華中農業大學

HUAZHONG AGRICULTURAL UNIVERSITY

Linux 编程与应用

实　验　报　告

题目: 基于共享内存的读者和写者问题程序设计

姓名: 高星杰

学号: 2021307220712

学院: 信息学院

指导老师: 任继平

中国 武汉

2024 年 5 月

实验 3　基于共享内存的读者和写者问题程序设计

一、目的与要求

1. 目的

基于 Linux 中 IPC 通信机制的应用开发，掌握有关 IPC 通信机制函数的使用方法，并掌握读者与消写模式的程序开发。

2. 要求

（1）先启动读进程，它负责创建共享内存，读共享内存的数据。

（2）后启动写者进程。向共享内存写数据，数据内容的数量自定。

（3）采用信号灯解决涉及的同步与互斥问题

（4）本次实验的上机时间为 4 学时。

二、实验内容

1. 查阅文件 I/O 的有关资料

通过教材和网络资料，理解实验有关的原理。

2. 设计程序流程

根据实验要求设计程序流程，并采用图形形式展现。

3. 算法实现

应用 c 语言编写程序并调试，分析自己实现代码的不足和改进的想法。

4. 撰写报告

撰写实验报告。

5. 代码参考

参考教材或网络有关实现的源代码。

三、实验原理

在进行实验之前我们需要先了解实验原理。

1. IPC 通信机制

IPC（Inter-Process Communication，进程间通信）是指多个进程之间传递数据的方式。Linux 操作系统提供了多种 IPC 机制，主要包括以下几种：

(1) **管道（Pipes）和命名管道（Named Pipes）**

(2) **消息队列（Message Queues）**

(3) **共享内存（Shared Memory）**

(4) **信号量（Semaphores）**

(5) 套接字（Sockets）

(6) 信号（Signals）

(1) 管道（Pipes）和命名管道（Named Pipes）

管道（Pipes）

- 管道是最简单的 IPC 形式，主要用于有亲缘关系的进程之间（父子进程之间）的通信。
- 管道是半双工的（即数据只能在一个方向上传输），如果需要双向通信需要建立两个管道。
- 通过 pipe()系统调用创建，返回两个文件描述符，一个用于读，一个用于写。

命名管道（Named Pipes or FIFOs）

- 命名管道是管道的扩展，允许在无亲缘关系的进程之间通信。
- 命名管道是一个存在于文件系统中的特殊文件，通过路径名访问。
- 通过 mkfifo()系统调用创建。

(2) 消息队列（Message Queues）

- 消息队列是一种在进程之间传递消息的数据结构。
- 每个消息具有一个类型，可以根据消息类型有选择地接收消息。
- 消息队列在系统范围内唯一，通过消息队列标识符（由 msgget()返回）来标识。
- 消息通过 msgsnd()发送，msgrcv()接收。

(3) 共享内存（Shared Memory）

- 共享内存允许多个进程直接读写同一块物理内存，是最快的 IPC 方式。
- 共享内存段通过 shmget()创建，并返回一个标识符。
- 进程通过 shmat()将共享内存附加到自身地址空间，通过 shmdt()分离。
- 需要同步机制（如信号量）来控制对共享内存的访问。

(4) 信号量（Semaphores）

- 信号量用于解决进程间或线程间的同步与互斥问题。

- 信号量是一种计数器，用于控制对共享资源的访问。
- 通过 semget()创建信号量集，通过 semop()操作信号量，通过 semctl()控制信号量。
- 信号量可以用于保护共享内存中的临界区。

(5) **套接字（Sockets）**

- 套接字用于网络通信，也可以用于本地进程间通信。
- 支持多种通信协议（如 TCP、UDP、UNIX 域套接字）。
- 套接字是双向通信的，可以实现全双工通信。
- 通过 socket()系统调用创建，通过 bind()、listen()、accept()、connect()、send()、recv()等函数进行操作。

(6) **信号（Signals）**

- 信号是用于通知进程某个事件发生的机制。
- 每个信号有一个信号编号，对应不同的事件（如 SIGINT、SIGTERM 等）。
- 进程可以通过 signal()或 sigaction()设置信号处理函数。
- 进程通过 kill()、raise()等函数发送信号。

使用场景与优缺点

- **管道和命名管道**
  - 使用场景：父子进程通信、简单的数据流传输。
  - 优点：简单、开销小。
  - 缺点：半双工（管道）、无结构数据传输、仅适合小数据量传输。
- **消息队列**
  - 使用场景：无亲缘关系的进程间通信，需要消息结构和类型。
  - 优点：消息有结构、支持消息优先级。
  - 缺点：内核维护消息队列，开销较大。
- **共享内存**
  - 使用场景：大数据量传输、需要高效访问的场景。

- 优点：速度快、开销小。
- 缺点：**需要额外的同步机制来保证数据一致性**。

- **信号量**
  - 使用场景：进程或线程同步、互斥控制。
  - 优点：灵活、支持复杂同步需求。
  - 缺点：编程复杂，易产生死锁。
- **套接字**
  - 使用场景：网络通信、本地复杂通信。
  - 优点：强大、灵活、支持多种协议。
  - 缺点：开销较大，编程复杂。
- **信号**
  - 使用场景：简单事件通知、中断处理。
  - 优点：简单、快速。
  - 缺点：信号处理函数的执行受限，信号队列容易丢失信号。

Linux 提供了多种 IPC 机制，每种机制都有其适用的场景和优缺点。在实际应用中，应根据具体需求选择合适的 IPC 方式，以便在保证系统性能和可靠性的前提下，达到最佳的通信效果。**本次实验使用的是共享内存+信号量，这样访问速度很快，并且也能满足同步互斥需求。**

2. 读者写者问题

读者写者问题是一个经典的同步问题,它描述了多个线程对共享资源进行读写操作时的并发控制问题。

在读者写者问题中,共享资源可以被分为两类线程访问:读者线程和写者线程。

- 读者线程只对共享资源进行读取操作,不会修改资源。
- 写者线程对共享资源进行写入操作,会修改资源的内容。

问题的关键在于如何协调读者线程和写者线程对共享资源的访问,使得:

(1)多个读者线程可以同时读取共享资源,因为读取操作不会改变资源的内容。

(2)写者线程必须独占地访问共享资源,当写者线程正在写入时,不允许其他读者线程或写者线程访问该资源。

(3)读者线程和写者线程之间要互斥,即当有读者线程正在读取时,写者线程不能写入;当写者线程正在写入时,读者线程不能读取。

读者写者问题的目的是尽可能提高并发性,允许多个读者线程同时读取,以提高读取效率;同时,写者线程需要独占访问,以保证数据的一致性和正确性。

举个例子,假设有一个共享的文件,多个读者线程可以同时读取该文件的内容,但是当一个写者线程需要修改文件内容时,就必须独占访问该文件,阻止其他读者线程和写者线程的访问,直到写入操作完成。

读者写者问题的解决方案通常使用同步原语,如互斥锁、信号量等,来协调读者线程和写者线程对共享资源的访问,以保证数据的一致性和并发性。常见的解决方案有读者优先、写者优先、公平性等,根据具体的需求和场景选择适当的方案。

读者写者问题是并发编程中的一个重要问题,它体现了对共享资源的读写操作在并发环境下的挑战和解决思路。理解和掌握读者写者问题的概念和解决方案,对于开发高效、正确的并发程序具有重要意

四、实验步骤

1. 任务分析

本次实验可以大致分为两部分，一部分是解决共享内存读写的问题，另一部分是的使用信号量机制解决同步互斥问题。

首先解决共享内存读写的问题，共享内存是系统负责维护的，所以我们要使用系统调用来对共享内存进行读写具体要使用哪些系统调用呢？

(1) 共享内存的创建和分配

使用 shmget()系统调用创建共享内存段，并返回一个共享内存标识符（shmid）。

- **关键参数：**
  - key：共享内存段的键值，通常使用 ftok()生成。
  - size：共享内存段的大小（以字节为单位）。
  - flags：控制共享内存段的创建和访问权限，例如 IPC_CREAT 和权限标志。

(2) 连接共享内存

使用 shmat()系统调用将共享内存段连接到当前进程的地址空间。

- **关键参数：**
  - shmid：共享内存标识符。
  - shmaddr：让系统选择一个合适的地址（通常传入 NULL）。
  - shmflg：标志位，控制共享内存段的连接方式（通常为 0 表示读写模式）。

(3) 分离共享内存

使用 shmdt()系统调用将共享内存段从当前进程的地址空间分离。

- **关键参数：**
  - shmaddr：共享内存段的起始地址。

(4) 控制共享内存

使用 shmctl()系统调用对共享内存段进行控制，例如删除共享内存段或获取共享内存段的信息。

- **关键参数：**
  - shmid：共享内存标识符。
  - cmd：控制命令，例如 IPC_RMID（删除共享内存段）。
  - buf：用于传递或接收共享内存段的信息（通常为 NULL）。

我们使用这些系统调用就可以实现对共享内存的读写。

然后另一部分是的同步互斥问题，我们使用信号量机制解决。

在 Linux 中,信号量是一种常用的同步和互斥机制,用于在多个进程或线程之间协调对共享资源的访问。信号量本质上是一个非负整数计数器,附带两个原子操作:

(1) P 操作(wait 或 decrement):如果信号量的值大于 0,将其减 1;如果信号量的值为 0,则进程或线程会被阻塞,直到信号量的值变为大于 0。

(2) V 操作(signal 或 increment):将信号量的值加 1,如果有进程或线程因为等待该信号量而被阻塞,则唤醒其中一个进程或线程。

信号量的使用方式如下:

(1) 初始化信号量:使用 semget()函数创建一个新的信号量集合或获取一个已存在的信号量集合,并使用 semctl()函数设置信号量的初始值。

(2) 获取信号量:在访问共享资源之前,使用 semop()函数对信号量执行 P 操作。如果信号量的值为 0,进程或线程将被阻塞,直到信号量的值变为大于 0。

(3) 释放信号量:在访问完共享资源后,使用 semop()函数对信号量执行 V 操作,将信号量的值加 1,并唤醒等待该信号量的其他进程或线程。

(4) 删除信号量:使用 semctl()函数删除不再需要的信号量集合。

根据以上我们就可以实现的针对读者写者的同步互斥逻辑。使用系统调用来实现的共享内存的读写，然后同样使用系统调用实现信号量机制的同步互斥逻辑，最终我们就可实现实验要求。

2. 概要设计

(1) 流程描述

a) **读进程流程：**

- 创建共享内存和信号量。
- 初始化信号量，设置初始状态。
- 进入读取循环，等待写进程通知。
- 读取共享内存中的数据。
- 通知写进程可以写入新数据。

b) **写进程流程：**

- 获取共享内存和信号量的标识符。
- 进入写入循环，等待读进程通知。
- 向共享内存写入数据。
- 通知读进程可以读取数据。

3. 详细设计

(1) 组件描述

a) **共享享内存：**

- 创建和管理共享内存，用于读进程和写进程之间的数据交换。
- 使用 shmget()、shmat()、shmdt()和 shmctl()函数进行操作。

b) **信号灯（信号量）：**

- 用于控制共享内存的同步与互斥访问。
- 使用 semget()、semop()和 semctl()函数进行操作。

c) **读进程：**

- 创建并初始化共享内存和信号灯。
- 等待信号量的通知，从共享内存中读取数据。
- 读取完成后，通过信号量通知写进程可以写入数据。

d) **写进程：**

- 获取共享内存和信号灯的标识符。
- 等待信号量的通知，向共享内存写入数据。
- 写入完成后，通过信号量通知读进程可以读取数据。

(2) 数据结构

- **共享内存标识符：**用于标识共享内存段。

- **信号量标识符**：用于标识信号量集。

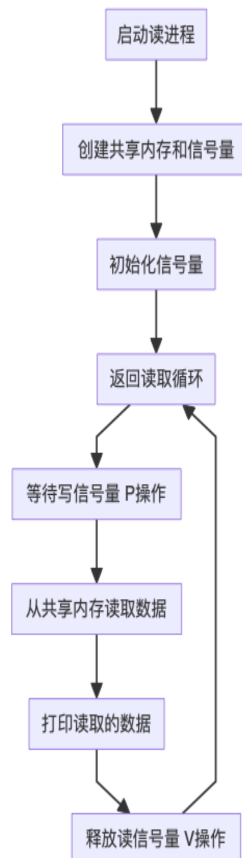
（3）同步与互斥控制

- **信号量初始化**：在读进程中初始化两个信号量，一个用于控制写进程写入，一个用于控制读进程读取。
- **信号量操作**：通过 semop()函数进行 P 操作和 V 操作，控制读写进程的访问顺序。

流程描述

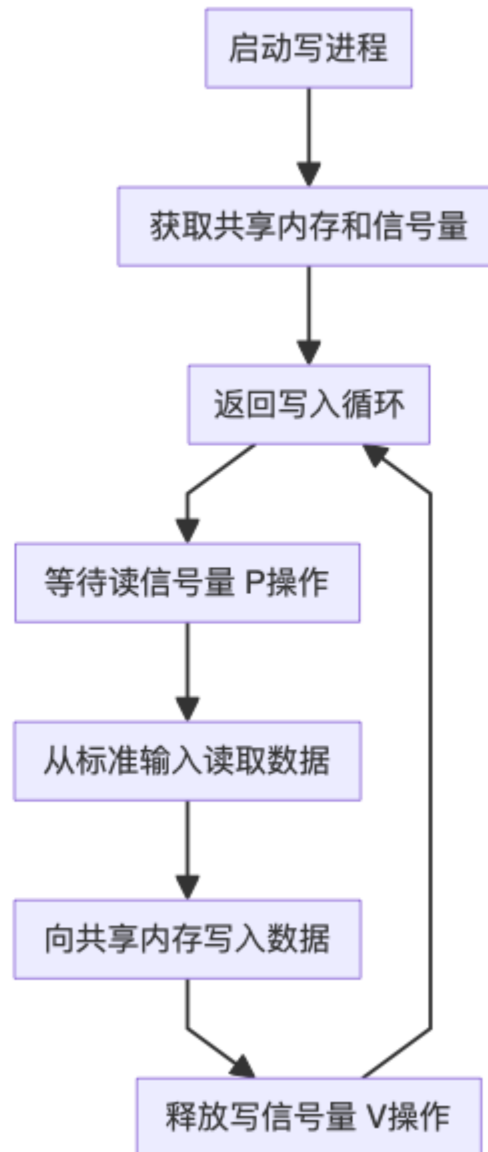
（1）读进程流程：

- 创建共享内存和信号量。
- 初始化信号量，设置初始状态。
- 进入读取循环：
  - 等待写信号量（P 操作）。
  - 从共享内存读取数据。
  - 打印读取的数据。
  - 释放读信号量（V 操作），通知写进程可以写入新数据。



(2) 写进程流程：

- 获取共享内存和信号量的标识符。
- 进入写入循环：
    - 等待读信号量（P 操作）。
    - 从标准输入读取数据。
    - 向共享内存写入数据。
    - 释放写信号量（V 操作），通知读进程可以读取新数据。



详细实现步骤

(1) 创建共享内存

- 使用 shmget()系统调用创建共享内存段，并返回共享内存标识符（shmid）。
- 共享内存大小设为 1024 字节（或根据需求调整）。

(2) 连接共享内存

- 使用 shmat()系统调用将共享内存段连接到当前进程的地址空间，并返回共享内存的起始地址。

(3) 分离共享内存

- 使用 shmdt()系统调用将共享内存段从当前进程的地址空间分离。

(4) 删除共享内存

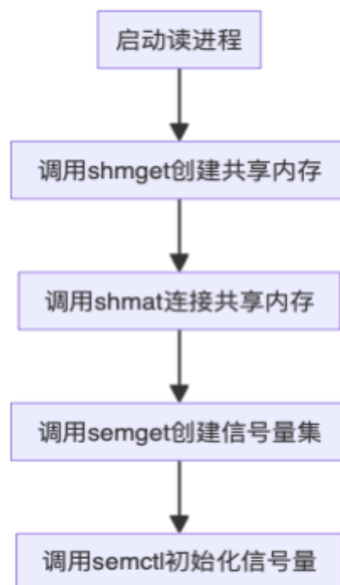
- 使用 shmctl()系统调用，使用 IPC_RMID 命令删除共享内存段。

(5) 创建信号量

- 使用 semget()系统调用创建信号量集，并返回信号量标识符（semid）。
- 信号量集包含两个信号量，一个控制读操作，一个控制写操作。

(6) 初始化信号量

- 使用 semctl()系统调用初始化信号量的值。
- 写信号量初始值设为 0，读信号量初始值设为 1。

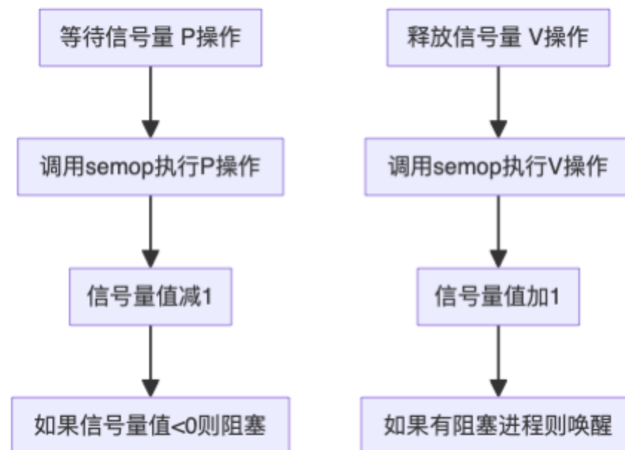


(7) 信号量 P 操作

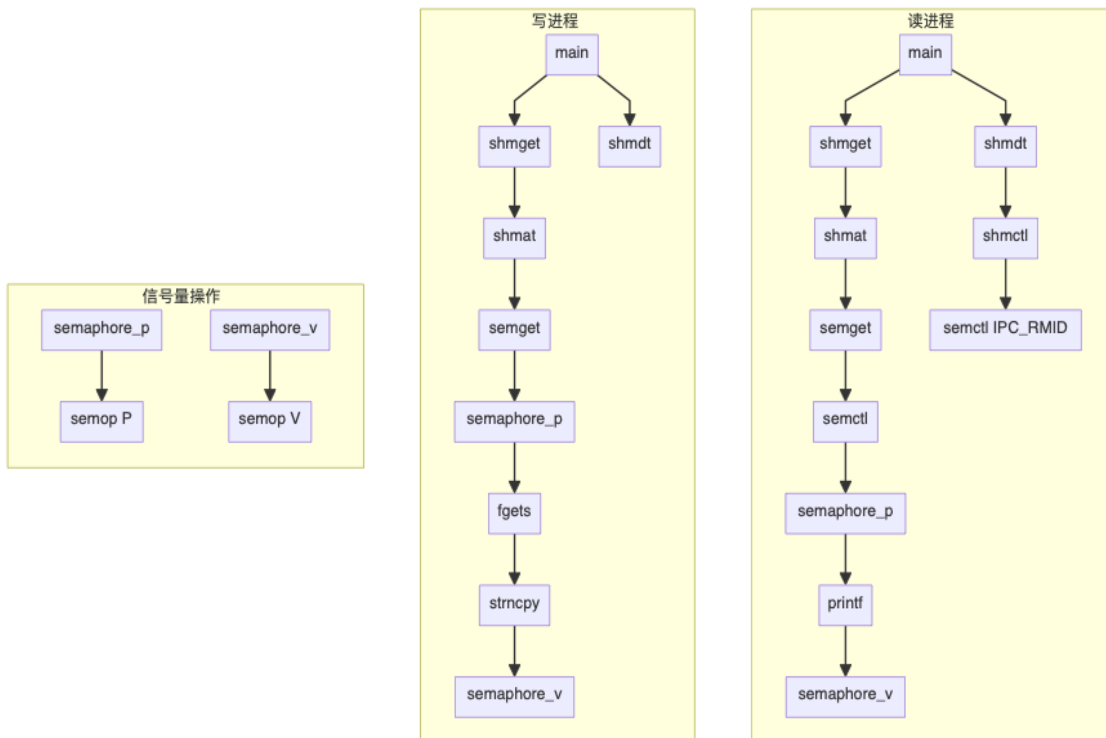
- 使用 semop()系统调用执行 P 操作，等待信号量变为正数并减 1。

(8) 信号量 V 操作

- 使用 semop()系统调用执行 V 操作，将信号量加 1，唤醒等待进程。



函数调用关系



4. 代码实现

Common.h

```
1. #ifndef COMMON_H
2. #define COMMON_H
3.
4. #include <sys/ipc.h>
5. #include <sys/shm.h>
6. #include <sys/sem.h>
7. #include <stdio.h>
8. #include <stdlib.h>
9. #include <string.h>
10. #include <unistd.h>
11.
12. #define SHM_KEY 0x1234
13. #define SEM_KEY 0x5678
14. #define SHM_SIZE 1024
15.
16. union my_semun
17. {
18.     int val;
19.     struct semid_ds *buf;
20.     unsigned short *array;
21.     struct seminfo *__buf;
22. };
23.
24. void init_semaphore(int semid, int semnum, int initval)
25. {
26.     union my_semun sem_union;
27.     sem_union.val = initval;
28.     if (semctl(semid, semnum, SETVAL, sem_union) == -1)
29.     {
30.         perror("semctl");
31.         exit(EXIT_FAILURE);
32.     }
33. }
34.
35. void semaphore_p(int semid, int semnum)
36. {
37.     struct sembuf sem_b;
38.     sem_b.sem_num = semnum;
39.     sem_b.sem_op = -1;
40.     sem_b.sem_flg = SEM_UNDO;
41.     if (semop(semid, &sem_b, 1) == -1)
42.     {
43.         perror("semop P");
44.         exit(EXIT_FAILURE);
45.     }
46. }
47.
48. void semaphore_v(int semid, int semnum)
49. {
50.     struct sembuf sem_b;
51.     sem_b.sem_num = semnum;
52.     sem_b.sem_op = 1;
53.     sem_b.sem_flg = SEM_UNDO;
```

```
54.     if (semop(semid, &sem_b, 1) == -1)
55.     {
56.         perror("semop V");
57.         exit(EXIT_FAILURE);
58.     }
59. }
60.
61. #endif // COMMON_H
```

Read. c

```
1.  #ifndef COMMON_H
2.  #define COMMON_H
3.
4.  #include <sys/ipc.h>
5.  #include <sys/shm.h>
6.  #include <sys/sem.h>
7.  #include <stdio.h>
8.  #include <stdlib.h>
9.  #include <string.h>
10. #include <unistd.h>
11.
12. #define SHM_KEY 0x1234
13. #define SEM_KEY 0x5678
14. #define SHM_SIZE 1024
15.
16. union my_semun
17. {
18.     int val;
19.     struct semid_ds *buf;
20.     unsigned short *array;
21.     struct seminfo *__buf;
22. };
23.
24. void init_semaphore(int semid, int semnum, int initval)
25. {
26.     union my_semun sem_union;
27.     sem_union.val = initval;
28.     if (semctl(semid, semnum, SETVAL, sem_union) == -1)
29.     {
30.         perror("semctl");
31.         exit(EXIT_FAILURE);
32.     }
33. }
34.
35. void semaphore_p(int semid, int semnum)
36. {
37.     struct sembuf sem_b;
38.     sem_b.sem_num = semnum;
39.     sem_b.sem_op = -1;
40.     sem_b.sem_flg = SEM_UNDO;
41.     if (semop(semid, &sem_b, 1) == -1)
42.     {
43.         perror("semop P");
44.         exit(EXIT_FAILURE);
45.     }
46. }
```

```
47.
48. void semaphore_v(int semid, int semnum)
49. {
50.     struct sembuf sem_b;
51.     sem_b.sem_num = semnum;
52.     sem_b.sem_op = 1;
53.     sem_b.sem_flg = SEM_UNDO;
54.     if (semop(semid, &sem_b, 1) == -1)
55.     {
56.         perror("semop V");
57.         exit(EXIT_FAILURE);
58.     }
59. }
60.
61. #endif // COMMON_H
```

Writer.c

```
1.  #include "common.h"
2.
3.  int main()
4.  {
5.      int shmid = shmget(SHM_KEY, SHM_SIZE, 0666);
6.      if (shmid == -1)
7.      {
8.          perror("shmget");
9.          exit(EXIT_FAILURE);
10.     }
11.
12.     void *shm = shmat(shmid, NULL, 0);
13.     if (shm == (void *)-1)
14.     {
15.         perror("shmat");
16.         exit(EXIT_FAILURE);
17.     }
18.
19.     int semid = semget(SEM_KEY, 2, 0666);
20.     if (semid == -1)
21.     {
22.         perror("semget");
23.         exit(EXIT_FAILURE);
24.     }
25.
26.     char buffer[SHM_SIZE];
27.     while (1)
28.     {
29.         printf("Enter some text: ");
30.         fgets(buffer, SHM_SIZE, stdin);
31.
32.         semaphore_p(semid, 1); // 等待读者完成读取
33.
34.         strncpy((char *)shm, buffer, SHM_SIZE);
35.
36.         semaphore_v(semid, 0); // 通知读者可以读取
37.     }
38.
39.     shmdt(shm);
```

```
40.     return 0;
41. }
```

5.　调试分析

编写完成的时候发报错，发现是 semun 的重复定义，原来系统中也有这个定义，所以改成了 my_semun 就好了。

```
new@newdeMBP code % gcc -o reader reader.c
In file included from reader.c:1:
./common.h:16:7: error: redefinition of 'semun'
union semun
      ^
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/sys/sem.h:177:7: note: previous definition is here
union semun {
      ^
1 error generated.
```

```
union my_semun
{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
    struct seminfo *__buf;
};

void init_semaphore(int semid, int semnum, int initval)
{
    union my_semun sem_union;
    sem_union.val = initval;
    if (semctl(semid, semnum, SETVAL, sem_union) == -1)
    {
        perror("semctl");
        exit(EXIT_FAILURE);
    }
}
```

```
new@newdeMBP code % gcc -o reader reader.c
new@newdeMBP code %
```

6. 测试结果

```
new@newdeMBP code % gcc -o reader reader.c
new@newdeMBP code % gcc -o writer writer.c
new@newdeMBP code % ./reader
Reader read: 123
```

```
new@newdeMBP code % ./writer
Enter some text: 123
```

7. 使用说明

(1) 编译代码：

```
gcc -o reader reader.c
gcc -o writer writer.c
```

(2) 先启动读进程：

```
./reader
```

(3) 再启动写者进程：

```
./writer
```

8. 改进设想

下面是一些可以改进的设想，以提升性能、可维护性和扩展性。

(1) 动态调整共享内存大小

问题：当前共享内存大小是固定的，无法根据需要动态调整。 **改进**：使用 shmctl()的 IPC_RESIZE 命令（如果系统支持）动态调整共享内存大小，或者创建新的共享内存段并复制数据。

(2) 信号量的动态管理

问题：信号量的数量和初始化值是固定的。 **改进**：设计一个信号量管理模块，能够根据需要动态创建、初始化和销毁信号量。可以使用一个结构来管理多个信号量和它们的状态。

(3) 增加超时机制

问题：当前信号量操作是阻塞的，没有超时机制。 **改进**：使用 semtimedop()代替 semop()，允许信号量操作设置超时时间。如果超时，则进行适当的处理。

(4) 读写优先级控制

问题：当前实现没有读写优先级控制，可能导致读饥饿或写饥饿。 **改进**：实现读优先或写优先机制，通过计数器和条件变量控制读写顺序。例如，可以通过增加一个写等待计数器来优先处理写操作。

(5) 增强错误处理机制

问题：当前错误处理简单，缺乏详细的日志和恢复机制。 **改进**：增加错误日志记录，详细记录每一步的错误信息。设计恢复机制，当出现错误时，尝试恢复共享内存和信号量的状态。

(6) 读写操作的批量处理

问题：当前每次读写操作只能处理一个数据单元，效率较低。 **改进**：设计批量处理机制，一次性读写多个数据单元，减少信号量操作的次数，提高效率。

(7) 内存保护机制

问题：共享内存段没有设置访问权限，存在潜在的安全问题。 **改进**：在创建共享内存段时，设置合适的权限，限制只有相关进程可以访问

五、实验总结

通过本次实验，我们成功实现了基于 Linux 共享内存和信号量机制的读者写者问题。在实验过程中，我们详细设计了共享内存的创建、连接、分离和删除操作，并通过信号量机制确保读写操作的同步与互斥。具体实现包括共享内存的创建与管理、信号量的创建与控制，以及读者写者问题的解决。共享内存的创建与管理涉及使用`shmget()`创建共享内存段，通过`shmat()`将共享内存段连接到当前进程的地址空间，并使用`shmdt()`将共享内存段从当前进程的地址空间分离，最后使用`shmctl()`删除共享内存段。而信号量的创建与控制则包括使用`semget()`创建信号量集，并返回信号量标识符，通过`semctl()`初始化信号量的值，使用`semop()`执行 P 操作和 V 操作，实现对共享内存的同步与互斥控制。读者写者问题的解决通过读进程和写进程的配合实现，读进程负责创建共享内存和信号量，初始化信号量，并通过 P 操作和 V 操作实现同步，写进程则通过获取共享内存和信号量标识符，进行数据写入操作，同步控制交由信号量机制实现。

通过本次实验，我深刻理解了共享内存的高效性、信号量的同步机制以及读者写者问题的挑战。共享内存是进程间通信最快捷的方式，因为它直接允许多个进程访问同一块物理内存，不需要在内核和用户空间之间进行数据拷贝。而信号量则是解决进程同步与互斥问题的有效工具，通过信号量可以轻松实现对共享资源的控制，避免竞争条件和数据不一致的问题。读者写者问题在于需要在保证数据一致性的前提下，最大化并发性能，通过实验了解到使用信号量可以有效协调读者和写者的操作，确保系统稳定性。

实验过程中，系统调用的应用是非常重要的，我们使用了多个系统调用（如`shmget`、`shmat`、`semget`、`semop`等），熟练掌握这些系统调用的使用方法是进行进程间通信编程的基础。此外，调试与错误处理也是实验中的一大挑战，我们遇到了一些如`semun`重复定义的问题，通过适当的调试和修改，最终解决了这

些问题。这让我意识到详细的错误日志记录和完善的错误处理机制对于系统的健壮性至关重要。

在实验基础上，我们提出了多个改进设想，如动态调整共享内存大小、信号量的动态管理、增加超时机制、读写优先级控制等。这些改进不仅提升了系统的性能和可维护性，也为未来的开发提供了参考方向。通过这些改进设想，我们可以进一步提升基于共享内存和信号量实现的读者写者问题的性能和可维护性。例如，通过使用`shmctl()`的`IPC_RESIZE`命令（如果系统支持）动态调整共享内存大小，或者创建新的共享内存段并复制数据，可以解决当前共享内存大小固定的问题。设计一个信号量管理模块，能够根据需要动态创建、初始化和销毁信号量，使用一个结构管理多个信号量及其状态，可以解决信号量的数量和初始化值固定的问题。使用`semtimedop()`代替`semop()`，允许信号量操作设置超时时间，如果超时则进行适当的处理，可以解决当前信号量操作阻塞的问题。实现读优先或写优先机制，通过计数器和条件变量控制读写顺序，例如通过增加一个写等待计数器优先处理写操作，可以解决当前实现没有读写优先级控制的问题。增加错误日志记录，详细记录每一步的错误信息，设计恢复机制，当出现错误时，尝试恢复共享内存和信号量的状态，可以增强错误处理机制。设计批量处理机制，一次性读写多个数据单元，减少信号量操作次数，提高效率，可以提升读写操作的效率。在创建共享内存段时，设置合适的权限，限制只有相关进程可以访问，可以增强内存保护机制。

通过本次实验和提出的改进设想，我们不仅掌握了 Linux IPC 机制的基本操作，还积累了处理进程间通信与同步问题的经验。这些经验和技巧对于未来的编程工作将非常有帮助。在未来的工作中，我会继续探索和应用更多的 IPC 机制，以提高系统的并发性能和稳定性。通过不断实践和总结，逐步提升自己的编程能力和项目开发水平。