.

- The `sem_destroy()` function removes the semaphore that was initialized with `sem_init()`. It takes a pointer to a semaphore as a call argument, and returns values according to the same scheme as the other functions that support semaphores for threads.

1.4 CONDITIONAL VARIABLES

These variables are used to synchronize threads so that necessary conditions dependent on other threads are fulfilled before proceeding to critical sections requiring these conditions to be fulfilled within the thread. Conditional variables can be handled by the following functions:

- The `pthread_cond_init()` function initializes a conditional variable. It takes two pointers as the call arguments. The first is a pointer to the conditional variable, and the second is a pointer to the structure of the variable attributes. The second argument is ignored by Linux and should have the value of null. This function, like all other functions related to conditional variables, returns zero if it performs correctly or a non-zero value otherwise.

• The `pthread_cond_signal()` function wakes up a single thread waiting on a conditional variable to which the pointer is passed as function argument.

• The `pthread_cond_wait()` function wakes up all threads waiting on the conditional variable to which the pointer is passed as function argument.

• The `pthread_cond_wait()` function allows the thread to wait for the condition to be fulfilled, previously unlocking the critical section by releasing the mutex that protects it. After waking the thread, this function will attempt to automatically acquire the mutex. Both operations on the mutex are performed in an indivisible manner. The function takes two arguments. One is a pointer to a conditional variable, and the other is a busy mutex. Since the wake-up of the thread may occur as a result of receiving a signal other than that sent using `pthread_cond_signal()` or `pthread_cond_broadcast()` functions, this function should be called in the loop until the event that the thread is waiting for occurs.

2 Exercises

2.1 Task 1

A program that utilizes POSIX threads. Source code can be found in the source section (here). Output for this task is shown below:

```
Process ID: 25980, Thread ID: 25981
Process ID: 25980, Thread ID: 25982
```

Output for task 1

2.2 Task 2

Thread 1 calculates sum of two numbers and thread 2 calculates the difference. Part of the source code is from Stackoverflow. Source code can be found in the source section (here). Output for this task is shown below:

```
10 + 32 = 42
10 - 32 = -22
```

Output for task 2

2.3 Task 3

Task 2 modified to return results to main function. Source code can be found in the source section (here). Output for this task is shown below:

```
10 + 32 = 42
```

```
10 - 32 = -22
```

Output for task 3

2.4 Task 4

Joined and detached threads. As can be seen, the detached thread gave the output first although it was initialized after the joined thread. I initially thought the difference was due to the first thread being joined later, but now I know I am wrong about this but I cannot explain the reason for this difference in output. Source code can be found in the source section (here). Output for this task is shown below:

```
10 - 32 = -22
10 + 32 = 42
```

Output for task 4

2.5 Task 5

Use of `sigaction()` to handle `SIGINT` signal from `pthread_kill()`. Source code can be found in the source section (here). Output for this task is shown below

```
Inside thread!!!
I handle signals
Inside thread!!!
```

Output for task 5

2.6 Task 6

Semaphore with initial value 2. Source code can be found in the source section (here). Output for this task is shown below:

```
Entered..

Entered..

Exiting...

Exiting...
```

Output for task 6

2.7 Task 7

Demonstration of a cleaning function. Source code can be found in the source section (here). Output for this task is shown below:

```
this is a new thread
before cancel
clean up ptr = dynamically allocated memory
process over
```

Output for task 7

2.8 Task 8

Demonstration of thread cancellation. Source code can be found in the source section (here). Output for this task is shown below:

```
In thread op no: 1
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
Thread op cancelled
```

Output for task 8

2.9 Task 9

I did not understand this question correctly.

2.10 Task 10

Enabling and disabling of thread cancellation. Source code can be found in the source section (here). Output for this task is shown below:

```
Not cancelled
Not cancelled
Not cancelled
Cancelled
```

Output for task 10

2.11 Task 11

Using a private variable. Source code can be found in the source section (here). Output for this task is shown below:

```
Data 1 = 123
Data 2 = 1234
```

Output for task 11

2.12 Task 12

Trying to use private key as a global variable from a different thread. Source code can be found in the source section (here). Output for this task is shown below:

```
Data 1 = 123
Segmentation fault (core dumped)
```

Output for task 12

2.13 Task 13

Thread stack size attribute. The stack size is displayed in bytes. Virtual stack size is 8 MB. This feature can be used to identify stack overflows when a thread tries to allocatee too much memory. It is Source code can be found in the source section (here). Output for this task is shown below:

```
Thread stack size = 8388608
This is a thread
This is a thread
```

Output for task 13

2.14 Task 14

The producer and consumer using a mutex and a conditional variable, part of the source code is from a YouTube tutorial. Source code can be found in the source section (here). Output for this task is shown below:

```
[ P ] Put value in the buffer, buffer = Message_01
[ P ] Go to sleep
[ C ] Take a value from the buffer, buffer = Message_0
[ C ] Go to sleep
[ P ] Put value in the buffer, buffer = Message_01
[ P ] Go to sleep
[ C ] Take a value from the buffer, buffer = Message_0
^C
```

Output for task 14

2.15 Task 15

The problem of readers and writers using a mutex, semaphore and conditional variable. It can handle multiple readers and 1 writer. Source code can be found in the source section (here). Output for this task is shown below:

```
Reader 2: read integer as 1
Reader 1: read integer as 1
Reader 6: read integer as 1
Reader 3: read integer as 1
Reader 4: read integer as 1
Reader 5: read integer as 1
Reader 8: read integer as 1
Reader 9: read integer as 1
Reader 7: read integer as 1
Reader 10: read integer as 1
Writer 3 modified integer to 2
Writer 1 modified integer to 4
Writer 2 modified integer to 8
Writer 5 modified integer to 16
Writer 4 modified integer to 32
```

Output for task 15

2.16 Task 16

I could not solve this problem.

3 Conclusion

During this lab, we discussed multithreading using POSIX threads. We learnt the POSIX implementation of semaphores and also discussed mutex locks. We also briefly covered conditional variables during this lab.

4 Source

4.1 Source code for Exercise One

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>


pid_t gettid(void);

void *myThread(void *vargp)
{

    printf("Process ID: %d, Thread ID: %d\n", getpid(), gettid());
    return NULL;
}

int main()
{
    pthread_t thread_id1, thread_id2;

    pthread_create(&thread_id1, NULL, myThread, NULL);
    pthread_join(thread_id1, NULL);
    pthread_create(&thread_id2, NULL, myThread, NULL);
    pthread_join(thread_id2, NULL);

    exit(0);
}
```

4.2 Source code for Exercise Two

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

typedef struct thread_data {
    int a;
    int b;
    int result;

} thread_data;

void *myThread(void *arg)
{
    thread_data *tdata=(thread_data *)arg;

    int a=tdata->a;
    int b=tdata->b;
    int result=a+b;

    tdata->result=result;
    printf("%d + %d = %d\n", tdata->a, tdata->b, tdata->result);

    pthread_exit(NULL);
}

void *myThread1(void *arg)
{
    thread_data *tdata=(thread_data *)arg;

    int a=tdata->a;
    int b=tdata->b;
    int result=a-b;

    tdata->result=result;
    printf("%d - %d = %d\n", tdata->a, tdata->b, tdata->result);

    pthread_exit(NULL);
}

int main()
{
    pthread_t tid1,tid2;
    thread_data tdata;
    tdata.a=10;
    tdata.b=32;

    pthread_create(&tid1, NULL, myThread, (void *)&tdata);
    pthread_join(tid1, NULL);

    pthread_create(&tid2, NULL, myThread1, (void *)&tdata);
    pthread_join(tid2, NULL);

    return 0;
}
```

4.3 Source code for Exercise Three

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

typedef struct thread_data {
    int a;
    int b;
    int result;

} thread_data;

void *myThread(void *arg)
{
    thread_data *tdata=(thread_data *)arg;

    int a=tdata->a;
    int b=tdata->b;
    int result=a+b;

    tdata->result=result;
    pthread_exit(NULL);
}

void *myThread1(void *arg)
{
    thread_data *tdata=(thread_data *)arg;

    int a=tdata->a;
    int b=tdata->b;
    int result=a-b;

    tdata->result=result;
    pthread_exit(NULL);
}

int main()
{
    pthread_t tid1,tid2;
    thread_data tdata;

    tdata.a=10;
    tdata.b=32;

    pthread_create(&tid1, NULL, myThread, (void *)&tdata);
    pthread_join(tid1, NULL);

    printf("%d + %d = %d\n", tdata.a, tdata.b, tdata.result);

    pthread_create(&tid2, NULL, myThread1, (void *)&tdata);
    pthread_join(tid2, NULL);

    printf("%d - %d = %d\n", tdata.a, tdata.b, tdata.result);

    return 0;
}
```

4.4 Source code for Exercise Four

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

typedef struct thread_data {
    int a;
    int b;
    int result;

} thread_data;

void *myThread(void *arg)
{
    thread_data *tdata=(thread_data *)arg;

    int a=tdata->a;
    int b=tdata->b;
    int result=a+b;

    tdata->result=result;
    printf("%d + %d = %d\n", tdata->a, tdata->b, tdata->result);

    pthread_exit(NULL);
}

void *myThread1(void *arg)
{
    thread_data *tdata=(thread_data *)arg;
    int a=tdata->a;
    int b=tdata->b;
    int result=a-b;
    tdata->result=result;
    printf("%d - %d = %d\n", tdata->a, tdata->b, tdata->result);
    pthread_exit(NULL);
}

int main()
{
```

```c
    int new_attr;
    pthread_attr_t attr;
    pthread_t tid1,tid2;
    thread_data tdata;
    tdata.a=10;
    tdata.b=32;
    new_attr=pthread_attr_init(&attr);
    new_attr=pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_DETACHED);
    pthread_create(&tid1, NULL, myThread, (void *)&tdata);
    pthread_create(&tid2, &attr, myThread1, (void *)&tdata);
    pthread_attr_destroy(&attr);
    pthread_join(tid1, NULL);
    return 0;
}
```

4.5 Source code for Exercise Five

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void handler(int num){
    write(STDOUT_FILENO, "I handle signals\n",17);
}

void *func(void *foo) {
    struct sigaction sa;
    sa.sa_handler = handler;
    sigaction(SIGINT, &sa, NULL);

    write(STDOUT_FILENO, "Inside thread!!!\n", 17);
    sleep(2);
    write(STDOUT_FILENO, "Inside thread!!!\n", 17);
    sleep(2);
    pthread_exit(NULL);
}

int main(void) {
    int result;
    struct sigaction sa;
    sa.sa_handler = handler;
    pthread_t thread;
    sigaction(SIGINT, &sa, NULL);
    if ((result = pthread_create(&thread, NULL, func, 0)) != 0) {
        perror("pthread create");
    }
    sleep(1);
    pthread_kill(thread, SIGINT);
    pthread_join(thread, NULL);

    return 0;
}
```

4.6 Source code for Exercise Six

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t sem1;

void* thread(void* arg)
{
    sem_wait(&sem1);
    printf("\nEntered..\n");

    sleep(4);

    printf("\nExiting...\n");
    sem_post(&sem1);
}

int main()
{
    sem_init(&sem1, 0, 2);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&sem1);
    return 0;
}
```

4.7 Source code for Exercise Seven

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <string.h>

void cleanup(void *arg)
{
    printf("clean up ptr = %s\n", (char *)arg);
    free((char *)arg);
}

void *thread(void *arg)
{
    char *ptr = NULL;

    printf("this is a new thread\n");
    ptr = (char*)malloc(100);
    pthread_cleanup_push(cleanup, (void*)(ptr));
            bzero(ptr, 100);
            strcpy(ptr, "dynamically allocated memory");
            sleep(3);

            printf("before pop\n");

    pthread_cleanup_pop(1);
    return NULL;
}

int main()
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    sleep(1);
    printf("before cancel\n");

    pthread_cancel(tid);
    pthread_join(tid,NULL);
    printf("process over\n");
    return 0;
}
```

4.8 Source code for Exercise Eight

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

typedef struct thread_data {
    int a;
    int cancel;
} thread_data;


void *myThread(void *arg)
{
    thread_data *tdata=(thread_data *)arg;
    usleep((rand()%45)*10000 + 10000);
    if(tdata->cancel==1){
        printf("Thread op cancelled\n");
        pthread_exit(NULL);
    }else{
        tdata->cancel=1;
    }

    tdata->a=tdata->a+1;
    printf("In thread op no: %d\n", tdata->a);
    pthread_exit(NULL);
}

int main()
{
    int new_attr,i;
    pthread_attr_t attr;
    pthread_t tid[20];
    thread_data tdata;
    tdata.a=0;
    tdata.cancel=0;

    for(i=0;i<20;i++)
        pthread_create(&tid[i], NULL, myThread, (void *)&tdata);

    for(i=0;i<20;i++)
        pthread_join(tid[i], NULL);
    return 0;
}
```

4.9 *Source code for Exercise Nine*

4.10 Source code for Exercise Ten

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

void *thread_cancel(void *arg)
{
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    while(1)
    {
        sleep(1);
        printf("Not cancelled\n");
        sleep(1);
        printf("Not cancelled\n");
        sleep(1);
        printf("Not cancelled\n");
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid1;
    int ret  = 0;
    pthread_create(&tid1, NULL, thread_cancel, NULL);
    if(ret!=0)
        perror("pthread_create");
    pthread_cancel(tid1);
    sleep(5);
    printf("Cancelled\n");
    pthread_join(tid1, NULL);
    return 0;
}
```

4.11 SOURCE CODE FOR EXERCISE ELEVEN

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>


void *myThread(void *vargp)
{
    pthread_key_t key1=0;
    int data=123;
    pthread_key_create(&key1, NULL);

    pthread_setspecific(key1, (void*) &data);
    int *x = pthread_getspecific(key1);
    printf("Data 1 = %d\n",*x);
    return NULL;
}

void *myThread1(void *vargp)
{
    pthread_key_t key2=0;
    int data1=1234;
    pthread_key_create(&key2, NULL);

    pthread_setspecific(key2, (void*) &data1);
    int *x = pthread_getspecific(key2);
    printf("Data 2 = %d \n",*x);
    return NULL;
}

int main()
{
    pthread_t thread_id1, thread_id2;

    pthread_create(&thread_id1, NULL, myThread, NULL);
    pthread_join(thread_id1, NULL);
    pthread_create(&thread_id2, NULL, myThread1, NULL);
    pthread_join(thread_id2, NULL);


    exit(0);
}
```

4.12 Source code for Exercise Twelve

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>

pthread_key_t key;

void *myThread(void *vargp)
{
    pthread_key_t key1=0;
    int data=123;
    pthread_key_create(&key1, NULL);

    pthread_setspecific(key1, (void*) &data);
    int *x = pthread_getspecific(key1);
    key=key1;
    printf("Data 1 = %d\n",*x);
    return NULL;
}

void *myThread1(void *vargp)
{
    pthread_key_t key2=0;
    int data1=1234;
    pthread_key_create(&key2, NULL);

    pthread_setspecific(key2, (void*) &data1);
    int *x = pthread_getspecific(key2);
    int *y = pthread_getspecific(key);
    printf("Data 2 = %d\nTest %d \n",*x, *y);
    return NULL;
}

int main()
{
    pthread_t thread_id1, thread_id2;

    pthread_create(&thread_id1, NULL, myThread, NULL);
    pthread_join(thread_id1, NULL);
    pthread_create(&thread_id2, NULL, myThread1, NULL);
    pthread_join(thread_id2, NULL);


    exit(0);
}
```

4.13 SOURCE CODE FOR EXERCISE THIRTEEN

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

void *reader(void *wno)
{
    printf("This is a thread\n");
    pthread_exit(NULL);
}

int main()
{
    pthread_t test;
    size_t stacksize;

    pthread_create(&test, NULL, (void *)reader, NULL);
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_getstacksize(&attr, &stacksize);

    printf("Thread stack size = %ld \n", stacksize);

    return 0;
}
```

4.14 SOURCE CODE FOR EXERCISE FOURTEEN

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

#define BUFF_SIZE 10

pthread_cond_t wakeup_consumer, wakeup_producer;
pthread_mutex_t mutex;
struct Buffer{
    int cur_size;
    char buffer[BUFF_SIZE];
} buffer;

char data[BUFF_SIZE] = {'M', 'e', 's', 's', 'a', 'g', 'e', '_', '0',
'1'};

void* producer(void* argv){
    while (1){
        pthread_mutex_lock(&mutex);
        if (buffer.cur_size == BUFF_SIZE){
            printf("[ P ] Go to sleep\n");
            while (buffer.cur_size == BUFF_SIZE){
                pthread_cond_signal(&wakeup_consumer);
                pthread_cond_wait(&wakeup_producer, &mutex);
                pthread_cond_init(&wakeup_producer, 0);
            }
        }
        buffer.buffer[buffer.cur_size] = data[buffer.cur_size];
        buffer.cur_size += 1;
        if(buffer.cur_size == BUFF_SIZE) {
            printf("[ P ] Put value in the buffer, buffer = %s\n",
buffer.buffer);
            sleep(1);
        }
        pthread_mutex_unlock(&mutex);
    }
}

void* consumer(void* argv){
    while (1){
        pthread_mutex_lock(&mutex);
        if (buffer.cur_size == 0){
            printf("[ C ] Go to sleep\n");
            while (buffer.cur_size == 0){
                pthread_cond_signal(&wakeup_producer);
                pthread_cond_wait(&wakeup_consumer, &mutex);
                pthread_cond_init(&wakeup_consumer, 0);
            }
        }
        buffer.buffer[buffer.cur_size-1] = '\0';

        if(buffer.cur_size==10) {
            printf("[ C ] Take a value from the buffer, buffer = %s\n",
buffer.buffer);
            sleep(1);
        }
        buffer.cur_size -= 1;

        pthread_mutex_unlock(&mutex);
    }
}

int main(){
    buffer.cur_size = 0;
    pthread_cond_init(&wakeup_consumer, 0);
    pthread_cond_init(&wakeup_producer, 0);
    pthread_mutex_init(&mutex, NULL);
    pthread_t p, c;

    pthread_create(&p, NULL, producer, NULL);
    pthread_create(&c, NULL, consumer, NULL);

    pthread_join(p, 0);
    pthread_join(c, 0);
    pthread_cond_destroy(&wakeup_consumer);
    pthread_cond_destroy(&wakeup_producer);
    return 0;
}
```

4.15 Source code for Exercise Fifteen

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

sem_t wrt;
pthread_mutex_t mutex;
int integer = 1;
int numreader = 0;

void *writer(void *wno)
{
    sem_wait(&wrt);
    integer = integer*2;
    printf("Writer %d modified integer to %d\n",(*((int
*)wno)),integer);
    sem_post(&wrt);

}
void *reader(void *rno)
{
    pthread_mutex_lock(&mutex);
    numreader++;
    if(numreader == 1) {
        sem_wait(&wrt);
    }
    pthread_mutex_unlock(&mutex);

    printf("Reader %d: read integer as %d\n",*((int *)rno),integer);

    pthread_mutex_lock(&mutex);
    numreader--;
    if(numreader == 0) {
        sem_post(&wrt);
    }
    pthread_mutex_unlock(&mutex);
    usleep(20);
}

int main()
{
    pthread_t read[10],write[5];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&wrt,0,1);

    int a[10] = {1,2,3,4,5,6,7,8,9,10};

    for(int i = 0; i < 10; i++) {
        pthread_create(&read[i], NULL, (void *)reader, (void *)&a[i]);
    }
    for(int i = 0; i < 5; i++) {
        pthread_create(&write[i], NULL, (void *)writer, (void *)&a[i]);
    }
    for(int i = 0; i < 10; i++) {
        pthread_join(read[i], NULL);
        usleep(100);
    }
    for(int i = 0; i < 5; i++) {
        pthread_join(write[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    sem_destroy(&wrt);

    return 0;
}
```