Week 3 Concurrency

Mayank Motwani

June 2024

1 The Concurrency Module: Pthreads Synchronization

Hello everyone, My name is Mayank Motwani and I will be your host for this part of the Unix flight. Hope you have enjoyed the prior parts of this trip. So, concurrency is an optimization paradigm and a fundamental building block for modern computing systems. It allows us to use a system's resources to their full potential. However, I would like you to consider this part as more of a game or puzzle.

We would be prepping with some theory for the problems that are ahead. You can find the theory in the theory folder. In my opinion, the theory is very straightforward since we won't be going into the gory details of how locks or other synchronization primitives are created/implemented using hardware/software assistance. We will just use the pthreads library as a blackbox for problem solving. However, the resources will be mentioned just in case.

Please familiarize yourself with the pthreads API thoroughly. You can check out Prof. Mythili Vutukuru's website for help. Many helpful tutorials and sample programs are available online. Practice writing simple programs with multiple threads, using locks and condition variables for synchronization across threads. Remember that you must include the header file ¡pthread.h¿ and compile code using the -lpthread flag when using this API.

So, let us set some ground rules. First of all, with the idea of concurrency, the first thing that comes to our mind is that of which synchronization primitive to use. I find conditional variables very intuitive and since it's my module, I don't care of your opinions(just kidding :), it is just easier to check if all of you use only one primitive). So, I would request you to use only pthreads_mutex_t locks and pthread_cond_t conditional variables.

2 Basic Problems

2.1 Problem 1: Baby example

Given below is a code that involves update of a global counter variable using two threads. Presently, without synchronization, if you compile the code given below and run it, it will lead to an incorrect update of the counter variable due to race conditions. Your task is to add synchronization to achieve correct update of global counter in all conditions. Although this is a very basic example, there is one optimization possible (I can see atleast one, don't know about more), try to find out what it is.

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

int counter = 0;

void *increment(void *arg){
    for(int i = 0; i < 10000; i++){
        counter++;
    }
    return NULL;
}

int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("counter = %d\n", counter);
    return 0;
}
```

2.1.1 Problem 1 Solution

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

int counter = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *increment(void *arg){
    for(int i = 0; i < 10000; i++){
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("counter = %d\n", counter);
    return 0;
}
```

2.2 Problem 2: Multiplex

This is an extension of the baby example, we want to allow some number of threads (an upper bound) to work in the critical section at the same time, also we want an efficient implementation. So, what you want is to allow only a specific range of threads to work in the critical section. The question is vague for a reason, and you can implement it any way you like with as much optimization as you want.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 100
#define MAX_PERMIT 2
void *critical(void *arg)
{
    /*
    critical section - maybe some server
    which can hold a load of only 15(people)
    and for safety, we want to ensure only 12
    people can access at a given time
    */
}

int main()
{
    pthread_t threads[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++){
        pthread_create(&threads[i], NULL, critical, NULL);
    }
    for (int i = 0; i < NUM_THREADS; i++){
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

2.2.1 Problem 2 Solution

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 100
#define MAX_PERMIT 2
int thread_counter = 0;

pthread_mutex_t thread_counter_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t accessory_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t thread_cv = PTHREAD_COND_INITIALIZER;

void *critical(void *arg)
{
    if(thread_counter == MAX_PERMIT){
        pthread_cond_wait(&thread_cv, &accessory_lock);
    }

    pthread_mutex_lock(&thread_counter_lock);
    thread_counter++;
    pthread_mutex_unlock(&thread_counter_lock);

    /*
    critical section - maybe some server
    which can hold a load of only 15(people)
    and for safety, we want to ensure only 12
    people can access at a given time
    */

    pthread_mutex_lock(&thread_counter_lock);
    if(thread_counter > MAX_PERMIT){
        printf("PANIC\n");
    }
    // printf("%d\n", thread_counter);
    thread_counter--;
    pthread_cond_signal(&thread_cv);
    pthread_mutex_unlock(&thread_counter_lock);
}

int main()
{
    pthread_t threads[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++)
    {
        pthread_create(&threads[i], NULL, critical, NULL);
    }
    for (int i = 0; i < NUM_THREADS; i++)
    {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

2.3 Problem 3: Serial

The following code mentions two functions A and B printing something, we would like to ensure A happens before B

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

void *A(void* arg_A){
    printf("Hello There\n");
    return NULL;
}

void *B(void* arg_B){
    printf("Bye Bye\n");
    return NULL;
}

int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, A, NULL);
    pthread_create(&t2, NULL, B, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

2.3.1 Problem 3 Solution

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

pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv1 = PTHREAD_COND_INITIALIZER;

void *A(void* arg_A){
    printf("Hello There\n");
    pthread_cond_signal(&cv1);
    return NULL;
}

void *B(void* arg_B){
    if(1){
        pthread_cond_wait(&cv1, &lock1);
    }
    printf("Bye Bye\n");
    return NULL;
}

int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, A, NULL);
    pthread_create(&t2, NULL, B, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

2.4 Problem 4: Rendezvous

This problem is similar to Serial. I will informally explain what needs to be done. What rendezvous means is that you want two threads to reach particular points p1 and p2 in their code trace and when thread1 reaches p1, it should wait for thread2 to reach p2 and vice versa. In this case, you want P1 to print A, and P2 to print 1 and then only continue their further execution. Hope this explains the problem. Don't hesitate to ask if you don't understand.

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

//the primary goal is to ensure the order of the alphabet and the
    numbers remains the same
void *P1(void *arg){
    printf("A\n");
    printf("2\n");
    return NULL;
}

void *P2(void *arg){
    printf("1\n");
    printf("B\n");
    return NULL;
}

int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, P1, NULL);
    pthread_create(&t2, NULL, P2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

2.4.1 Problem 4 Solution

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

int counter1 = 0;
int counter2 = 0;
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t equal = PTHREAD_COND_INITIALIZER;

//the primary goal is to ensure the order of the alphabet and the numbers remains the same
void *P1(void *arg){
    printf("A\n");

    pthread_mutex_lock(&lock1);
    counter1 = 1;
    pthread_cond_signal(&equal);
    pthread_mutex_unlock(&lock1);

    pthread_mutex_lock(&lock2);
    while(counter1 != counter2){
        pthread_cond_wait(&equal, &lock2);
    }
    pthread_mutex_unlock(&lock2);

    printf("2\n");

    return NULL;
}

void *P2(void *arg){
    printf("1\n");

    pthread_mutex_lock(&lock2);
    counter2 = 1;
    pthread_cond_signal(&equal);
    pthread_mutex_unlock(&lock2);

    pthread_mutex_lock(&lock1);
    while(counter1 != counter2){
        pthread_cond_wait(&equal, &lock1);
    }
    pthread_mutex_unlock(&lock1);

    printf("B\n");

    return NULL;
}

int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, P1, NULL);
    pthread_create(&t2, NULL, P2, NULL);
    pthread_join(t1, NULL);
```

```
55      pthread_join(t2, NULL);
56      return 0;
57  }
```

2.5 Problem 5: Barrier

This is an easy generalization of the rendezvous problem for more than two threads. You want multiple threads to reach specific points in their execution and wait for other threads to reach their own specific points before any of them can continue further execution. In this case, you want all hellos printed before all byes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 20

void *critical(void *arg)
{
    printf("Hello\n");

    printf("Bye\n");
}

int main()
{
    pthread_t threads[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++)
    {
        pthread_create(&threads[i], NULL, critical, NULL);
    }
    for (int i = 0; i < NUM_THREADS; i++)
    {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

2.5.1 Problem 5 Solution

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 20

int counter = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t cvlock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

void *critical(void *arg)
{
    printf("Hello\n");
    // sleep(0.002);

    pthread_mutex_lock(&lock);
    counter++;
    pthread_cond_broadcast(&cv);
    pthread_mutex_unlock(&lock);

    pthread_mutex_lock(&lock);
    while (counter != NUM_THREADS)
    {
        pthread_cond_wait(&cv, &lock);
    }
    pthread_mutex_unlock(&lock);

    printf("Bye\n");
}

int main()
{
    pthread_t threads[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++)
    {
        pthread_create(&threads[i], NULL, critical, NULL);
    }
    for (int i = 0; i < NUM_THREADS; i++)
    {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

2.6 Some More Practice Problems

The problems given below are very similar to the ones discussed above and are there just for your practice

- **Problem 1**: Write a program that has a counter as a global variable. Spawn 1000 threads in the program, and let each thread increment the counter 1000 times in a loop. Print the final value of the counter after all the threads finish—the expected value of the counter is 1000000. Run this program first without using locking across threads, and observe the incorrect update of the counter due to race conditions (the final value will be slightly less than 1000000). Next, use locks when accessing the shared counter and verify that the counter is now updated correctly.

- **Problem 2**: Write a program where the main default thread spawns N threads. Thread i should print the message "I am thread i" to screen and exit. The main thread should wait for all N threads to finish, then print the message "I am the main thread", and exit.

- **Problem 3**: Write a program where the main default thread spawns N threads. When started, thread i should sleep for a random interval between 1 and 10 seconds, print the message "I am thread i" to screen, and exit. Without any synchronization between the threads, the threads will print their messages in any order. Add suitable synchronization using condition variables such that the threads print their messages in the order 1, 2, ..., N . You may want to start with N = 2 and then move on to larger values of N.

- **Problem 4**: Write a program with N threads. Thread i must print number i in a continuous loop. Without any synchronization between the threads, the threads will print their numbers in any order. Now, add synchronization to your code such that the numbers are printed in the order 1, 2, ..., N , 1, 2, ..., N , and so on. You may want to start with N = 2 and then move on to larger values of N.

2.6.1 Practice Problem 1 Solution

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

int counter = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* inc(void* arg){
    for(int i = 0; i < 1000; i++){
      pthread_mutex_lock(&lock);
        counter++;
      pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main(){
    pthread_t ids[1000];
    for(int i = 0; i < 1000; i++){
        pthread_create(&ids[i], NULL, inc, NULL);
    }
    for(int i = 0; i  < 1000; i++){
        pthread_join(ids[i], NULL);
    }
    printf("counter: %d\n", counter);
    return 0;
}
```

2.6.2 Practice Problem 2 Solution

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

void* inc(void* arg){
    int i = *(int*)arg;
    printf("I am thread %d\n", i);
    return NULL;
}

int main(){
    int N;
    scanf("%d", &N);
    pthread_t ids[N];
    int args[N];
    for(int i = 0; i < N; i++){
        args[i] = i;
        pthread_create(&(ids[i]), NULL, inc, &(args[i]));
    }
    for(int i = 0; i  < N; i++){
        pthread_join(ids[i], NULL);
    }
    printf("I am the main thread\n");
}
```

2.6.3 Practice Problem 3 Solution

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

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

int current = 0;

void* inc(void* arg){
    sleep(rand() % 10);
    int i = *(int*)arg;
    pthread_mutex_lock(&lock);
    while(current != i){
        pthread_cond_wait(&cond, &lock);
    }
    printf("I am thread %d\n", i);
    current++;
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(){
    int N;
    scanf("%d", &N);
    pthread_t ids[N];
    int args[N];
    for(int i = 0; i < N; i++){
      args[i] = i;
        pthread_create(&(ids[i]), NULL, inc, &(args[i]));
    }
    for(int i = 0; i < N; i++){
        pthread_join(ids[i], NULL);
    }
    printf("I am the main thread\n");
}
```

2.6.4 Practice Problem 4 Solution

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

pthread_cond_t cond;
pthread_mutex_t lock;

int current = 0;

void* inc(void* arg){
    int i = ((int*)arg)[0];
    int N = ((int*)arg)[1];
    while(1){
        pthread_mutex_lock(&lock);
        while(current != i){
            pthread_cond_wait(&cond, &lock);
        }
        printf("%d\n", i + 1);
        current = (current + 1) % N;
        pthread_cond_broadcast(&cond);
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main(){
    pthread_cond_init(&cond, NULL);
    pthread_mutex_init(&lock, NULL);
    int N;
    scanf("%d", &N);
    pthread_t ids[N];
    int args[N][2];
    for(int i = 0; i < N; i++){
        args[i][0] = i;
        args[i][1] = N;
        pthread_create(&(ids[i]), NULL, inc, &(args[i]));
    }
    for(int i = 0; i  < N; i++){
        pthread_join(ids[i], NULL);
    }
    printf("I am the main thread\n");
}
```

3 Classical Synchronization Problems

In this part, we examine the classical problems that appear in literature. They are usually presented in terms of real-world problems. These problems are analogous to common problems that operating systems need to solve.

3.1 The Producer Consumer Problem

3.1.1 Vague problem statement to get an idea

In multithreaded programs there is often a division of labor between threads.

In one common pattern, some threads are producers and some are consumers. Producers create items of some kind and add them to a data structure; consumers remove the items and process them.

Event-driven programs are a good example. An "event" is something that happens that requires the program to respond: the user presses a key or moves the mouse, a block of data arrives from the disk, a packet arrives from the network, a pending operation completes.

Whenever an event occurs, a producer thread creates an event object and adds it to the event buffer. Concurrently, consumer threads take events out of the buffer and process them. In this case, the consumers are called "event handlers."

There are several synchronization constraints that we need to enforce to make this system work correctly:

- While an item is being added to or removed from the buffer, the buffer is in an inconsistent state. Therefore, threads must have exclusive access to the buffer.
- If a consumer thread arrives while the buffer is empty, it blocks until a producer adds a new item.

Assume that producers perform the following operations over and over: Basic producer code:

1. event = waitForEvent()
2. buffer.add(event)

Also, assume that consumers perform the following operations: Basic consumer code:

1. event = buffer.get()
2. event.process()

As specified above, access to the buffer has to be exclusive, but waitForEvent and event.process can run concurrently.

Add synchronization statements to the producer and consumer code to enforce the synchronization constraints

3.1.2 Better problem statement

You can find this in the practice problems

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

#define PROD_THREADS 25
#define CONS_THREADS 25
#define ARR_SIZE 12

int common_array[ARR_SIZE];

void *prod_func(void *args){
    int random_entry = rand();
    int entered_entry = ((float)random_entry/RAND_MAX)*5;

    /*
    enter the above entry to the common array
    */

    printf("Entry %d added to the array\n", entered_entry);
    return NULL;
}

void* cons_func(void *args){

    /*
    procure entries from the common array and print them
    */

    // enter a print statement to print the value procured/consumed
     from the array
    return NULL;
}

int main(){
    pthread_t prod[PROD_THREADS], cons[CONS_THREADS];
    for(int i = 0; i < ARR_SIZE; i++){
        common_array[i] = -1;
    }
    for(int i = 0; i < PROD_THREADS; i++){
        pthread_create(&prod[i], NULL, prod_func, NULL);
    }
    for(int j = 0; j < CONS_THREADS; j++){
        pthread_create(&cons[j], NULL, cons_func, NULL);
    }
    for(int i = 0; i < PROD_THREADS; i++){
        pthread_join(prod[i], NULL);
    }
    for(int j = 0; j < CONS_THREADS; j++){
        pthread_join(cons[j], NULL);
    }
    return 0;
}
```

3.1.3 Producer Consumer Solution

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

#define PROD_THREADS 25
#define CONS_THREADS 25
#define ARR_SIZE 12

int common_array[ARR_SIZE];
int prod_counter = 0;
int cons_counter = 0;
int elements = 0;

pthread_mutex_t array_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t prod_cv = PTHREAD_COND_INITIALIZER;
pthread_cond_t cons_cv = PTHREAD_COND_INITIALIZER;

void *prod_func(void *args){
    int random_entry = rand();
    int entered_entry = ((float)random_entry/RAND_MAX)*5;

    pthread_mutex_lock(&array_lock);
    if(elements == ARR_SIZE){
        pthread_cond_wait(&prod_cv, &array_lock);
    }
    common_array[prod_counter++] = entered_entry;
    prod_counter = prod_counter % ARR_SIZE;
    elements++;
    pthread_mutex_unlock(&array_lock);

    pthread_cond_signal(&cons_cv);
    printf("Entry %d added to the array\n", entered_entry);

    return NULL;
}

void* cons_func(void *args){
    pthread_mutex_lock(&array_lock);
    if(elements == 0){
        pthread_cond_wait(&cons_cv, &array_lock);
    }
    int extracted_entry = common_array[cons_counter];
    common_array[cons_counter++] = -1;
    cons_counter = cons_counter % ARR_SIZE;
```

```
    elements--;
    pthread_mutex_unlock(&array_lock);

    pthread_cond_signal(&prod_cv);
    printf("Entry %d removed from the array\n", extracted_entry);
    return NULL;
}

int main(){
    pthread_t prod[PROD_THREADS], cons[CONS_THREADS];
    for(int i = 0; i < ARR_SIZE; i++){
        common_array[i] = -1;
    }
    for(int i = 0; i < PROD_THREADS; i++){
        pthread_create(&prod[i], NULL, prod_func, NULL);
    }
    for(int j = 0; j < CONS_THREADS; j++){
        pthread_create(&cons[j], NULL, cons_func, NULL);
    }
    for(int i = 0; i < PROD_THREADS; i++){
        pthread_join(prod[i], NULL);
    }
    for(int j = 0; j < CONS_THREADS; j++){
        pthread_join(cons[j], NULL);
    }
    return 0;
}
```

3.2 The Reader Writer Problem

3.2.1 Vague problem statement to get an idea

The next classical problem, called the Reader-Writer Problem, pertains to any situation where a data structure, database, or file system is read and modified by concurrent threads. While the data structure is being written or modified it is often necessary to bar other threads from reading, in order to prevent a reader from interrupting a modification in progress and reading inconsistent or invalid data.

As in the producer-consumer problem, the solution is asymmetric. Readers and writers execute different code before entering the critical section. The synchronization constraints are:

1. Any number of readers can be in the critical section simultaneously.

2. Writers must have exclusive access to the critical section.

In other words, a writer cannot enter the critical section while any other thread (reader or writer) is there, and while the writer is there, no other thread may enter.

The exclusion pattern here might be called categorical mutual exclusion. A thread in the critical section does not necessarily exclude other threads, but the presence of one category in the critical section excludes other categories.

Enforce these constraints, while allowing readers and writers to access the data structure, and avoiding the possibility of deadlock.

There are two ways to solve the above problem: if there is a reader is the critical section and there is a writer in the queue, you could stop all incoming readers and allow the writer to do it's work after the current readers have left the critical section or you could say fuck off to the writer and allow all incoming readers to do their work and only when there is a time when there are no readers active in the critical section(which may not come in an online system and thus the writer may starve), then the writer might get a chance to do it's work.

Ok, so I will be vague in this problem, you can implement this any way you like, if you want to follow a structured approach, just look at the problem statement of the practice problems. Else, you can just make your own code of updating an array using writers in a continuous fashion(read block 0,1,. . . ,MAX_SIZE-1, 0,1,. . .) and also read in this fashion using a readers. I have written a sample code(which is hopefully correct), you can see it for reference(and please tell if you find mistakes in the synchronization logic, but please try it on your own first)

3.2.2 Better problem statement

You can find this in the practice problems

3.3 A Sample Solution for Reader Writer Problem

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

#define READERS 40
#define WRITERS 30
#define ARR_SIZE 12

int common_array[ARR_SIZE];

void *read_func(void *args)
{
    int reader_number = *(int *)args;
    while (1)
    {
        sleep(2);
        int temp_read_counter = -1;
        pthread_mutex_lock(&array_lock);
        while (writer_access == 1)
        {
            pthread_cond_wait(&cv_read, &array_lock);
        }
        if (reader_access == 0)
        {
            reader_access = 1;
        }
        reader_in_demand++;
        temp_read_counter = reader_counter;
        reader_counter++;
        reader_counter = reader_counter % ARR_SIZE;
        pthread_mutex_unlock(&array_lock);

        int seen = common_array[reader_counter];
        printf("Reader %d reads number %d at index %d\n",
    reader_number, seen, reader_counter);

        pthread_mutex_lock(&array_lock);
        reader_in_demand--;
        if (reader_in_demand == 0)
        {
            reader_access = 0;
            pthread_cond_signal(&cv_write);
        }
        pthread_mutex_unlock(&array_lock);
    }
}

void *write_func(void *args)
{
    int writer_number = *(int *)args;
    int random = rand();
    random = ((float)random / RAND_MAX) + writer_number;

    while (1)
```

```
55      {
56          sleep(2);
57          pthread_mutex_lock(&array_lock);
58          while (reader_access == 1)
59          {
60              pthread_cond_wait(&cv_write, &array_lock);
61          }
62          writer_access = 1;
63          common_array[writer_counter++] = random;
64          writer_counter = writer_counter % ARR_SIZE;
65          printf("Writer %d writes number %d at index %d\n",
    writer_number, random, writer_counter);
66          writer_access = 0;
67          pthread_cond_broadcast(&cv_read);
68          pthread_mutex_unlock(&array_lock);
69      }
70  }
71
72  int main()
73  {
74      pthread_t reader[READERS], writer[WRITERS], checker;
75      for (int i = 0; i < ARR_SIZE; i++){
76          common_array[i] = -1;
77      }
78      for (int i = 0; i < READERS; i++){
79          pthread_create(&reader[i], NULL, read_func, &i);
80      }
81      for (int j = 0; j < WRITERS; j++){
82          pthread_create(&writer[j], NULL, write_func, &j);
83      }
84      pthread_create(&checker, NULL, checker1, NULL);
85      for (int i = 0; i < READERS; i++){
86          pthread_join(reader[i], NULL);
87      }
88      for (int j = 0; j < WRITERS; j++) {
89          pthread_join(writer[j], NULL);
90      }
91      return 0;
92  }
```