计算机操作系统原理

课程设计

题　　目 System V IPC 进程通信

学生姓名 _____

专业班级 计算机科学与技术　班

完成时间 _____

指导教师 _____

青岛大学　计算机学院

2019 年 7 月

设计目的

1、理解 System V IPC 通信机制工作原理；

2、掌握和使用共享主存实现进程通信；

3、掌握和使用消息队列实现进程通信；

4、掌握和使用信号量实现进程同步。

Linux 进程间通信（IPC，Inter-Process Communication）

Linux 进程间通信——使用共享内存

一、什么是共享内存

顾名思义，共享内存就是允许两个不相关的进程访问同一个逻辑内存。共享内存是在两个正在运行的进程之间共享和传递数据的一种非常有效的方式。不同进程之间共享的内存通常安排为同一段物理内存。进程可以将同一段共享内存连接到它们自己的地址空间中，所有进程都可以访问共享内存中的地址，就好像它们是由用 C 语言函数 malloc 分配的内存一样。而如果某个进程向共享内存写入数据，所做的改动将立即影响到可以访问同一段共享内存的任何其他进程。

特别提醒：共享内存并未提供同步机制，也就是说，在第一个进程结束对共享内存的写操作之前，并无自动机制可以阻止第二个进程开始对它进行读取。所以我们通常需要用其他的机制来同步对共享内存的访问，例如前面说到的信号量。

二、共享内存的使得

与信号量一样，在 Linux 中也提供了一组函数接口用于使用共享内存，而且使用共享共存的接口还与信号量的非常相似，而且比使用信号量的接口来得简单。它们声明在头文件 sys/shm.h 中。

1、shmget 函数

该函数用来创建共享内存，它的原型为：

```
int shmget(key_t key, size_t size, int shmflg);
```

- 第一个参数，与信号量的 semget 函数一样，程序需要提供一个参数 key（非 0 整数），它有效地为共享内存段命名，shmget 函数成功时返回一个与 key 相关的共享内存标识符（非负整数），用于后续的共享内存函数。调用失败返回-1.不相关的进程可以通过该函数的返回值访问同一共享内存，它代表程序可能要使用的某个资源，程序对所有共享内存的访问都是间接的，程序先通过调用 shmget 函数并提供一个键，再由系统生成一个相应的共享内存标识符（shmget 函数的返回值），只有 shmget 函数才直接使用信号量键，所有其他的信号量函数使用由 semget 函数返回的信号量标识符。
- 第二个参数，size 以字节为单位指定需要共享的内存容量.
- 第三个参数，shmflg 是权限标志，它的作用与 open 函数的 mode 参数一样，如果要想在 key 标识的共享内存不存在时，创建它的话，可以与 IPC_CREAT 做或操作。共享内存的权限标志与文件的读写权限一样，举例来说，0644,它表示允许一个进程创建的共享内存被内存创建者所拥有的进程向共享内存读取和写入数据，同时其他用户创建的进程只能读取共享内存。

2、shmat 函数

第一次创建完共享内存时，它还不能被任何进程访问，shmat 函数的作用就是用来启动对该共享内存的访问，并把共享内存连接到当前进程的地址空间。它的原型如下：

```
void *shmat(int shm_id, const void *shm_addr, int shmflg);
```

- 第一个参数，shm_id 是由 shmget 函数返回的共享内存标识。
- 第二个参数，shm_addr 指定共享内存连接到当前进程中的地址位置，通常为空，表示让系统来选择共享内存的地址。
- 第三个参数，shm_flg 是一组标志位，通常为 0。

调用成功时返回一个指向共享内存第一个字节的指针，如果调用失败返回-1.

3、shmdt 函数
该函数用于将共享内存从当前进程中分离。注意，将共享内存分离并不是删除它，只是使该共享内存对当前进程不再可用。它的原型如下：

```
int shmdt(const void *shmaddr);
```

参数 shmaddr 是 shmat 函数返回的地址指针，调用成功时返回 0，失败时返回-1.

4、shmctl 函数
与信号量的 semctl 函数一样，用来控制共享内存，它的原型如下：

```
int shmctl(int shm_id, int command, struct shmid_ds *buf);
```

- 第一个参数，shm_id 是 shmget 函数返回的共享内存标识符。
- 第二个参数，command 是要采取的操作，它可以取下面的三个值 ：

IPC_STAT：把 shmid_ds 结构中的数据设置为共享内存的当前关联值，即用共享内存的当前关联值覆盖 shmid_ds 的值。

IPC_SET：如果进程有足够的权限，就把共享内存的当前关联值设置为 shmid_ds 结构中给出的值

IPC_RMID：删除共享内存段

- 第三个参数，buf 是一个结构指针，它指向共享内存模式和访问权限的结构。

shmid_ds 结构至少包括以下成员：

```
struct shmid_ds
{
    uid_t    shm_perm.uid;
    uid_t    shm_perm.gid;
    mode_t    shm_perm.mode;
};
```

三、使用共享内存进行进程间通信

下面就以两个不相关的进程来说明进程间如何通过共享内存来进行通信。其中一个文件 shmread.c 创建共享内存，并读取其中的信息，另一个文件 shmwrite.c 向共享内存中写入数据。为了方便操作和数据结构的统一，为这两个文件定义了相同的数据结构 shared_use_st。结构 shared_use_st 中的 written 作为一个可读或可写的标志，非 0：表示可读，0 表示可写，

text 则是内存中的文件。

源文件 shmread.c 的源代码如下：

```
 #include <unistd.h>
 #include <stdlib.h>
 #include <string.h>
#include <stdio.h>
#include <sys/shm.h>

#define TEXT_SZ 2048
struct shared_use_st
{
    int written;//作为一个标志，非 0：表示可读，0 表示可写
    char text[TEXT_SZ];//记录写入和读取的文本
};

int main()
{
    int running = 1;//程序是否继续运行的标志
    void *shm = NULL;//分配的共享内存的原始首地址
    struct shared_use_st *shared;//指向 shm
    int shmid;//共享内存标识符
    //创建共享内存
    shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666|IPC_CREAT);
    if(shmid == -1)
    {
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }
    //将共享内存连接到当前进程的地址空间
    shm = shmat(shmid, 0, 0);
    if(shm == (void*)-1)
    {
        fprintf(stderr, "shmat failed\n");
        exit(EXIT_FAILURE);
    }
    printf("\nMemory attached at %X\n", (int)shm);
    //设置共享内存
    shared = (struct shared_use_st*)shm;
    shared->written = 0;
    while(running)//读取共享内存中的数据
    {
        //没有进程向共享内存定数据有数据可读取
        if(shared->written != 0)
```

```
            {
                printf("You wrote: %s", shared->text);
                sleep(rand() % 3);
                //读取完数据，设置 written 使共享内存段可写
                shared->written = 0;
                //输入了 end，退出循环（程序）
                if(strncmp(shared->text, "end", 3) == 0)
                    running = 0;
            }
            else//有其他进程在写数据，不能读取数据
                sleep(1);
        }
        //把共享内存从当前进程中分离
        if(shmdt(shm) == -1)
        {
            fprintf(stderr, "shmdt failed\n");
            exit(EXIT_FAILURE);
        }
        //删除共享内存
        if(shmctl(shmid, IPC_RMID, 0) == -1)
        {
            fprintf(stderr, "shmctl(IPC_RMID) failed\n");
            exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
    }
```

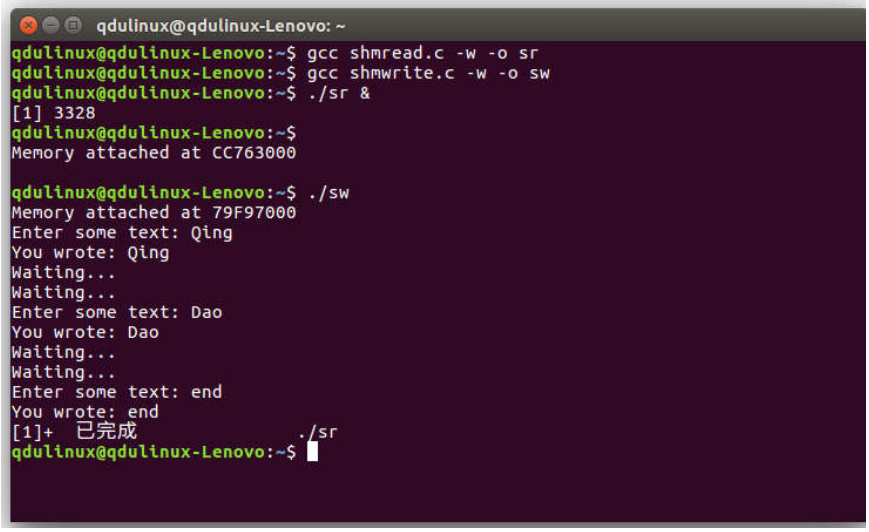
源文件 shmwrite.c 的源代码如下：

```
    #include <unistd.h>
    #include <stdlib.h>
    #include <stdio.h>
    #include <string.h>
    #include <sys/shm.h>

    #define TEXT_SZ 2048
    struct shared_use_st
    {
        int written;//作为一个标志，非 0：表示可读，0 表示可写
        char text[TEXT_SZ];//记录写入和读取的文本
    };

    int main()
    {
        int running = 1;
```

```
void *shm = NULL;
struct shared_use_st *shared = NULL;
char buffer[BUFSIZ + 1];//用于保存输入的文本
int shmid;
//创建共享内存
shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666|IPC_CREAT);
if(shmid == -1)
{
    fprintf(stderr, "shmget failed\n");
    exit(EXIT_FAILURE);
}
//将共享内存连接到当前进程的地址空间
shm = shmat(shmid, (void*)0, 0);
if(shm == (void*)-1)
{
    fprintf(stderr, "shmat failed\n");
    exit(EXIT_FAILURE);
}
printf("Memory attached at %X\n", (int)shm);
//设置共享内存
shared = (struct shared_use_st*)shm;
while(running)//向共享内存中写数据
{
    //数据还没有被读取，则等待数据被读取,不能向共享内存中写入文本
    while(shared->written == 1)
    {
        sleep(1);
        printf("Waiting...\n");
    }
    //向共享内存中写入数据
    printf("Enter some text: ");
    fgets(buffer, BUFSIZ, stdin);
    strncpy(shared->text, buffer, TEXT_SZ);
    //写完数据，设置 written 使共享内存段可读
    shared->written = 1;
    //输入了 end，退出循环（程序）
    if(strncmp(buffer, "end", 3) == 0)
        running = 0;
}
//把共享内存从当前进程中分离
if(shmdt(shm) == -1)
{
    fprintf(stderr, "shmdt failed\n");
    exit(EXIT_FAILURE);
```

```
        }
        sleep(2);
        exit(EXIT_SUCCESS);
    }
```

再来看看运行的结果：



分析：

1、程序 shmread 创建共享内存，然后将它连接到自己的地址空间。在共享内存的开始处使用了一个结构 struct_use_st。该结构中有个标志 written，当共享内存中有其他进程向它写入数据时，共享内存中的 written 被设置为 0，程序等待。当它不为 0 时，表示没有进程对共享内存写入数据，程序就从共享内存中读取数据并输出，然后重置设置共享内存中的 written 为 0，即让其可被 shmwrite 进程写入数据。

2、程序 shmwrite 取得共享内存并连接到自己的地址空间中。检查共享内存中的 written，是否为 0，若不是，表示共享内存中的数据还没有被完，则等待其他进程读取完成，并提示用户等待。若共享内存的 written 为 0，表示没有其他进程对共享内存进行读取，则提示用户输入文本，并再次设置共享内存中的 written 为 1，表示写完成，其他进程可对共享内存进行读操作。

四、关于前面的例子的安全性讨论

这个程序是不安全的，当有多个程序同时向共享内存中读写数据时，问题就会出现。可能你会认为，可以改变一下 written 的使用方式，例如，只有当 written 为 0 时进程才可以向共享内存写入数据，而当一个进程只有在 written 不为 0 时才能对其进行读取，同时把 written 进行加 1 操作，读取完后进行减 1 操作。这就有点像文件锁中的读写锁的功能。咋看之下，它似乎能行得通。但是这都不是原子操作，所以这种做法是行不能的。试想当 written 为 0 时，如果有两个进程同时访问共享内存，它们就会发现 written 为 0，于是两个进程都对其进行写操作，显然不行。当 written 为 1 时，有两个进程同时对共享内存进行读操作时也是如些，当这两个进程都读取完是，written 就变成了-1.

要想让程序安全地执行，就要有一种进程同步的进制，保证在进入临界区的操作是原子

操作。例如，可以使用信号量来进行进程的同步。因为信号量的操作都是原子性的。

五、使用共享内存的优缺点
1、优点：我们可以看到使用共享内存进行进程间的通信真的是非常方便，而且函数的接口也简单，数据的共享还使进程间的数据不用传送，而是直接访问内存，也加快了程序的效率。同时，它也不像匿名管道那样要求通信的进程有一定的父子关系。
2、缺点：共享内存没有提供同步的机制，这使得我们在使用共享内存进行进程间通信时，往往要借助其他的手段来进行进程间的同步工作。

Linux 进程间通信——使用信号量

一、什么是信号量
为了防止出现因多个程序同时访问一个共享资源而引发的一系列问题，我们需要一种方法，它可以通过生成并使用令牌来授权，在任一时刻只能有一个执行线程访问代码的临界区域。临界区域是指执行数据更新的代码需要独占式地执行。而信号量就可以提供这样的一种访问机制，让一个临界区同一时间只有一个线程在访问它，也就是说信号量是用来调协进程对共享资源的访问的。

信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行等待（即 P(信号变量))和发送（即 V(信号变量))信息操作。最简单的信号量是只能取 0 和 1 的变量，这也是信号量最常见的一种形式，叫做二进制信号量。而可以取多个正整数的信号量被称为通用信号量。这里主要讨论二进制信号量。

二、信号量的工作原理
由于信号量只能进行两种操作等待和发送信号，即 P(sv)和 V(sv),他们的行为是这样的：
P(sv)：如果 sv 的值大于零，就给它减 1；如果它的值为零，就挂起该进程的执行
V(sv)：如果有其他进程因等待 sv 而被挂起，就让它恢复运行，如果没有进程因等待 sv 而挂起，就给它加 1.

举个例子，就是两个进程共享信号量 sv，一旦其中一个进程执行了 P(sv)操作，它将得到信号量，并可以进入临界区，使 sv 减 1。而第二个进程将被阻止进入临界区，因为当它试图执行 P(sv)时，sv 为 0，它会被挂起以等待第一个进程离开临界区域并执行 V(sv)释放信号量，这时第二个进程就可以恢复执行。

三、Linux 的信号量机制
Linux 提供了一组精心设计的信号量接口来对信号进行操作，它们不只是针对二进制信号量，下面将会对这些函数进行介绍，但请注意，这些函数都是用来对成组的信号量值进行操作的。它们声明在头文件 sys/sem.h 中。

1、semget 函数
它的作用是创建一个新信号量或打开一个已有信号量，原型为：
```
int semget(key_t key, int num_sems, int sem_flags);
```

第一个参数 key 是整数值（唯一非零），不相关的进程可以通过它访问一个信号量，它代表

程序可能要使用的某个资源，程序对所有信号量的访问都是间接的，程序先通过调用 semget 函数并提供一个键，再由系统生成一个相应的信号标识符（semget 函数的返回值），只有 semget 函数才直接使用信号量键，所有其他的信号量函数使用由 semget 函数返回的信号量标识符。如果多个程序使用相同的 key 值，key 将负责协调工作。
第二个参数 num_sems 指定需要的信号量数目，它的值几乎总是 1。
第三个参数 sem_flags 是一组标志，当想要当信号量不存在时创建一个新的信号量，可以和值 IPC_CREAT 做按位或操作。设置了 IPC_CREAT 标志后，即使给出的键是一个已有信号量的键，也不会产生错误。而 IPC_CREAT | IPC_EXCL 则可以创建一个新的，唯一的信号量，如果信号量已存在，返回一个错误。

semget 函数成功返回一个相应信号标识符（非零），失败返回-1.

2、semop 函数
它的作用是增加或者减小信号量的值，原型为：

```
int semop(int sem_id, struct sembuf *sem_opa, size_t num_sem_ops);
```

sem_id 是由 semget 返回的信号量标识符，sembuf 结构的定义如下：

```
struct sembuf{
    short sem_num;   //除非使用一组信号量，否则它为 0
    short sem_op; //信号量在一次操作中需要改变的数据，通常是两个数，一个是-1，即 P（等待）操作，
                  //一个是+1，即 V（发送信号）操作。
    short sem_flg;    //通常为 SEM_UNDO,使操作系统跟踪信号，
                  //并在进程没有释放该信号量而终止时，操作系统释放信号量
};
```

3、semctl 函数
该函数用来直接控制信号量信息，它的原型为：

```
int semctl(int sem_id, int sem_num, int command, ...);
```

如果有第四个参数，它通常是一个 union semum 结构，定义如下：

```
union semun{
    int val;
    struct semid_ds    *buf;
    unsigned short    *arry;
};
```

前两个参数与前面一个函数中的一样，command 通常是下面两个值中的其中一个
SETVAL：用来把信号量初始化为一个已知的值。p 这个值通过 union semun 中的 val 成员设置，其作用是在信号量第一次使用前对它进行设置。
IPC_RMID：用于删除一个已经无需继续使用的信号量标识符。

四、进程使用信号量通信

下面使用一个例子来说明进程间如何使用信号量来进行通信，这个例子是两个相同的程序同时向屏幕输出数据，我们可以看到如何使用信号量来使两个进程协调工作，使同一时间只有一个进程可以向屏幕输出数据。注意，如果程序是第一次被调用（为了区分，第一次调用程序时带一个要输出到屏幕中的字符作为一个参数），则需要调用 set_semvalue 函数初始化信号并将 message 字符设置为传递给程序的参数的第一个字符，同时第一个启动的进程还负责信号量的删除工作。如果不删除信号量，它将继续在系统中存在，即使程序已经退出，它可能在你下次运行此程序时引发问题，而且信号量是一种有限的资源。

在 main 函数中调用 semget 来创建一个信号量，该函数将返回一个信号量标识符，保存于全局变量 sem_id 中，然后以后的函数就使用这个标识符来访问信号量。

源文件为 seml.c，代码如下：

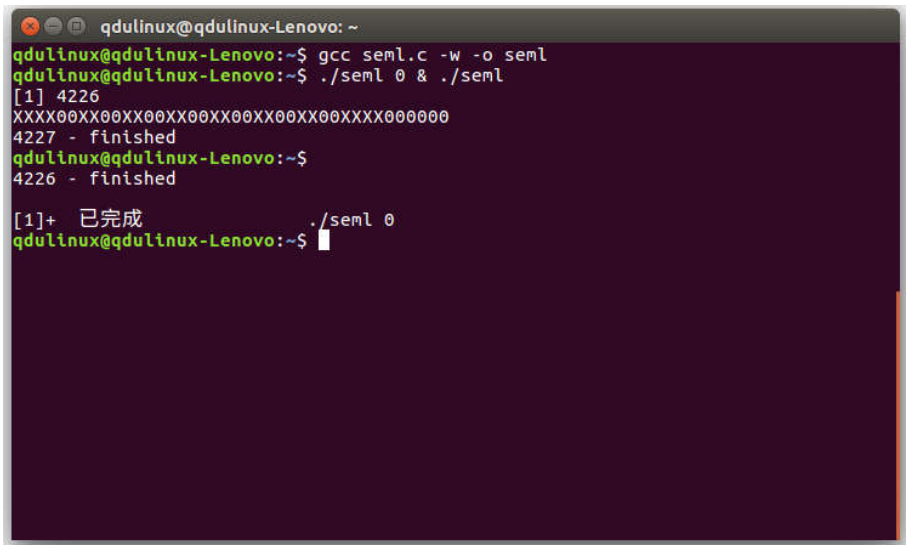
```
 #include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/sem.h>

union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short *arry;
};

static int sem_id = 0;

static int set_semvalue();
static void del_semvalue();
static int semaphore_p();
static int semaphore_v();

int main(int argc, char *argv[])
{
    char message = 'X';
    int i = 0;

    //创建信号量
    sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);
```

```
    if(argc > 1)
    {
        //程序第一次被调用，初始化信号量
        if(!set_semvalue())
        {
            fprintf(stderr, "Failed to initialize semaphore\n");
            exit(EXIT_FAILURE);
        }
        //设置要输出到屏幕中的信息，即其参数的第一个字符
        message = argv[1][0];
        sleep(2);
    }
    for(i = 0; i < 10; ++i)
    {
        //进入临界区
        if(!semaphore_p())
            exit(EXIT_FAILURE);
        //向屏幕中输出数据
        printf("%c", message);
        //清理缓冲区，然后休眠随机时间
        fflush(stdout);
        sleep(rand() % 3);
        //离开临界区前再一次向屏幕输出数据
        printf("%c", message);
        fflush(stdout);
        //离开临界区，休眠随机时间后继续循环
        if(!semaphore_v())
            exit(EXIT_FAILURE);
        sleep(rand() % 2);
    }

    sleep(10);
    printf("\n%d - finished\n", getpid());

    if(argc > 1)
    {
        //如果程序是第一次被调用，则在退出前删除信号量
        sleep(3);
        del_semvalue();
    }
    exit(EXIT_SUCCESS);
}
```

```
static int set_semvalue()
{
    //用于初始化信号量，在使用信号量前必须这样做
    union semun sem_union;

    sem_union.val = 1;
    if(semctl(sem_id, 0, SETVAL, sem_union) == -1)
        return 0;
    return 1;
}

static void del_semvalue()
{
    //删除信号量
    union semun sem_union;

    if(semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
        fprintf(stderr, "Failed to delete semaphore\n");
}

static int semaphore_p()
{
    //对信号量做减 1 操作，即等待 P（sv）
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = -1;//P()
    sem_b.sem_flg = SEM_UNDO;
    if(semop(sem_id, &sem_b, 1) == -1)
    {
        fprintf(stderr, "semaphore_p failed\n");
        return 0;
    }
    return 1;
}

static int semaphore_v()
{
    //这是一个释放操作，它使信号量变为可用，即发送信号 V（sv）
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = 1;//V()
    sem_b.sem_flg = SEM_UNDO;
    if(semop(sem_id, &sem_b, 1) == -1)
    {
```

```
            fprintf(stderr, "semaphore_v failed\n");
            return 0;
        }
        return 1;
    }
```

运行结果如下：



注：这个程序的临界区为 main 函数 for 循环不的 semaphore_p 和 semaphore_v 函数中间的代码。

例子分析 ：同时运行一个程序的两个实例，注意第一次运行时，要加上一个字符作为参数，例如本例中的字符‘O’，它用于区分是否为第一次调用，同时这个字符输出到屏幕中。因为每个程序都在其进入临界区后和离开临界区前打印一个字符，所以每个字符都应该成对出现，正如你看到的上图的输出那样。在 main 函数中循环中我们可以看到，每次进程要访问 stdout（标准输出），即要输出字符时，每次都要检查信号量是否可用（即 stdout 有没有正在被其他进程使用）。所以，当一个进程 A 在调用函数 semaphore_p 进入了临界区，输出字符后，调用 sleep 时，另一个进程 B 可能想访问 stdout，但是信号量的 P 请求操作失败，只能挂起自己的执行，当进程 A 调用函数 semaphore_v 离开了临界区，进程 B 马上被恢复执行。然后进程 A 和进程 B 就这样一直循环了 10 次。

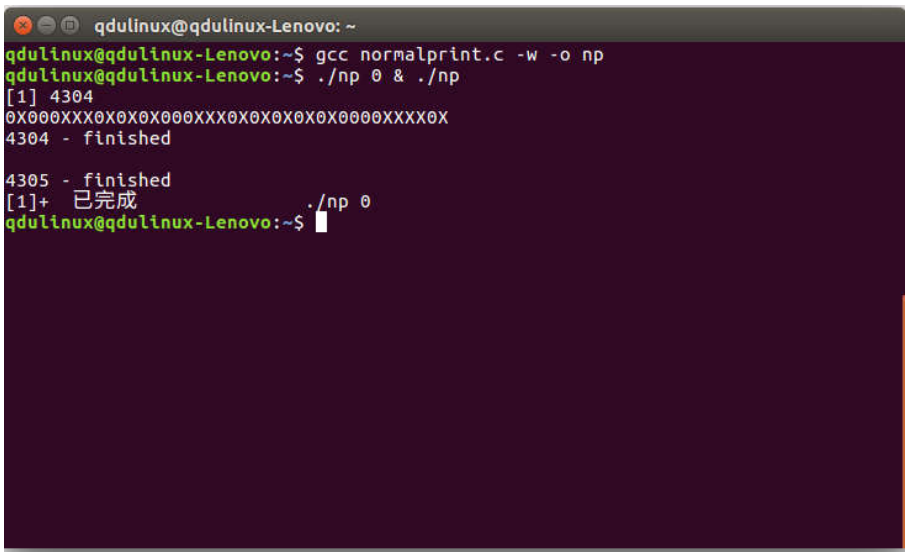
五、对比例子——进程间的资源竞争

看了上面的例子，你可能还不是很明白，不过没关系，下面我就以另一个例子来说明一下，它实现的功能与前面的例子一样，运行方式也一样，都是两个相同的进程，同时向 stdout 中输出字符，只是没有使用信号量，两个进程在互相竞争 stdout。它的代码非常简单，文件名为 normalprint.c，代码如下：

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

int main(int argc, char *argv[])
```

```
{
    char message = 'X';
    int i = 0;
    if(argc > 1)
        message = argv[1][0];
    for(i = 0; i < 10; ++i)
    {
        printf("%c", message);
        fflush(stdout);
        sleep(rand() % 3);
        printf("%c", message);
        fflush(stdout);
        sleep(rand() % 2);
    }
    sleep(10);
    printf("\n%d - finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

运行结果如下：



例子分析：
从上面的输出结果，我们可以看到字符‘X’和‘O’并不像前面的例子那样，总是成对出现，因为当第一个进程 A 输出了字符后，调用 sleep 休眠时，另一个进程 B 立即输出并休眠，而进程 A 醒来时，再继续执行输出，同样的进程 B 也是如此。所以输出的字符就是不成对的出现。这两个进程在竞争 stdout 这一共同的资源。通过两个例子的对比，我想信号量的意义和使用应该比较清楚了。

六、信号量的总结

信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行等待（即 P(信号变量))和发送（即 V(信号变量))信息操作。我们通常通过信号来解决多个进程对同一资源的访问竞争的问题，使在任一时刻只能有一个执行线程访问代码的临界区域，也可以说它是协调进程间的对同一资源的访问权，也就是用于同步进程的。

课程设计

题目：信号量实现进程同步

1 实验说明

利用信号量解决生产者-消费者问题。

2 解决方案

进程同步是操作系统多进程/多线程并发执行的关键之一， 进程同步指为完成共同任务的并发进程基于某个条件来协调它们的活动， 这是进程之间发生的一种直接制约关系， 生产者-消费者问题是典型的进程同步问题， 其本质是如何控制并发进程对有界共享主存区的访问。生产者进程生产产品， 然后将产品放置在一个空缓冲区中供消费者进程消费。 消费者进程从缓冲区中获得产品， 然后释放缓冲区。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空缓冲区。 当消费者进程消费产品时，如果没有满的缓冲区， 那么消费者进程将被阻塞， 直到新的产品被生产出来。

- 下面以生产者进程不断向数组添加数据(写入 100 次)， 消费者从数组读取数据并求和为例，给出基于信号量解决生产者-消费者问题的程序框架。 该程序假设有一个生产者进程和两个消费者进程， 创建了 fullid、 emptyid 和 mutexid 共 3 个信号量， 供进程间同步访问临界区。同时， 还建立 4 个共享主存区, 其中 array 用于维护生产者、消费者进程之间的共享数据， sum 保存当前求和结果， 而 set 和 get 分别记录当前生产者进程和消费者进程的读写次数。

代码：

```
#include <sys/mman.h>
#include <sys/types.h>
#include <linux/sem.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <time.h>
#define MAXSEM 5
//声明三个信号灯 ID
int fullid;
int emptyid;
int mutxid;
int main()
{
```

```
struct sembuf P, V;
union semun arg;
//声明共享主存
int *array;
int *sum;
int *set;
int *get;
//映射共享主存
array = (int *)mmap(NULL, sizeof( int ) * MAXSEM, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
 sum = (int *)mmap(NULL, sizeof( int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
 get = (int *)mmap(NULL, sizeof( int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
 set = (int *)mmap(NULL, sizeof( int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
*sum = 0;
*get = 0;
*set = 0;
//生成信号灯
fullid = semget(IPC_PRIVATE, 1, IPC_CREAT | 00666);
emptyid = semget(IPC_PRIVATE, 1, IPC_CREAT | 00666);
mutxid = semget(IPC_PRIVATE, 1, IPC_CREAT | 00666);
//为信号灯赋值
arg.val = 0;
if(semctl(fullid, 0, SETVAL, arg) == -1) perror("semctl setval error");
arg.val = MAXSEM;
if(semctl(emptyid, 0, SETVAL, arg) == -1) perror("semctl setval error");
arg.val = 1;
if(semctl(mutxid, 0, SETVAL, arg) == -1) perror("setctl setval error");
//初始化 P,V 操作
V.sem_num = 0;
V.sem_op = 1;
V.sem_flg = SEM_UNDO;
P.sem_num = 0;
P.sem_op = -1;
P.sem_flg = SEM_UNDO;

//生产者进程
if(fork() == 0 )    {
      int i = 0;
      while( i < 100) {
            semop(emptyid, &P, 1 );
            semop(mutxid, &P, 1);
            array[*(set) % MAXSEM] = i + 1;
            printf("Producer %d\n", array[(*set) % MAXSEM]);
            (*set)++;
            semop(mutxid, &V, 1);
```

```
            semop(fullid, &V, 1);
            i++;
        }
        sleep(10);
        printf("Producer is over");
        exit(0);
    }
    else {
        //ConsumerA    进程
        if(fork() == 0) {

            while(1) {
                semop(fullid, &P, 1);
                semop(mutxid, &P, 1);
                if(*get == 100)
                    break;
                *sum += array[(*get) % MAXSEM];

                printf("The ComsumerA Get Number %d\n", array[(*get) % MAXSEM] );
                (*get)++;
                if( *get == 100)
                    printf("The sum is %d \n ", *sum);
                semop(mutxid, &V, 1);
                semop(emptyid, &V, 1 );
                sleep(1);
            }
            printf("ConsumerA is over");
            exit(0);
        }
        else {
            //Consumer B 进程
            补充代码；
            }
        }
    }
    // sleep(20);
    return 0;
}
```