Linux 编程与应用

实　验　报　告

题目:	文件 I/O 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1　文件I/O程序设计

一、目的与要求

1. 目的

基于Linux中文件I/O相关的应用开发，掌握有关文件操作函数的使用方法，并掌握生产者与消费者模式的程序开发。

2. 要求

（1）先启动生产者进程，它负责创建模拟FIFO结构的文件（其实是一个普通文件，不能使用FIFO文件）并投入生产，向文件写入自动生成的字符，字符内容自定义。

（2）后启动的消费者进程按照给定的数目进行消费。首先从文件中读取相应的数目的字符并在屏幕上显示，然后从文件中删除刚才消费过的数据。为了模拟FIFO结构，此时需要使用两次复制来实现文件内容的前移。

（3）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

要进行文件IO程序编程，我们就需要对文件IO进行了解。

1. 什么是文件IO？

文件IO也称为系统调用IO，是操作系统为"用户态"运行的进程和硬件交互提供的一组接口，即操作系统内核留给用户程序的一个接口，按照操作系统的结构划分，Linux系统自上而下依次是：用户进程、Linux内核、物理硬件。其中Linux内核包括系统调用接口和内核子系统两部分。Linux内核处于“承上启下”的关键位置，向

下管理物理硬件，向上为操作系统和应用程序提供接口，这里的接口就是系统调用。

2. Linux系统调用函数

我们要进行文件I/O程序设计就一定了解Linux的系统调用IO。

(1) open()：打开文件或创建新文件。

- 函数原型：int open(const char *pathname, int flags, mode_t mode);
- pathname：文件路径。
- flags：打开文件的标志，如 O_RDONLY（只读）、O_WRONLY（只写）、O_RDWR（读写）等。
- mode：创建新文件时指定文件的权限，如 S_IRUSR（用户读权限）、S_IWUSR（用户写权限）等。
- 返回值：成功返回文件描述符，失败返回 -1。

(2) read()：从文件中读取数据。

- 函数原型：ssize_t read(int fd, void *buf, size_t count);
- fd：文件描述符。
- buf：存储读取数据的缓冲区。
- count：要读取的字节数。
- 返回值：成功返回实际读取的字节数，失败返回 -1。

(3) write()：将数据写入文件。

- 函数原型：ssize_t write(int fd, const void *buf, size_t count);
- fd：文件描述符。
- buf：要写入的数据缓冲区。
- count：要写入的字节数。
- 返回值：成功返回实际写入的字节数，失败返回 -1。

(4) close()：关闭文件。

- 函数原型：int close(int fd);
- fd：要关闭的文件描述符。
- 返回值：成功返回 0，失败返回 -1。

(5) lseek()：调整文件的读写位置。

- 函数原型：off_t lseek(int fd, off_t offset, int whence);
- fd：文件描述符。
- offset：偏移量。
- whence：偏移起点，可以是 SEEK_SET（文件起始位置）、

SEEK_CUR（当前位置）或 SEEK_END（文件末尾）。

- 返回值：成功返回新的文件位置，失败返回 -1。

(6) stat()：获取文件的属性信息。

- 函数原型：int stat(const char *pathname, struct stat *statbuf);
- pathname：文件路径。
- statbuf：存储文件属性信息的结构体指针。
- 返回值：成功返回 0，失败返回 -1。

(7) dup()/dup2()：复制文件描述符。

- 函数原型：int dup(int oldfd); 和 int dup2(int oldfd, int newfd);
- oldfd：原始文件描述符。
- newfd：新的文件描述符（仅适用于 dup2()）。
- 返回值：成功返回新的文件描述符，失败返回 -1。

3. 生产者和消费者问题

本次实验的关键就是解决这个问题，生产者消费者问题（英语：Producer-consumer problem），也称有限缓冲问题（英语：Bounded-buffer problem），是一个多线程同步问题的经典案例。该问题描述了共享固定大小缓冲区的两个线程——即所谓的“生产者”和“消费者”——在实际运行时会发生的问题。生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程。与此同时，消费者也在缓冲区消耗这些数据。该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。

要解决该问题，就**必须让生产者在缓冲区满时休眠（要么干脆就放弃数据），等到下次消费者消耗缓冲区中的数据的时候，生产者才能被唤醒，开始往缓冲区添加数据。**同样，也可以让消费者在缓冲区空时进入休眠，等到生产者往缓冲区添加数据之后，再唤醒消费者。通常采用进程间通信的方法解决该问题。如果解决方法不够完善，则容易出现死锁的情况。出现死锁时，两个线程都会陷入休眠，等待对方唤醒自己。

四、实验步骤

1. 任务分析

按照实验要求，**有两问题一是 IO 系统调用，二是同步和互斥问题**。首先我们要设计两个程序都可以进行 IO 系统调用，一个可以调用系统调用进行创建文件和写入文件数据，另一个可以也同样可以进行调用系统调用可以进行数据读取和删除

数据文件。其次就是要处理同时运行所产生的同步和互斥问题，一个作为生产者，一个作为消费者，解决消费者和生产者的问题。

查阅 linux 的 IO 系统调用可以解决文件的读写问题，而同步互斥问题的解决办法就是加锁，本次实验使用的是信号量机制来解决同步互斥问题。

2. 信号量机制分析（概要设计）

IO 系统调用的接口在实验原理中已经讲过，直接调用即可，故不再赘述。直接进行分析同步互斥的解决方案——信号量机制。

解决方案分析：

(1) **信号量的创建和初始化：**

- **生产者进程**创建一个命名信号量，并将其初始值设置为 1。初始值 1 表示资源（文件）当前是可用的。
- **消费者进程**打开同一个命名信号量。

(2) **获取和释放信号量：**

- **获取信号量**（sem_wait）：在访问共享文件之前，进程必须获取信号量。如果信号量的值为 0（文件正被另一个进程使用），进程将阻塞，等待信号量的值变为大于 0。
- **释放信号量**（sem_post）：在完成文件访问后，进程必须释放信号量，将信号量的值加 1，从而允许其他阻塞的进程继续执行。

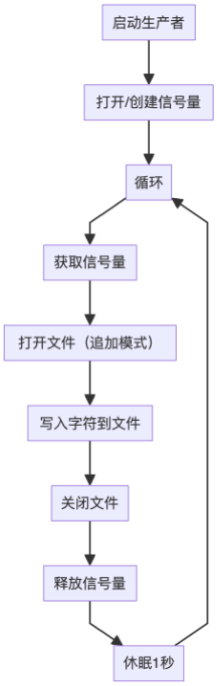
(3) **文件访问的同步和互斥控制：**

- **生产者进程：**
  - 生产者获取信号量后，打开文件并写入数据，然后关闭文件。
  - 释放信号量，表示文件现在是可用的。
- **消费者进程：**
  - 消费者获取信号量后，打开文件并读取数据，然后关闭文件。
  - 读取后，消费者重新整理文件内容（删除已消费的数据），然后再次释放信号量。

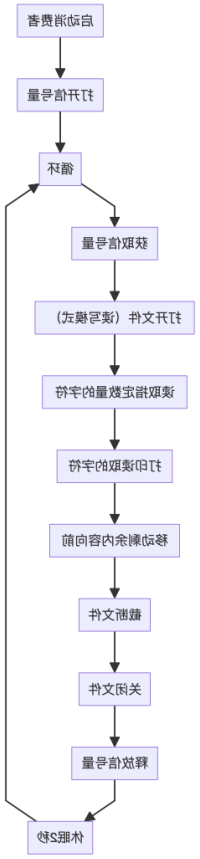
3. 程序流程设计（详细设计）

- **生产者流程**
  - 打开/创建信号量。
  - 循环生成字符：
    - 获取信号量。
    - 打开文件（追加模式）。
    - 写入字符。

        - 关闭文件。
        - 释放信号量。
        - 休眠一秒。



- **消费者流程**
    - 打开信号量。
    - 循环消费数据：
        - 获取信号量。
        - 打开文件（读写模式）。
        - 读取指定数量的字符。
        - 打印读取的字符。
        - 将文件剩余内容前移并截断文件。
        - 关闭文件。
        - 释放信号量。
        - 休眠两秒。



4. 代码实现

生产者：

```
#include <iostream>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <ctime>
#include <cstdlib>

#define FIFO_FILE "fifo_file.txt"
#define SEM_NAME "/fifo_semaphore"

void producer() {
    sem_t* sem = sem_open(SEM_NAME, O_CREAT, 0644, 1);
    if (sem == SEM_FAILED) {
        std::cerr << "Failed to create semaphore" << std::endl;
        exit(EXIT_FAILURE);
    }

    std::string chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    srand(time(0));

    while (true) {
        sem_wait(sem); // 获取信号量

        int fd = open(FIFO_FILE, O_WRONLY | O_CREAT | O_APPEND, 0644);
```

```
        if (fd == -1) {
            std::cerr << "Failed to open file" << std::endl;
            sem_post(sem);
            exit(EXIT_FAILURE);
        }

        char c = chars[rand() % chars.size()];
        if (write(fd, &c, 1) != 1) {
            std::cerr << "Failed to write to file" << std::endl;
            close(fd);
            sem_post(sem);
            exit(EXIT_FAILURE);
        }

        close(fd); // 关闭文件描述符
        std::cout << "Producer wrote: " << c << std::endl;

        sem_post(sem); // 释放信号量
        sleep(1); // 等待 1 秒
    }

    sem_close(sem); // 关闭信号量
}

int main() {
    producer();
    return 0;
}
```

消费者

```
#include <iostream>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <vector>

#define FIFO_FILE "fifo_file.txt"
#define SEM_NAME "/fifo_semaphore"

void consumer(int n) {
    sem_t* sem = sem_open(SEM_NAME, 0);
    if (sem == SEM_FAILED) {
        std::cerr << "Failed to open semaphore" << std::endl;
        exit(EXIT_FAILURE);
    }

    while (true) {
        sem_wait(sem); // 获取信号量

        int fd = open(FIFO_FILE, O_RDWR);
        if (fd == -1) {
            std::cerr << "Failed to open file" << std::endl;
            sem_post(sem);
            exit(EXIT_FAILURE);
        }

        std::vector<char> buffer(n);
        ssize_t bytesRead = read(fd, buffer.data(), n);
        if (bytesRead == -1) {
            std::cerr << "Failed to read from file" << std::endl;
```

```
            close(fd);
            sem_post(sem);
            exit(EXIT_FAILURE);
        }

        if (bytesRead > 0) {
            std::cout << "Consumer read: ";
            for (ssize_t i = 0; i < bytesRead; ++i) {
                std::cout << buffer[i];
            }
            std::cout << std::endl;

            off_t offset = lseek(fd, 0, SEEK_SET);
            if (offset == -1) {
                std::cerr << "Failed to seek in file" << std::endl;
                close(fd);
                sem_post(sem);
                exit(EXIT_FAILURE);
            }

            ssize_t remainingBytes = lseek(fd, 0, SEEK_END) - bytesRead;
            lseek(fd, bytesRead, SEEK_SET);
            std::vector<char> remainingBuffer(remainingBytes);

            if (read(fd, remainingBuffer.data(), remainingBytes) != remainingBytes) {
                std::cerr << "Failed to read remaining data" << std::endl;
                close(fd);
                sem_post(sem);
                exit(EXIT_FAILURE);
            }

            ftruncate(fd, 0);
            lseek(fd, 0, SEEK_SET);
            if (write(fd, remainingBuffer.data(), remainingBytes) != remainingBytes) {
                std::cerr << "Failed to write remaining data" << std::endl;
                close(fd);
                sem_post(sem);
                exit(EXIT_FAILURE);
            }
        }

        close(fd); // 关闭文件描述符
        sem_post(sem); // 释放信号量
        sleep(2); // 等待 2 秒
    }

    sem_close(sem); // 关闭信号量
}

int main() {
    consumer(5);
    return 0;
}
```

5. 程序运行结果分析

```
new@newdeMBP code %
```

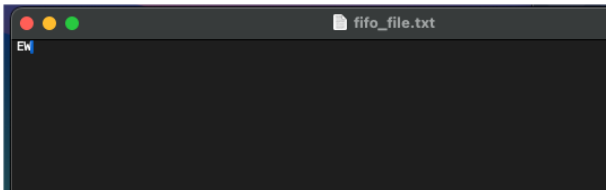
首先使用编译器编译文件，因为文件涉及到文件I/O操作和POSIX信号量，所以编译是要携带pthread库

可以发现编译成功。

然后要先运行producer再运行consumer（上次运行剩余的KFQWLGJJGR）

```
new@newdeMBP code % ./producer
Producer wrote: R
Producer wrote: V
Producer wrote: E
Producer wrote: K
Producer wrote: X
Producer wrote: C
Producer wrote: V
Producer wrote: D
Producer wrote: O
Producer wrote: N
Producer wrote: B
Producer wrote: F
Producer wrote: Y
Producer wrote: O
Producer wrote: S
Producer wrote: B
Producer wrote: U
Producer wrote: C
Producer wrote: R
Producer wrote: Y
Producer wrote: F
Producer wrote: Y
Producer wrote: M
```

```
new@newdeMBP code % ./consumer
Consumer read: KFQWL
Consumer read: GJJGR
Consumer read: VEKXC
Consumer read: VDONB
Consumer read: FY
Consumer read: OS
Consumer read: BU
Consumer read: CR
Consumer read: YF
Consumer read: YM
```



可以发现生产者产生的内容都被消费者消费了，而且是按照FIFO的顺序的，并且最后生产者多生产的也在文件中可以看到，说明互斥和同步、文件IO的问题都得到了解决。

五、实验总结

通过本次实验，我在实际操作中加深了对生产者-消费者模型的理解，并掌握了在 Linux 环境下实现该模型的具体方法。本实验的主要目标是通过文件 I/O 操作

模拟 FIFO（先进先出）结构，同时利用信号量解决生产者和消费者之间的同步与互斥问题。

1. 理论与实践结合

在上学期的操作系统课程中，我们已经从理论上学习了生产者-消费者问题，这次实验是在此理论基础上的一次实际应用。通过编写和调试生产者和消费者进程代码，我不仅巩固了理论知识，也对代码实现的细节有了更深的掌握。

2. 文件I/O操作

本次实验要求我们熟练使用 Linux 的文件 I/O 系统调用，如 open(), read(), write(), close(), lseek()等。这些系统调用是实现文件读写操作的基础，通过实验，我对这些系统调用的使用有了更加全面的了解。

3. 同步与互斥

实验的关键是解决生产者和消费者之间的同步与互斥问题。我们使用了 POSIX 信号量（sem_t）来控制对共享文件的访问，确保在任何时刻只有一个进程能够访问文件。这避免了竞态条件的发生，保证了数据的一致性和程序的正确性。

4. 实验结果

实验的最终结果显示，生产者进程能够不断地生成字符并写入文件，而消费者进程能够按照 FIFO 顺序从文件中读取字符并删除已消费的数据。通过观察文件内容和程序输出，可以确认生产者和消费者之间的同步与互斥问题得到了有效解决。

5. 实验体会

本次实验不仅让我深入理解了生产者-消费者模型的实际应用，也让我体会到了在多进程编程中处理同步与互斥问题的重要性。通过实际操作，我学会了如何使用信号量来控制进程间的资源竞争，确保程序的正确性和高效性。这次实验增强了我对操作系统原理的理解，同时提高了我的编程能力和解决实际问题的能力。