

1 实验一 进程的建立

1.1 实验目的

学会通过基本的 Windows 或者 Linux 进程控制函数，由父进程创建子进程，并实现父子进程协同工作。

1.2 实验软硬件环境

Linux version 5.13.0-40-generic (buildd@ubuntu)
gcc (Ubuntu 9.4.0-1ubuntu1 20.04.1) 9.4.0,GNU ld (GNU Binutils for Ubuntu) 2.34

1.3 实验内容

创建两个进程，让子进程读取一个文件，父进程等待子进程读取完文件后继续执行，实现进程协同工作。进程协同工作就是协调好两个进程，使之安排好先后次序并以此执行，可以用等待函数来实现这一点。当需要等待子进程运行结束时，可在父进程中调用等待函数。

1.4 实验程序及分析

```
1  #include <unistd.h>
2  #include <sys/types.h>
3  #include <sys/wait.h>
4  #include <stdlib.h>
5  #include <stdio.h>
6
7  int main() {
8      pid_t pid = fork();
9      if (pid < 0) {
10         fprintf(stderr, "Fork error %d\n", pid);
11         exit(pid);
12     }
13     else if (pid == 0) {
14         printf("Child process started! PID = %d\n", getpid());
15         freopen("./data", "r", stdin);
16         puts("Child read file:");
17         char buf[205];
18         int n;
19         while ((n = read(0, buf, 200)) > 0)
20             write(1, buf, n);
21         puts("Child process will exit in 2 seconds!");
22         sleep(2);
23     }
24     else {
```

```

25     printf("Father process started! PID = %d\n", getpid());
26     puts("Wait for child process...");
27     int status;
28     waitpid(pid, &status, 0);
29     puts("Child process exited!");
30     puts("Father process will exit in 2 seconds!");
31     sleep(2);
32 }
33 return 0;
34 }

```

使用 `fork()` 函数创建子进程。

若 `fork()` 函数返回值等于 0，则说明此进程为子进程。子进程读取文件并输出到屏幕，然后退出。

若 `fork()` 函数返回值大于 0，则说明此进程为父进程，返回值为子进程 PID。父进程中使用 `waitpid()` 函数挂起，等待子进程运行结束。

1.5 实验截图

```

~/tmp/lib1
> touch data

~/tmp/lib1
> echo 123456 >> data

~/tmp/lib1
> echo abcdef >> data

~/tmp/lib1
> gcc lib1.c -o lib1

~/tmp/lib1
> ./lib1
Father process started! PID = 9368
Wait for child process...
Child process started! PID = 9369
Child read file:
123456
abcdef
Child process will exit in 2 seconds!
Child process exited!
Father process will exit in 2 seconds

~/tmp/lib1 took 4s
>

```

Figure 1: 实验一 进程的建立

1.6 实验心得体会

本次实验使用了 `fork()` 函数创建子进程，使用了 `waitpid()` 函数挂起父进程，达到父子进程协同工作的目的。通过实验，理解了进程的创建过程。

2 实验二 线程共享进程数据

2.1 实验目的

了解线程与进程之间的数据共享关系。创建一个线程，在线程中更改进程中的数据。

2.2 实验软硬件环境

Linux version 5.13.0-40-generic (buldd@ubuntu)
gcc (Ubuntu 9.4.0-1ubuntu1 20.04.1) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34

2.3 实验内容

在进程中定义全局共享数据，在线程中直接引用该数据进行更改并输出该数据。

2.4 实验程序及分析

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <pthread.h>
4  #include <string.h>
5  #include <stdlib.h>
6
7  char shared_data[100];
8
9  void *thread_fun(void *arg) {
10     printf("Thread started! Thread ID = %lu\n", pthread_self());
11     strcpy(shared_data, "thread");
12     puts("Shared data changed in thread!");
13     puts("Thread ended!");
14 }
15
16 int main() {
17     printf("Main process started! Process PID = %u\n", getpid());
18     strcpy(shared_data, "main");
19     printf("Shared data is '%s' now\n", shared_data);
20     pthread_t tid;
21     unsigned int err_code = pthread_create(&tid, NULL, thread_fun,
    ↪     NULL);
```

```

22     if (err_code) {
23         fprintf(stderr, "Thread create error: %u\n", err_code);
24         exit(err_code);
25     }
26     sleep(1);
27     printf("Shared data is '%s' now\n", shared_data);
28     puts("Main process ended!");
29     return 0;
30 }

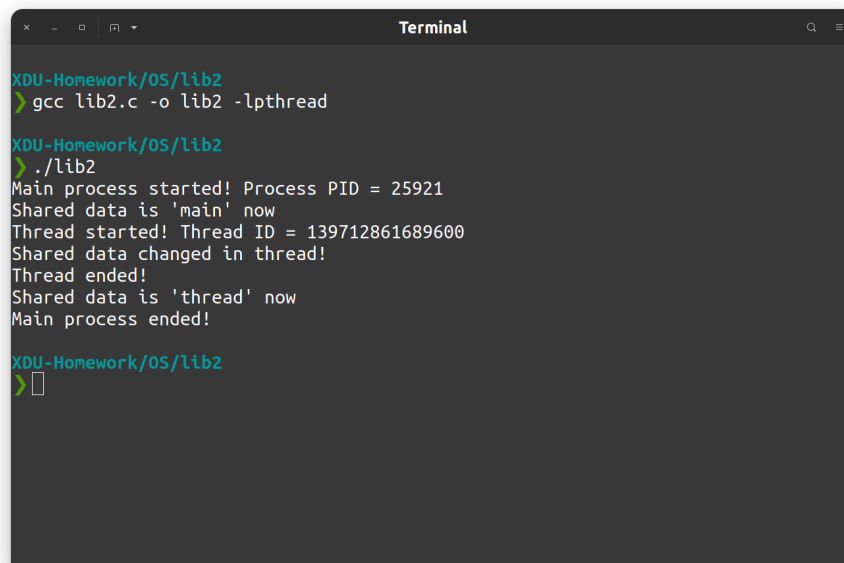
```

全局变量被进程和其子线程共享。故全局变量可被线程直接修改。

使用 `pthread_create()` 函数创建线程。其中第三个参数是函数指针，表示线程执行函数。

在进程中初始化全局共享变量并输出，然后在线程中修改全局共享变量并输出，最后等待线程函数执行完毕后，在进程输出全局共享变量。可以观察到线程成功修改了全局共享变量。

2.5 实验截图



```

XDU-Homework/OS/lib2
> gcc lib2.c -o lib2 -lpthread

XDU-Homework/OS/lib2
> ./lib2
Main process started! Process PID = 25921
Shared data is 'main' now
Thread started! Thread ID = 139712861689600
Shared data changed in thread!
Thread ended!
Shared data is 'thread' now
Main process ended!

XDU-Homework/OS/lib2
> 

```

Figure 2: 实验二 线程共享进程数据

2.6 实验心得体会

本次实验使用了 `pthread_create()` 函数创建线程。并通过定义全局变量在进程和各个线程中共享，以达到线程通信的目的。通过本次实验，理解了如何定义全局共享变量，如何实现进程与线程之间的简单通信。

3 实验三 信号通信

3.1 实验目的

利用信号通信机制在父子进程及兄弟进程间进行通信。

3.2 实验软硬件环境

Linux version 5.13.0-40-generic (buildd@ubuntu)
gcc (Ubuntu 9.4.0-1ubuntu1 20.04.1) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34

3.3 实验内容

父进程创建一个有名事件，由子进程发送事件信号，父进程获取事件信号后进行相应的处理。

3.4 实验程序及分析

阻塞型通信

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <signal.h>
4  #include <stdlib.h>
5  #include <sys/wait.h>
6
7  void handler(int sig) {
8      pid_t pid;
9      int status;
10     while (pid = waitpid(-1, &status, WNOHANG) > 0) {
11         printf("Child process %d died: %d\n", pid,
12             ↪ WEXITSTATUS(status));
13         printf("Parent process received SIGCHLD signal!");
14     }
15 }
16
17 int main() {
18     pid_t pid = fork();
19     if (pid == 0) {
20         printf("Child process started! PID = %u\n", getpid());
```

```

20     sleep(1);
21     printf("After sleep one second, child process PID = %u\n",
        ↪ getpid());
22     exit(0);
23 }
24 else if (pid > 0) {
25     printf("Parent process started! PID = %u\n", getpid());
26     signal(SIGCHLD, handler);
27     sleep(2);
28 }
29 else {
30     fprintf(stderr, "Fork error: %d\n", pid);
31     exit(pid);
32 }
33 return 0;
34 }

```

一个进程终止或者停止时，SIGCHLD 信号将被发送给其父进程。

创建一个子进程，打印子进程 PID。子进程运行一段时间后退出。

父进程调用信号处理函数 signal()，捕获信号 SIGCHLD 并在 handler() 函数中处理。在 handler() 函数中输出退出的子进程的 PID。可以观察到，父进程成功捕获子进程的终止信号。

非阻塞型通信

```

1  #include <stdio.h>
2  #include <unistd.h>
3  #include <signal.h>
4  #include <stdlib.h>
5
6  void handler(int sig) {
7      puts("Received SIGINT signal!");
8  }
9
10 int main() {
11     pid_t pid = fork();
12     if (pid == 0) {
13         printf("Child process started! PID = %u\n", getpid());
14         sleep(1);
15         printf("After sleep one second, child process PID = %u\n",
            ↪ getpid());
16         sleep(1);
17         printf("After sleep two seconds, child process PID = %u\n",
            ↪ getpid());
18         exit(0);
19     }

```

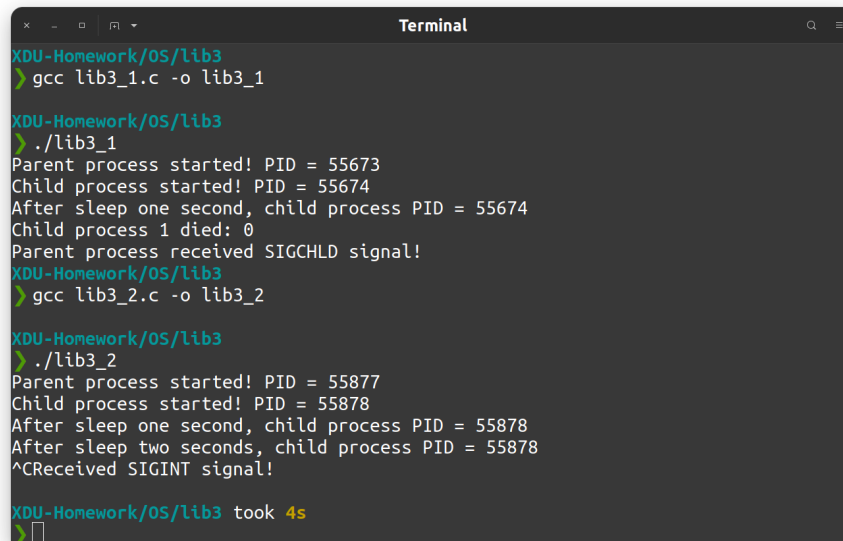
```

20     else if (pid > 0) {
21         printf("Parent process started! PID = %u\n", getpid());
22         signal(SIGINT, handler);
23         pause();
24     }
25     else {
26         fprintf(stderr, "Fork error: %d\n", pid);
27         exit(pid);
28     }
29     return 0;
30 }

```

在 Terminal 中按 Ctrl + c 可发送中断信号 SIGINT。
父进程调用信号处理函数 signal(), 捕获信号 SIGINT 并在 handler() 函数中打印信息。

3.5 实验截图



```

XDU-Homework/OS/lib3
> gcc lib3_1.c -o lib3_1

XDU-Homework/OS/lib3
> ./lib3_1
Parent process started! PID = 55673
Child process started! PID = 55674
After sleep one second, child process PID = 55674
Child process 1 died: 0
Parent process received SIGCHLD signal!
XDU-Homework/OS/lib3
> gcc lib3_2.c -o lib3_2

XDU-Homework/OS/lib3
> ./lib3_2
Parent process started! PID = 55877
Child process started! PID = 55878
After sleep one second, child process PID = 55878
After sleep two seconds, child process PID = 55878
^CReceived SIGINT signal!

XDU-Homework/OS/lib3 took 4s
>

```

Figure 3: 实验三 信号通信

3.6 实验心得体会

本次实验使用了 signal() 函数捕获信号并做相应的处理。通过本次实验，理解了如何在进程之间进行信号通信。

4 实验四 匿名管道通信

4.1 实验目的

学习使用匿名管道在两个进程间建立通信。

4.2 实验软硬件环境

Linux version 5.15.0-30-generic (buldd@lgw01-amd64-058)
gcc (Ubuntu 11.2.0-19ubuntu1) 11.2.0, GNU ld (GNU Binutils for Ubuntu)
2.38

4.3 实验内容

分别建立名为 Parent 的单文档应用程序和 Child 的单文档应用程序作为父子进程，由父进程创建一个匿名管道，实现父子进程向匿名管道写入和读取数据。

4.4 实验程序及分析

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <stdlib.h>
5  #include <string.h>
6
7  #define READ_END    0
8  #define WRITE_END   1
9  #define BUFFER_SIZE 100
10
11 int main() {
12     int fd[2];
13     if (pipe(fd) == -1) {
14         puts("Create pipe failed");
15         exit(-1);
16     }
17     pid_t pid = fork();
18     if (pid < 0) {
19         puts("Fork failed");
20         exit(pid);
21     }
22     else if (pid > 0) {
23         printf("Parent process started! PID = %u\n", getpid());
24         close(fd[READ_END]);
25         char write_msg[BUFFER_SIZE] = "Hello, pipe!";
26         write(fd[WRITE_END], write_msg, strlen(write_msg) + 1);
27         printf("Write data: %s\n", write_msg);
```



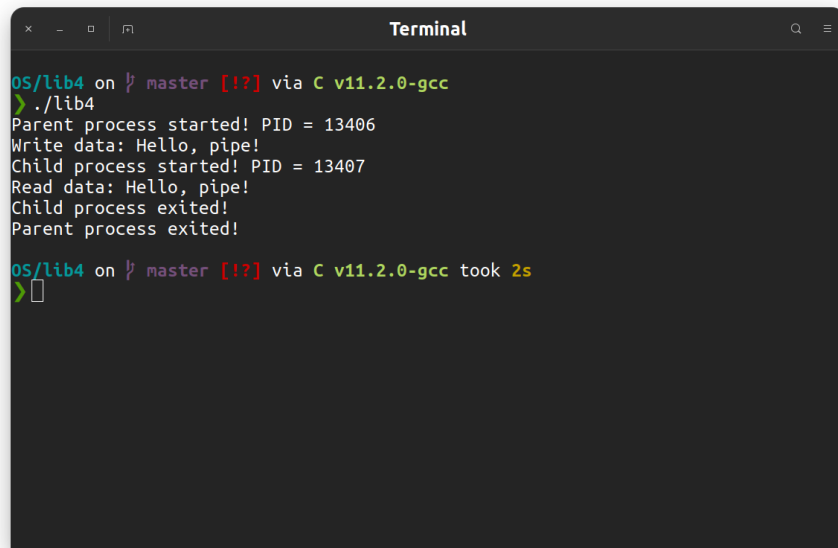
```

28         close(fd[WRITE_END]);
29         sleep(2);
30         puts("Parent process exited!");
31     }
32     else {
33         printf("Child process started! PID = %u\n", getpid());
34         close(fd[WRITE_END]);
35         sleep(1);
36         char read_msg[BUFFER_SIZE];
37         read(fd[READ_END], read_msg, BUFFER_SIZE);
38         printf("Read data: %s\n", read_msg);
39         close(fd[READ_END]);
40         puts("Child process exited!");
41     }
42     return 0;
43 }

```

创建一个匿名管道，父子进程共享该管道。父进程向管道中写入数据，子进程从管道中读数据并输出。

4.5 实验截图



```

OS/lib4 on  master [!?] via C v11.2.0-gcc
> ./lib4
Parent process started! PID = 13406
Write data: Hello, pipe!
Child process started! PID = 13407
Read data: Hello, pipe!
Child process exited!
Parent process exited!

OS/lib4 on  master [!?] via C v11.2.0-gcc took 2s
>

```

Figure 4: 实验四 匿名管道通信

4.6 实验心得体会

本次实验使用了 `pipe()` 函数创建匿名管道。通过父子进程共享数据实现管道通信。通过本次实验，理解了如何使用匿名管道进行通信。

5 实验五 命名管道通信

5.1 实验目的

学习使用命名管道在多进程间建立通信。

5.2 实验软硬件环境

Linux version 5.15.0-30-generic (buildd@lgw01-amd64-058)
gcc (Ubuntu 11.2.0-19ubuntu1) 11.2.0, GNU ld (GNU Binutils for Ubuntu)
2.38

5.3 实验内容

建立父子进程，由父进程创建一个命名管道，由子进程向命名管道写入数据，由父进程从命名管道读取数据。

5.4 实验程序及分析

```
1  #include <stdio.h>
2  #include <fcntl.h>
3  #include <sys/types.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6  #include <sys/stat.h>
7  #include <string.h>
8
9  #define FIFO_SERVER "./fifo_server"
10 #define BUFFER_SIZE 100
11
12 int main() {
13     if (mkfifo(FIFO_SERVER, 0664) == -1) {
14         puts("Create fifo failed");
15         exit(-1);
16     }
17     pid_t pid = fork();
18     if (pid < 0) {
19         puts("Fork failed");
20         exit(pid);
21     }
22     if (pid > 0) {
```

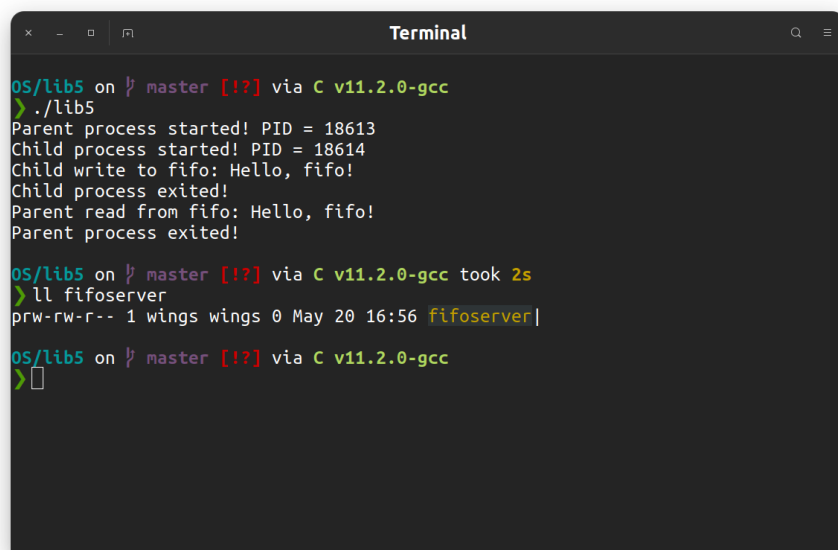
```

23     printf("Parent process started! PID = %u\n", getpid());
24     sleep(1);
25     int fd = open(FIFO_SERVER, O_RDONLY);
26     char read_msg[BUFFER_SIZE];
27     read(fd, read_msg, BUFFER_SIZE);
28     printf("Parent read from fifo: %s\n", read_msg);
29     close(fd);
30     sleep(1);
31     puts("Parent process exited!");
32 }
33 else {
34     printf("Child process started! PID = %u\n", getpid());
35     int fd = open(FIFO_SERVER, O_WRONLY);
36     char write_msg[BUFFER_SIZE] = "Hello, fifo!";
37     write(fd, write_msg, strlen(write_msg) + 1);
38     printf("Child write to fifo: %s\n", write_msg);
39     close(fd);
40     puts("Child process exited!");
41 }
42 return 0;
43 }

```

在当前目录下创建一个命名管道。子进程以写的方式打开命名管道，并向其中写入数据。父进程以读的方式打开命名管道，并从其中读取数据并输出。同时还可以观察到在当前目录下创建的命名管道文件。

5.5 实验截图



```
OS/lib5 on  master [!?] via C v11.2.0-gcc
> ./lib5
Parent process started! PID = 18613
Child process started! PID = 18614
Child write to fifo: Hello, fifo!
Child process exited!
Parent read from fifo: Hello, fifo!
Parent process exited!

OS/lib5 on  master [!?] via C v11.2.0-gcc took 2s
> ll fifoserver
prw-rw-r-- 1 wings wings 0 May 20 16:56 fifoserver|

OS/lib5 on  master [!?] via C v11.2.0-gcc
>
```

Figure 5: 实验五 命名管道通信

5.6 实验心得体会

本次实验使用了 `mkfifo()` 函数创建命名管道，利用命名管道实现进程间的通信。通过本次实验，理解了如何使用命名管道进行通信。

6 实验六 信号量实现进程同步

6.1 实验目的

进程同步是操作系统多进程/多线程并发执行的关键之一，进程同步是并发进程为了完成共同任务采用某个条件来协调他们的活动，这是进程之间发生的一种直接制约关系。本次试验是利用信号量进行进程同步。

6.2 实验软硬件环境

Linux version 5.15.0-30-generic (buldd@lgw01-amd64-058)
gcc (Ubuntu 11.2.0-19ubuntu1) 11.2.0, GNU ld (GNU Binutils for Ubuntu)
2.38

6.3 实验内容

- 生产者进程生产产品，消费者进程消费产品。
- 当生产者进程生产产品时，如果没有空缓冲区可用，那么生产者进程必须等待消费者进程释放出一个缓冲区。
- 当消费者进程消费产品时，如果缓冲区中没有产品，那么消费者进程将被阻塞，直到新的产品被生产出来。

6.4 实验程序及分析

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <semaphore.h>
5  #include <stdbool.h>
6  #include <unistd.h>
7
8  sem_t mutex, empty, full;
9  int item_buf[100], top = 0;
10
11 void *producer(void *arg) {
12     int id = 0;
13     puts("Producer started!");
14     while (true) {
15         sleep(2);
16         sem_wait(&empty);
17         sem_wait(&mutex);
18         item_buf[top++] = id;
19         printf("Produced %d\n", id++);
20         sem_post(&mutex);
21         sem_post(&full);
22     }
23 }
24
25 void *consumer_a(void *arg) {
26     sleep(15);
27     puts("Consumer A started!");
28     while (true) {
29         sem_wait(&full);
30         sem_wait(&mutex);
31         int id = item_buf[--top];
32         printf("Consumer A consumed %d\n", id);
33         sem_post(&mutex);
34         sem_post(&empty);
35         sleep(3);
36     }
37 }
```

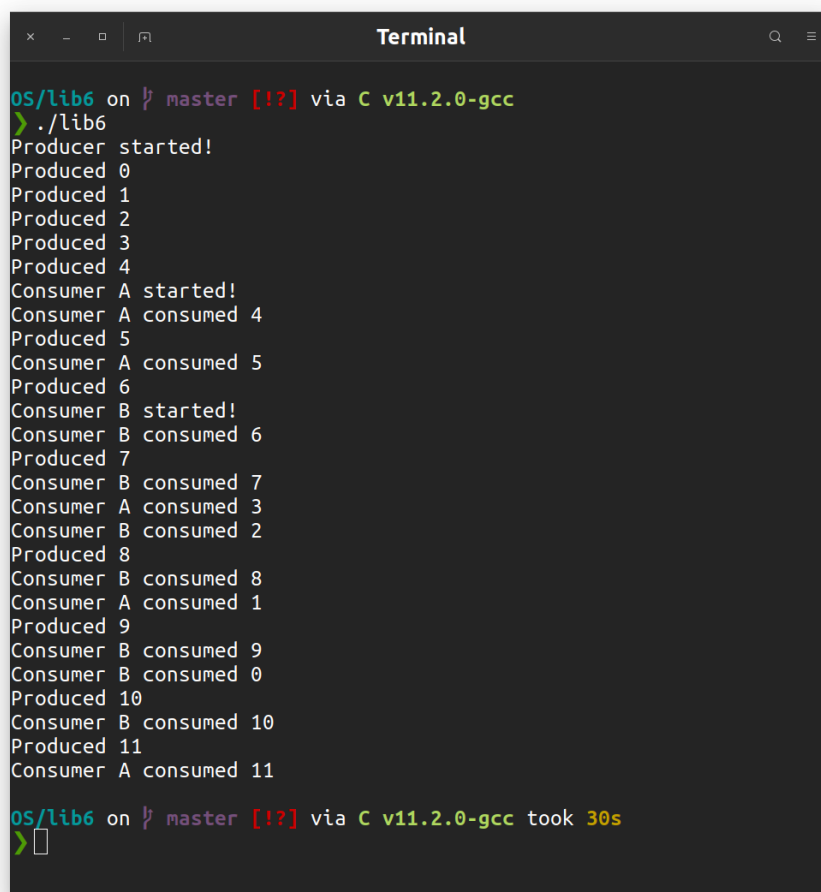
```

36     }
37 }
38
39 void *consumer_b(void *arg) {
40     sleep(20);
41     puts("Consumer B started!");
42     while (true) {
43         sem_wait(&full);
44         sem_wait(&mutex);
45         int id = item_buf[--top];
46         printf("Consumer B consumed %d\n", id);
47         sem_post(&mutex);
48         sem_post(&empty);
49         sleep(1);
50     }
51 }
52
53 int main() {
54     sem_init(&mutex, 0, 1);
55     sem_init(&empty, 0, 5);
56     sem_init(&full, 0, 0);
57     pthread_t tid_ca, tid_cb, tid_p;
58     pthread_create(&tid_p, NULL, producer, NULL);
59     pthread_create(&tid_ca, NULL, consumer_a, NULL);
60     pthread_create(&tid_cb, NULL, consumer_b, NULL);
61     sleep(30);
62     sem_destroy(&mutex);
63     sem_destroy(&empty);
64     sem_destroy(&full);
65     return 0;
66 }

```

生产者先生产，当信号量 empty 为 0 时阻塞。待消费者开始消费，使得 empty 大于 0 后，生产者才可以继续生产。消费者将物品消费完后，信号量 full 为 0 时阻塞。等待生产者生产后，再唤醒并消费。同时，使用信号量 mutex 实现互斥锁，使得对生产者和消费者无法同时进入临界区对缓冲队列进行修改。

6.5 实验截图



```
OS/lib6 on  master [!?] via C v11.2.0-gcc
> ./lib6
Producer started!
Produced 0
Produced 1
Produced 2
Produced 3
Produced 4
Consumer A started!
Consumer A consumed 4
Produced 5
Consumer A consumed 5
Produced 6
Consumer B started!
Consumer B consumed 6
Produced 7
Consumer B consumed 7
Consumer A consumed 3
Consumer B consumed 2
Produced 8
Consumer B consumed 8
Consumer A consumed 1
Produced 9
Consumer B consumed 9
Consumer B consumed 0
Produced 10
Consumer B consumed 10
Produced 11
Consumer A consumed 11

OS/lib6 on  master [!?] via C v11.2.0-gcc took 30s
>
```

Figure 6: 实验六 信号量实现进程同步

6.6 实验心得体会

本次实验使用了信号量进行生产者与消费者的同步。通过本次实验，理解了如何使用信号量来进行同步。