Operating Systems

Lab 05: Pthread

Goals:

Learn to work with Pthread.

Policies:

It should go without saying that all the work that you will turn in for this lab will be yours. Do not surf the web to get inspiration for this assignment, do not include code that was not written by you. You should try your best to debug your code on your own, but it's fine to get help from a colleague as long as that means getting assistance to identify the problem and doesn't go as far as receiving source code to fix it (in writing or orally).

Contents:

1. What is POSIX thread (pthread) libraries?

1) The POSIX (Portable Operating System Interface) thread libraries are a standards-based thread API.
2) The purpose of using the POSIX thread library in your software is to execute software faster.

2. Creating A thread - pthread_create()

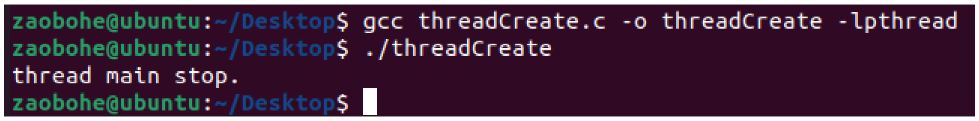
```
int pthread_create(pthread_t * thread,
                const pthread_attr_t * attr,
                void * (*start_routine)(void *),
                void *arg);
```

- **thread:** pointer to an unsigned integer value that returns the thread id of the thread created.
- **attr:** pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
- **start_routine:** pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void *. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
- **arg:** pointer to void that contains the arguments to the function defined in the earlier argument.
- **Return:** return 0 on success; If an error occurs, the error number is returned.

Example 1 of pthread_create():

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

/**
 * This method runs in a child thread.
 * @param args
 * @return
 */
void *threadA_run(void *arg) {
    printf("thread A start...\n");
    sleep(1000); // TODO something
    printf("thread A stop.\n");
    return NULL;
}

int main() {
    // This is a unsigned integer.You may understand as id or handle.
    pthread_t threadA;
    pthread_create(&threadA, NULL, threadA_run, NULL);
    printf("thread main stop.\n");
    return 0;
}
```

Execution reults:



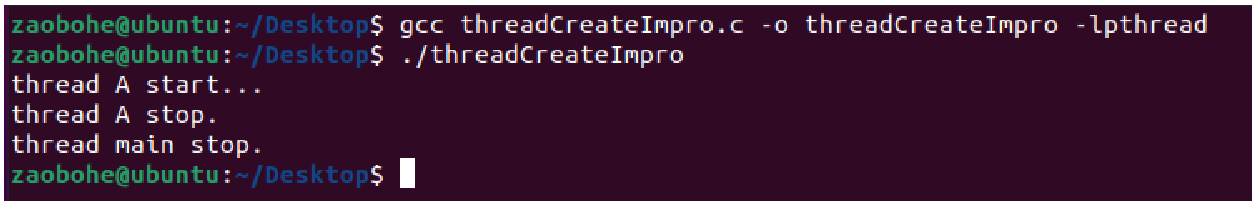
```
zaobohe@ubuntu:~/Desktop$ gcc threadCreate.c -o threadCreate -lpthread
zaobohe@ubuntu:~/Desktop$ ./threadCreate
thread main stop.
zaobohe@ubuntu:~/Desktop$
```

Thread A has no chance to execute because child thread is automatically destroyed by the system after the main thread exits by default.

Example 2 of pthread_create(): improve Example 1

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
/**
 * This method runs in a child thread.
 * @param args
 * @return
 */
void *threadA_run(void *arg) {
    printf("thread A start...\n");
    sleep(1000); // TODO something
    printf("thread A stop.\n");
    return NULL;
}

int main() {
    pthread_t threadA;
    pthread_create(&threadA, NULL, threadA_run, NULL);
    sleep(1100);
    printf("thread main stop.\n");
    return 0;
}
```

Execution results:


```
zaobohe@ubuntu:~/Desktop$ gcc threadCreateImpro.c -o threadCreateImpro -lpthread
zaobohe@ubuntu:~/Desktop$ ./threadCreateImpro
thread A start...
thread A stop.
thread main stop.
zaobohe@ubuntu:~/Desktop$ 
```

Thread A has a chance to finish because the main thread has slept long enough to exit. In practice, the execution time of the child thread is not fixed. How does the main thread wait for the child thread to exit? You can do this with the pthread_join function.

3. Used to wait for the termination of thread - pthread_join()

int pthread_join(pthread_t * thread,

void **thread_return);

- **thread:** thread id of the thread for which the current thread waits.
- **thread_return:** pointer to the location where the exit status of the thread mentioned in thread is stored.

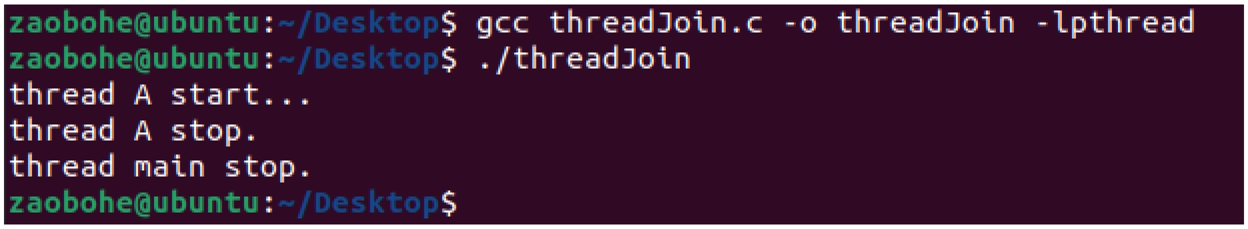
- **Return:** Returns 0 on success and an error code on failure

The calling thread waits for another thread, Thread, to exit. If the thread is still executing, the calling function is blocked until the Thread finishes. If it has finished, the calling thread returns directly.

Example 1 of pthread_join():

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

/**
 * This method runs in a child thread.
 * @param args
 * @return
 */
void *threadA_run(void *arg) {
    printf("thread A start...\n");
    sleep(5); // TODO something
    printf("thread A stop.\n");
    return NULL;
}

int main() {
    pthread_t threadA; // This is a unsigned integer.You may understand as id or
handle.
    char *retval;
    char arg[128];
    pthread_create(&threadA, NULL, threadA_run, NULL);
    pthread_join(threadA, NULL);
    printf("thread main stop.\n");
    return 0;
}
```

Execution results:



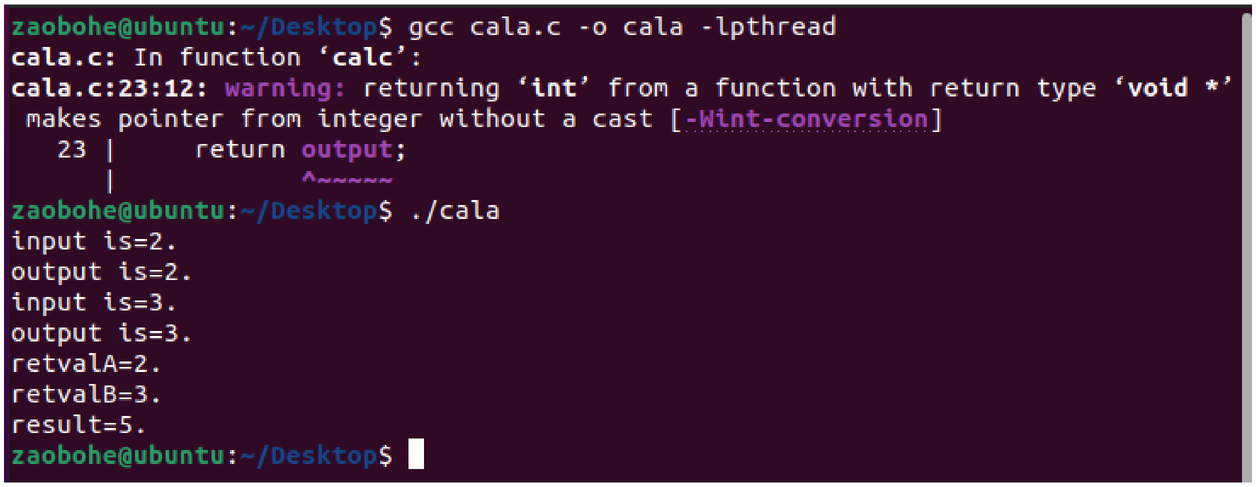
```
zaobohe@ubuntu:~/Desktop$ gcc threadJoin.c -o threadJoin -lpthread
zaobohe@ubuntu:~/Desktop$ ./threadJoin
thread A start...
thread A stop.
thread main stop.
zaobohe@ubuntu:~/Desktop$
```

4. Threads Collaboration

Example of threads collaboration:

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

/**
 * To calc data.
 * @param calc input
 * @return calc output
 */
void *calc(void *arg) {
    int *input = (int *) arg;
    int output = 0;

    printf("input is=%d.\n", *input);

    while ((*input)--) {
        output++;
    }

    printf("output is=%d.\n", output);

    return output;
}


int main() {
    pthread_t threadA, threadB;
    long retvalA, retvalB;
    long result;
    int argA = 2, argB = 3;
    pthread_create(&threadA, NULL, calc, &argA);
    pthread_create(&threadB, NULL, calc, &argB);
    pthread_join(threadA, (void *) &retvalA);
    pthread_join(threadB, (void *) &retvalB);

    printf("retvalA=%ld.\n", retvalA);
    printf("retvalB=%ld.\n", retvalB);
    result = retvalA + retvalB;
    printf("result=%ld.\n", result);
    return 0;
}
```

Execution result:



```
zaobohe@ubuntu:~/Desktop$ gcc cala.c -o cala -lpthread
cala.c: In function ‘calc’:
cala.c:23:12: warning: returning ‘int’ from a function with return type ‘void *’
 makes pointer from integer without a cast [-Wint-conversion]
   23 |     return output;
      |            ^~~~~~
zaobohe@ubuntu:~/Desktop$ ./cala
input is=2.
output is=2.
input is=3.
output is=3.
retvalA=2.
retvalB=3.
result=5.
zaobohe@ubuntu:~/Desktop$ 
```

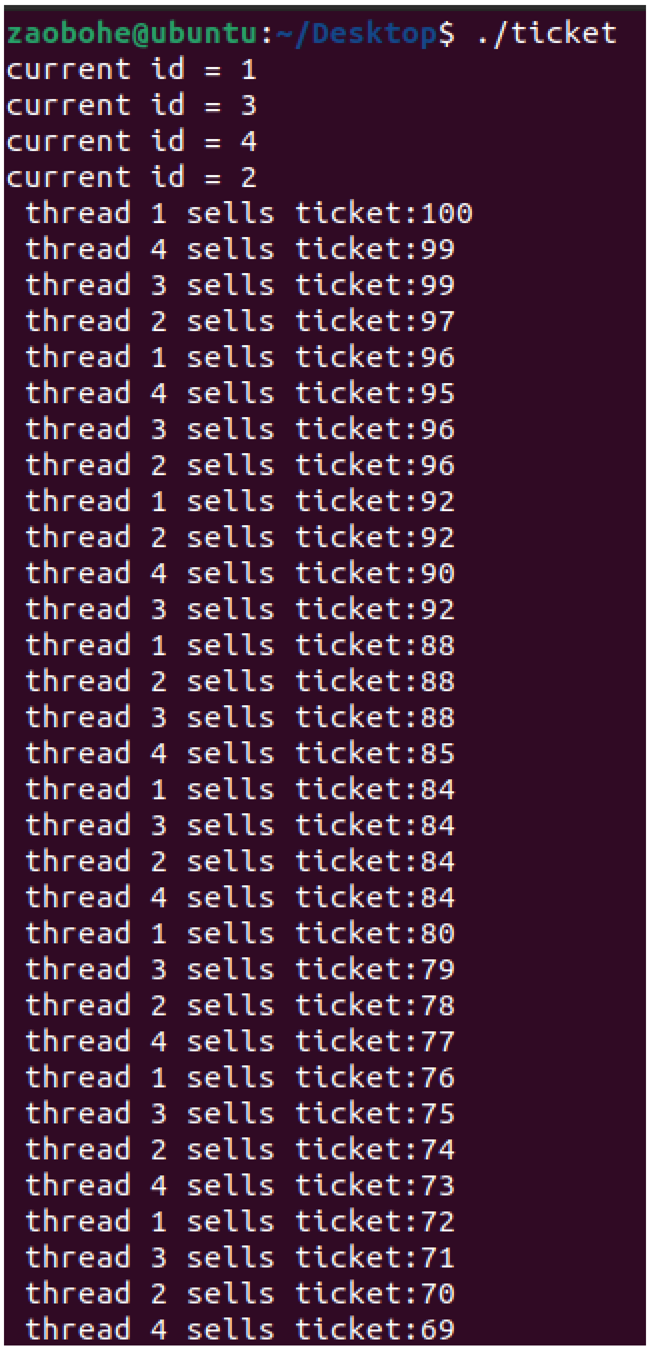
The above example shows three threads working together to complete a calculation. The calculation process for retvalA and retvalB does not depend on each other, so it can be done in parallel. However, the calculation of result must wait for both retvalA and retvalB to be calculated, so we use pthread_join to wait for the end of thread A and thread B respectively. This example also covers how to pass an argument to a subthread function and how to get the return value of the subthread function (the execution state of the subprogram).

5. Thread collaboration

Example of threads which cannot collaborate with each other in a right way:

```
  // 操作共享变量会有问题的售票系统代码
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
int ticket = 100;
void *route( void *arg)
{
   char id = *(char*)arg;
   printf("current id = %c\n", id);
   while (  1 )  {
      if (  ticket > 0 )  {
         usleep( 1000) ;
         printf( " thread %c sells ticket:%d\n" , id, ticket) ;
         ticket--;
      }  else {
         break;
      }
   }
}
int main(void)
{
   pthread_t t1, t2, t3, t4;
   char a1='1',a2='2',a3='3',a4='4';
   pthread_create( &t1, NULL, route, &a1);
   pthread_create( &t2, NULL, route, &a2);
   pthread_create( &t3, NULL, route, &a3);
   pthread_create( &t4, NULL, route, &a4);
   pthread_join( t1, NULL) ;
   pthread_join( t2, NULL) ;
   pthread_join( t3, NULL) ;
   pthread_join( t4, NULL) ;
}
```

Execution results:



```
zaobohe@ubuntu:~/Desktop$ ./ticket
current id = 1
current id = 3
current id = 4
current id = 2
 thread 1 sells ticket:100
 thread 4 sells ticket:99
 thread 3 sells ticket:99
 thread 2 sells ticket:97
 thread 1 sells ticket:96
 thread 4 sells ticket:95
 thread 3 sells ticket:96
 thread 2 sells ticket:96
 thread 1 sells ticket:92
 thread 2 sells ticket:92
 thread 4 sells ticket:90
 thread 3 sells ticket:92
 thread 1 sells ticket:88
 thread 2 sells ticket:88
 thread 3 sells ticket:88
 thread 4 sells ticket:85
 thread 1 sells ticket:84
 thread 3 sells ticket:84
 thread 2 sells ticket:84
 thread 4 sells ticket:84
 thread 1 sells ticket:80
 thread 3 sells ticket:79
 thread 2 sells ticket:78
 thread 4 sells ticket:77
 thread 1 sells ticket:76
 thread 3 sells ticket:75
 thread 2 sells ticket:74
 thread 4 sells ticket:73
 thread 1 sells ticket:72
 thread 3 sells ticket:71
 thread 2 sells ticket:70
 thread 4 sells ticket:69
```

Why can't we get the right result?

After the if statement determines that the condition is true, the code can switch to other threads concurrently; usleep is a process that simulates a long busy operation, in which many threads may enter the code segment

Ticket -- The operation itself is not an atomic operation

6. Pthread_Mutex – Thread synchronization

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a critical section. Processes' access to critical section is controlled by using synchronization techniques. When one thread starts executing the critical section (a serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a race condition where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

Mutex

- A Mutex is a lock that we set before using a shared resource and release after using it.
- When the lock is set, no other thread can access the locked region of code.
- So we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code.
- So this ensures synchronized access of shared resources in the code.

Working of a mutex

1) Suppose one thread has locked a region of code using mutex and is executing that piece of code.
2) Now if scheduler decides to do a context switch, then all the other threads which are ready to execute the same region are unblocked.
3) Only one of all the threads would make it to the execution but if this thread tries to execute the same region of code that is already locked then it will again go to sleep.
4) Context switch will take place again and again but no thread would be able to execute the locked region of code until the mutex lock over it is released.
5) Mutex lock will only be released by the thread who locked it.
6) So this ensures that once a thread has locked a piece of code then no other thread can execute the same region until it is unlocked by the thread who locked it.

A mutex is initialized and then a lock is achieved by calling the following two functions: The first function initializes a mutex and through second function any critical region in the code can be locked.

1) **Define a mutex:**

```
pthread_mutex_t  mutex;
```

2) **Initialize a mutex:**

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr)
```

   a) **mutex:** creates a mutex, referenced by mutex.
   b) **attr:** with attributes specified by attr, can be NULL.
   c) **Returned value:** if successful, returns 0, and the state of the mutex becomes **initialized and unlocked**. If unsuccessful, returns -1.

3) **Lock a mutex:**

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

   a) **Function:** If the mutex is already locked by another thread, the thread waits for the mutex to become available.
   b) **Function:** The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it.
   c) **Returned value:** If successful, returns 0. If unsuccessful, returns -1.

The mutex can be **unlocked** and **destroyed** by calling following two functions: The first function releases the lock and the second function destroys the lock so that it cannot be used anywhere in future.

4) **Unlock a mutex:**

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

   a) **Returned value:** If successful, returns 0. If unsuccessful, returns -1.

5) **Deletes a mutex:**

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

   a) **Returned value:** If successful, returns 0. If unsuccessful, returns -1.

```c
#include <pthread.h>
/**
 * 初始化一个锁。
 * @param __mutex 锁
 * @param __mutexattr 属性
 * @return 如果成功返回 0，失败返回错误码
 */
int pthread_mutex_init (pthread_mutex_t *__mutex,
                               const pthread_mutexattr_t *__mutexattr);
/**
 * 销毁一个锁。
 * @param __mutex 锁
 * @return 如果成功返回 0，失败返回错误码
 */
int pthread_mutex_destroy (pthread_mutex_t *__mutex);
/**
 * 上锁。
 * @param mutex 锁
 * @return 如果成功返回 0，失败返回错误码
 */
int pthread_mutex_lock(pthread_mutex_t *mutex);
/**
 * 尝试上锁。
 * @param mutex 锁
 * @return 如果成功返回 0，失败返回错误码
 */
int pthread_mutex_trylock(pthread_mutex_t *mutex);
/**
 * 解锁。
 * @param __mutex 锁
 * @return 如果成功返回 0，失败返回错误码
 */
int pthread_mutex_unlock (pthread_mutex_t *__mutex);
```

Example of thread sychornization:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
int ticket = 100;
pthread_mutex_t  mutexBuf = PTHREAD_MUTEX_INITIALIZER;

void *route( void *arg)
{
   char id = *(char*)arg;
   printf("current id = %c\n", id);
   while (  1 )  {
      if (  ticket > 0 )  {
         usleep( 1000) ;
          pthread_mutex_lock(&mutexBuf);
         printf( " thread %c sells ticket:%d\n" , id, ticket) ;
         ticket--;
          pthread_mutex_unlock(&mutexBuf);
      }  else {
         break;
      }
   }
}
int main(void)
{
   pthread_t t1, t2, t3, t4;
   char a1='1',a2='2',a3='3',a4='4';
   pthread_create( &t1, NULL, route, &a1);
   pthread_create( &t2, NULL, route, &a2);
   pthread_create( &t3, NULL, route, &a3);
   pthread_create( &t4, NULL, route, &a4);
   pthread_join( t1, NULL) ;
   pthread_join( t2, NULL) ;
   pthread_join( t3, NULL) ;
   pthread_join( t4, NULL) ;
}
```

7. Producer-Consumer Problem

1) Condition variable:

Producer consumption problem to solve another problem is the synchronization problem. When a thread accesses a variable mutually exclusively, it may find that it can do nothing until the other thread changes state. For example, when a thread accesses a queue and finds the queue empty, it can only wait until another thread adds a node to the queue. In this case, the condition variable is needed.

A. Define a condition variable:

```
pthread_cond_t cond;
```

B. Initialize a condition variable:

```
pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *restrict attr)
```

cond：condition variable
attr：variable attribute: NULL

C. Wait a condition variable:

```
int pthread_cond_wait(pthread_cond_t *restrict cond,pthread_mutex_t *restrict mutex);
```

cond：wait on this condition
mutex：mutex

D. Release a condition variable:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

cond：signal on this condition

E. Delete a condition variable:

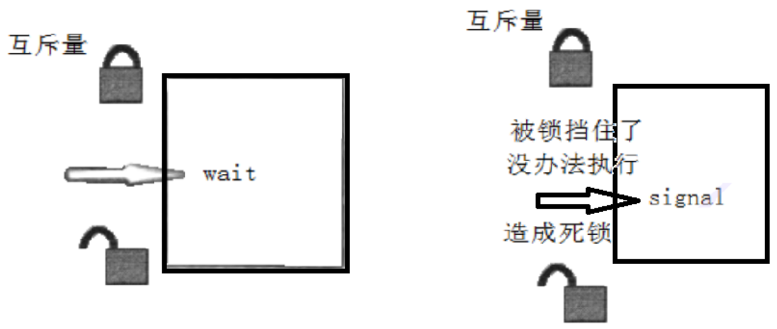
```
int pthread_cond_ destroy (pthread_cond_t *cond);
```

cond：destroy on this condition

2) Why does pthread_cond_wait need a mutex?

int pthread_cond_wait(cond, mutex);

1) Conditional waiting is a means of synchronization between threads. If there is only one thread, the condition is not met, and it will not be met all the time. Therefore, there must be a thread through some operations to change the shared variable, so that the original unmet condition becomes satisfied, and friendly notification waiting on the thread in the condition variable.
2) Conditions will not suddenly become satisfied for no reason at all, and will inevitably involve changes in shared data. So be sure to protect it with mutexes. Shared data cannot be safely obtained or modified without a mutex.



Therefore, the **WRONG** design:

```
// Wrong design
pthread_mutex_lock(&mutex);
while (condition_is_false) {
   pthread_mutex_unlock(&mutex);
   //解锁之后，等待之前，条件可能已经满足，信号已经发出，但是该信号可能被错过
   pthread_cond_wait(&cond);
   pthread_mutex_lock(&mutex);
}
pthread_mutex_unlock(&mutex);
```

1) Because unlocking and waiting are not atomic operations. If, after the call is unlocked and before pthread_cond_wait, another thread has already acquired the mutex and sent a signal, if the condition is satisfied, then pthread_cond_wait will miss the signal, possibly causing the thread to block permanently on the pthread_cond_wait. So unlocking and waiting must be an atomic operation.
2) int pthread_cond_wait: when we go into this function, we're going to see if the conditional variable is 0? If it equals 0, and changes the mutex to 1. Until cond_wait returns, changing the condition to 1 and restoring the mutex to its original form.

Pseudo code of Producer-Consumer Problem:

Waiting for the condition:

```
pthread_mutex_lock(&mutex);
while (the condition is false）
    pthread_cond_wait(&cond, &mutex);
        //pthread_cond_wait 会先解除之前的 pthread_mutex_lock 锁定的 mutex，
        //然后阻塞在等待队列里休眠，直到再次被唤醒
        //（大多数情况下是等待的条件成立而被唤醒，唤醒后，
        //该进程会进行 pthread_mutex_lock(&mutex)先锁定，然后再读取资源
修改条件
pthread_mutex_unlock(&mutex);
```

Setting the condition:

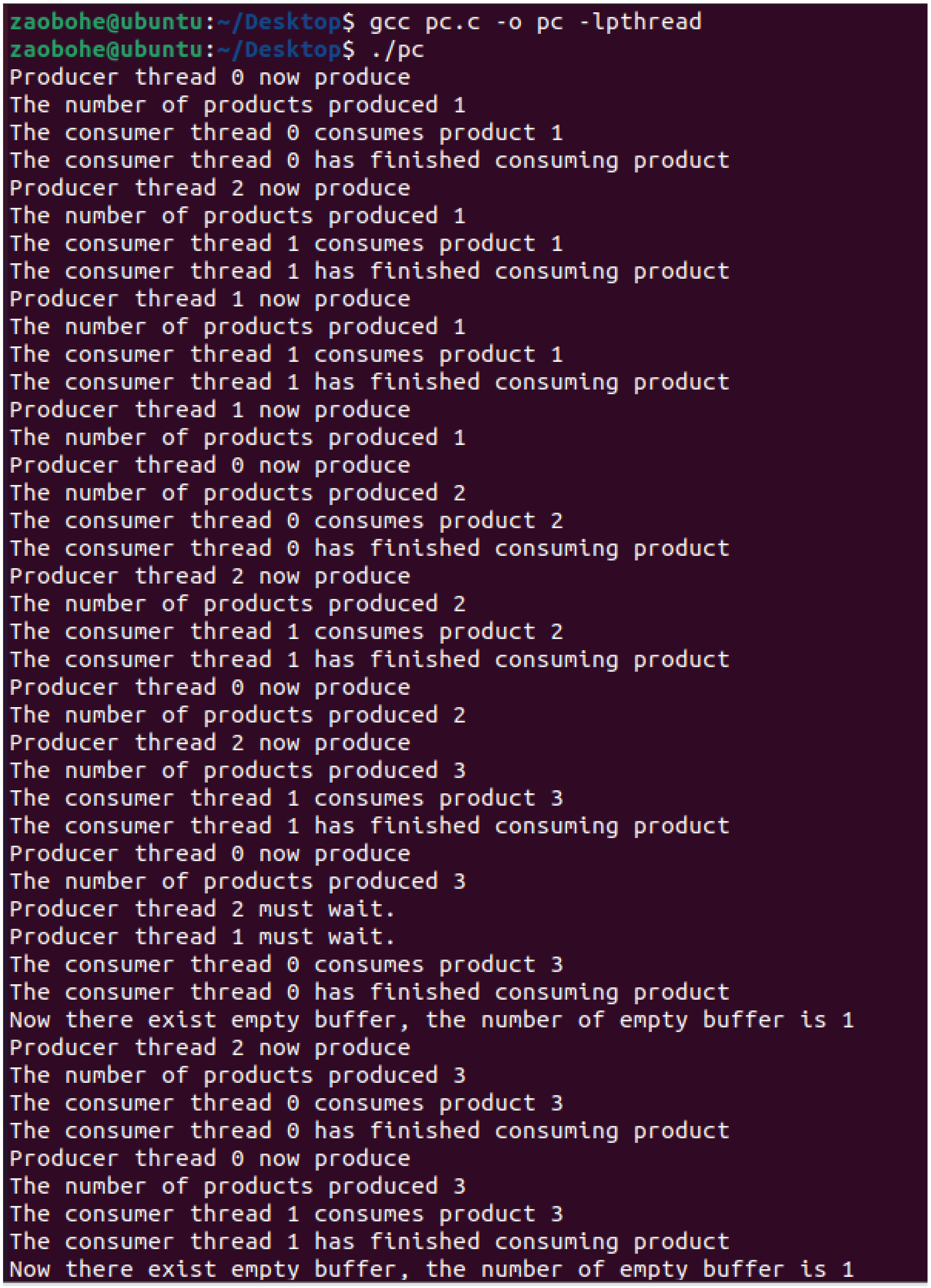
```
pthread_mutex_lock(&mutex);
//设置条件为真
pthread_cond_signal(cond);
pthread_mutex_unlock(&mutex);
```

8. **What should you do:**

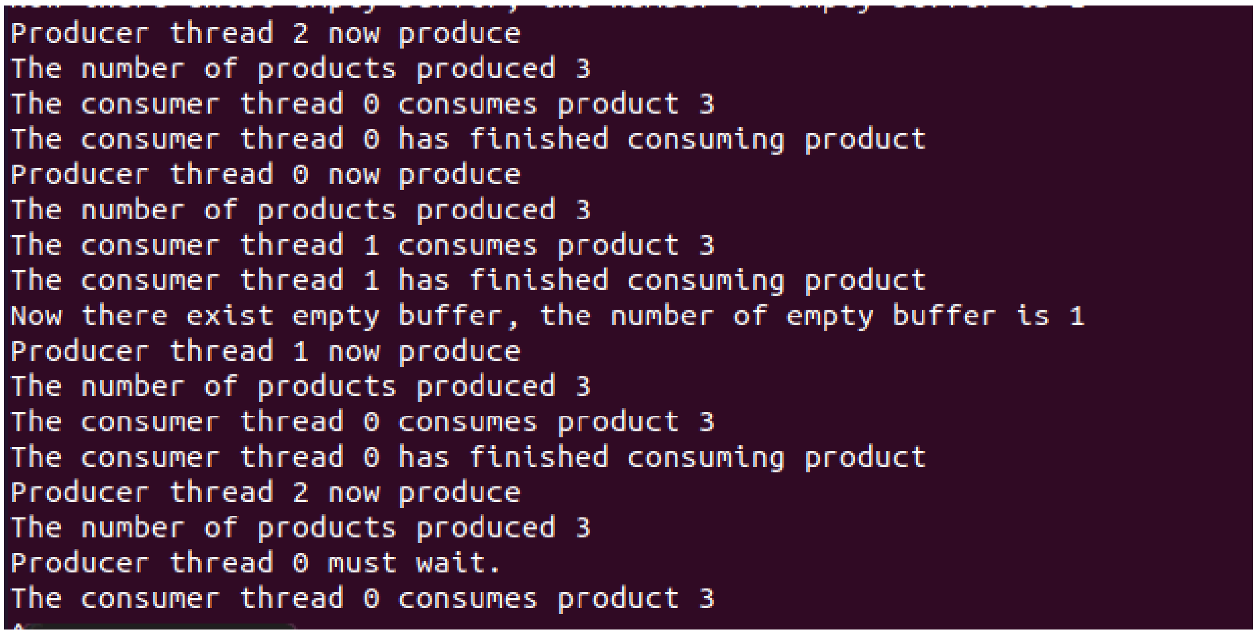
   Write code to solve the producer-consumer synchronization problem.

   Partial of the code have been given in **pc.c**.

   The expected execution results:



```
zaobohe@ubuntu:~/Desktop$ gcc pc.c -o pc -lpthread
zaobohe@ubuntu:~/Desktop$ ./pc
Producer thread 0 now produce
The number of products produced 1
The consumer thread 0 consumes product 1
The consumer thread 0 has finished consuming product
Producer thread 2 now produce
The number of products produced 1
The consumer thread 1 consumes product 1
The consumer thread 1 has finished consuming product
Producer thread 1 now produce
The number of products produced 1
The consumer thread 1 consumes product 1
The consumer thread 1 has finished consuming product
Producer thread 1 now produce
The number of products produced 1
Producer thread 0 now produce
The number of products produced 2
The consumer thread 0 consumes product 2
The consumer thread 0 has finished consuming product
Producer thread 2 now produce
The number of products produced 2
The consumer thread 1 consumes product 2
The consumer thread 1 has finished consuming product
Producer thread 0 now produce
The number of products produced 2
Producer thread 2 now produce
The number of products produced 3
The consumer thread 1 consumes product 3
The consumer thread 1 has finished consuming product
Producer thread 0 now produce
The number of products produced 3
Producer thread 2 must wait.
Producer thread 1 must wait.
The consumer thread 0 consumes product 3
The consumer thread 0 has finished consuming product
Now there exist empty buffer, the number of empty buffer is 1
Producer thread 2 now produce
The number of products produced 3
The consumer thread 0 consumes product 3
The consumer thread 0 has finished consuming product
Producer thread 0 now produce
The number of products produced 3
The consumer thread 1 consumes product 3
The consumer thread 1 has finished consuming product
Now there exist empty buffer, the number of empty buffer is 1
```



```
Producer thread 2 now produce
The number of products produced 3
The consumer thread 0 consumes product 3
The consumer thread 0 has finished consuming product
Producer thread 0 now produce
The number of products produced 3
The consumer thread 1 consumes product 3
The consumer thread 1 has finished consuming product
Now there exist empty buffer, the number of empty buffer is 1
Producer thread 1 now produce
The number of products produced 3
The consumer thread 0 consumes product 3
The consumer thread 0 has finished consuming product
Producer thread 2 now produce
The number of products produced 3
Producer thread 0 must wait.
The consumer thread 0 consumes product 3
```

9. What should you submit:

1) **Please submit a zip file which includes the C code of pc.c:**
2) **Your zip file should be named as: "姓名_lab5"**
3) **Please submit your zip file by sending email to me as attachment: my email address is: hzb564@jnu.edu.cn**