

# 西安交通大学

## 操作系统实验

2023-2024 年度秋季学期

---

### 操作系统专题实验报告

姓 名: 宋欣衡

学 号: 2216113545

专 业 班 级: 计算机 2103



## 目录

实验一 进程、线程相关编程经验.....	8
1.1 进程相关编程实验.....	8
1.1.1 实验目的.....	8
1.1.2 实验内容.....	8
1.1.3 实验思想.....	10
1.1.4 实验步骤.....	11
1.1.5 测试数据设计.....	15
1.1.6 程序运行初值及运行结果分析.....	15
1.1.7 实验总结.....	15
1.1.7.1 实验中的问题与解决过程.....	15
1.1.7.2 实验收获 .....	16
1.1.7.3 意见与建议.....	16
1.1.8 附件.....	16
1.1.8.1 附件 1 程序.....	16
1.1.8.2 附件 2 Readme.....	17
1.2 线程相关编程实验.....	17

1.2.1 实验目的 .....	17
1.2.2 实验内容 .....	17
1.2.3 实验思想 .....	18
1.2.4 实验步骤 .....	19
1.2.5 测试数据设计 .....	27
1.2.6 程序运行初值及运行结果分析 .....	27
1.2.7 实验总结 .....	27
1.2.7.1 实验中的问题与解决过程 .....	27
1.2.7.2 实验收获 .....	28
1.2.7.3 意见与建议 .....	29
1.2.8 附件 .....	29
1.2.8.1 附件 1 程序 .....	29
1.2.8.2 附件 2 Readme .....	30
1.3 自旋锁实验 .....	30
1.3.1 实验目的 .....	30
1.3.2 实验内容 .....	30
1.3.3 实验思想 .....	31

1.3.4 实验步骤.....	31
1.3.5 测试数据设计.....	34
1.3.6 程序运行初值及运行结果分析.....	34
1.3.7 实验总结.....	34
1.3.7.1 实验中的问题与解决过程.....	34
1.3.7.2 实验收获 .....	35
1.3.7.3 意见与建议.....	35
1.3.8 附件.....	35
1.3.8.1 附件 1 程序.....	35
1.3.8.2 附件 2 Readme.....	35
实验二 进程通信与内存管理.....	36
2.1 进程的软中断通信.....	36
2.1.1 实验目的.....	36
2.1.2 实验内容.....	36
2.1.3 实验前准备 .....	37
2.1.4 实验步骤.....	41
2.1.5 程序运行初值及运行结果分析.....	41

2.1.6 实验总结 .....	43
2.1.6.1 实验中的问题与解决过程 .....	43
2.1.6.2 实验收获 .....	44
2.1.6.3 意见与建议 .....	44
2.1.7 附件 .....	44
2.1.7.1 附件 1 程序 .....	44
2.1.7.2 附件 2 Readme .....	44
2.2 进程的管道通信 .....	44
2.2.1 实验目的 .....	44
2.2.2 实验内容 .....	45
2.2.3 实验前准备 .....	45
2.2.4 实验步骤 .....	48
2.2.5 运行结果分析 .....	50
2.2.6 实验总结 .....	50
2.2.6.1 实验中的问题与解决过程 .....	50
2.2.6.2 实验收获 .....	50
2.2.6.3 意见与建议 .....	51

2.2.7 附件 .....	51
2.2.7.1 附件 1 程序.....	51
2.2.7.2 附件 2 Readme.....	51
2.3 内存的分配与回收 .....	51
2.3.1 实验目的 .....	51
2.3.2 实验内容 .....	51
2.3.3 实验前准备 .....	52
2.3.4 实验步骤 .....	56
2.3.5 测试数据设计 .....	61
2.3.6 程序运行初值及运行结果分析 .....	62
2.3.7 实验总结 .....	65
2.3.7.1 实验中的问题与解决过程.....	65
2.3.7.2 实验收获 .....	65
2.3.7.3 意见与建议 .....	67
2.3.8 附件 .....	67
2.3.8.1 附件 1 程序.....	67
2.3.8.2 附件 2 Readme.....	67

实验三 Linux 动态模块与设备驱动 .....	68
3.1 实验目的 .....	68
3.2 实验内容 .....	68
3.3 实验思想 .....	69
3.4 实验步骤 .....	70
3.4.1 编写和测试基本的 "Hello World" 内核模块 .....	70
3.4.2 编写和测试字符设备驱动程序 <code>char_dever**</code> .....	72
3.5 程序运行初值及运行结果分析 .....	76
3.6 实验总结 .....	77
3.6.1 实验中的问题与解决过程 .....	77
3.6.2 实验收获 .....	78
3.6.3 意见与建议 .....	78
3.7 附件 .....	78
3.7.1 附件 1 程序 .....	78
3.7.2 附件 2 Readme .....	78

## 实验一 进程、线程相关编程经验

### 1.1 进程相关编程实验

#### 1.1.1 实验目的

- (1) 熟悉 Linux 操作系统的基本环境和操作方法，通过运行系统命令查看系统基本信息以了解系统；
- (2) 编写并运行简单的进程调度相关程序，体会进程调度、进程间变量的管理等机制在操作系统实际运行中的作用。

#### 1.1.2 实验内容

- (1) 熟悉操作命令、编辑、编译、运行程序。完成图 1-1 程序的运行验证，多运行几次程序观察结果；去除 wait 后再观察结果并进行理论分析。



```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        pid1 = getpid();
        printf("child: pid = %d",pid); /* A */
        printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
        pid1 = getpid();
        printf("parent: pid = %d",pid); /* C */
        printf("parent: pid1 = %d",pid1); /* D */
        wait(NULL);
    }

    return 0;
}

```

图 1-1 教材中所给代码（p103 作业 3.7）

(2) 扩展图 1-1 的程序:

- a) 添加一个全局变量并在父进程和子进程中对这个变量做不同操作，输出操作结果并解释;
- b) 在 return 前增加对全局变量的操作并输出结果，观察并解释;

c) 修改程序体会在子进程中调用 `system` 函数和在子进程中调用 `exec` 族函数;

### 1.1.3 实验思想

(1) 进程： 进程是计算机科学中的一个重要概念，它是操作系统中的基本执行单位。进程代表着一个正在执行的程序实例，它包括了程序的代码、数据和执行状态等信息。操作系统通过进程管理来实现对计算机资源的有效分配和控制;

(2) PID： PID 是进程标识符 (Process Identifier) 的缩写，它是用来唯一标识一个操作系统中的进程的数值。每个正在运行或已经终止的进程都会被分配一个唯一的 PID，这个标识符可以用来在操作系统内部识别和管理进程;

(3) `fork()`函数： `fork()` 是一个在类 Unix 操作系统中常见的系统调用，用于创建一个新的进程，新进程是原进程（父进程）的副本。新进程被称为子进程，它与父进程共享很多资源，但也有一些独立的属性。`fork()` 被用于实现多进程编程，常见于操作系统和并发编程中。函数返回一个整数，如果返回值为负数，则表示创建进程失败。如果返回值为 0，表示当前正在执行的代码是在子进程中。如果返回值大于 0，表示当前正在执行的代码是在父进程中，返回值是子进程的 PID。调用 `fork()` 函数时，操作系统会创建一个新的进程，该进程是调用进程的一个副本，称为子进程。子进程几乎与父进程相同，包括代码、数据、文件描述符等。但是子进程拥有自己的独立的内存空间和资源。

### 1.1.4 实验步骤

本实验通过在程序中输出父、子进程的 pid，分析父子进程 pid 之间的关系，进一步加入 wait()函数分析其作用。

步骤一： 编写并多次运行图 1-1 中代码

```
[root@kp-test01 lab1]# gcc 1.c -o 1-1
[root@kp-test01 lab1]# ./1-1
parent: pid = 6350
child: pid = 0
parent: pid_new = 6349
child: pid_new = 6350
[root@kp-test01 lab1]# ./1-1
parent: pid = 6352
child: pid = 0
parent: pid_new = 6351
child: pid_new = 6352
[root@kp-test01 lab1]# ./1-1
parent: pid = 6354
child: pid = 0
parent: pid_new = 6353
child: pid_new = 6354
[root@kp-test01 lab1]#
```

步骤二： 删去图 1-1 代码中的 wait()函数并多次运行程序，分析运行结果。

```
[root@kp-test01 lab1]# ./1-1
parent: pid = 6362
child: pid = 0
parent: pid_new = 6361
child: pid_new = 6362
[root@kp-test01 lab1]# ./1-1
child: pid = 0
parent: pid = 6364
child: pid_new = 6364
parent: pid_new = 6363
[root@kp-test01 lab1]# ./1-1
parent: pid = 6366
child: pid = 0
parent: pid_new = 6365
child: pid_new = 6366
[root@kp-test01 lab1]# ./1-1
parent: pid = 6368
child: pid = 0
parent: pid_new = 6367
child: pid_new = 6368
[root@kp-test01 lab1]#
```

可以发现当前情况下存在 child 有可能先于 parent 进程运行。

步骤三： 修改图 1-1 中代码，增加一个全局变量并在父子进程中对其进行不同的操作，观察并解释所做操作和输出结果。

```
[root@kp-test01 lab1]# ./1-1
parent: pid = 8835
child: pid = 0
parent: pid_new = 8834
child: pid_new = 8835
parent: flag = 2000
child: flag = 1000
[root@kp-test01 lab1]# ./1-1
parent: pid = 8837
child: pid = 0
parent: pid_new = 8836
child: pid_new = 8837
parent: flag = 2000
child: flag = 1000
[root@kp-test01 lab1]# ./1-1
parent: pid = 8839
parent: pid_new = 8838
child: pid = 0
parent: flag = 2000
child: pid_new = 8839
child: flag = 1000
[root@kp-test01 lab1]# |
```

我定义了一个全局变量 flag，在 parent 进程中修改其为 2000，child 进程中修改为 1000 并分别打印结果。

父进程与子进程先后交替运行，合理的。

步骤四：在步骤三基础上，在 return 前增加对全局变量的操作（自行设计）

并输出结果，观察并解释所做操作和输出结果

```

[root@kp-test01 lab1]# ./1-1
parent: pid = 6378
child: pid = 0
parent: pid_new = 6377
child: pid_new = 6378
parent: flag = 2000
child: flag = 1000
pid = 0 program before end: flag = 1000
pid = 6378 program before end: flag = 2000
[root@kp-test01 lab1]# ./1-1
parent: pid = 6380
child: pid = 0
parent: pid_new = 6379
child: pid_new = 6380
parent: flag = 2000
child: flag = 1000
pid = 0 program before end: flag = 1000
pid = 6380 program before end: flag = 2000
[root@kp-test01 lab1]#

```

清晰明了。

步骤五： 修改图 1-1 程序，在子进程中调用 `system()`与 `exec` 族函数。 编写 `systemcall.c` 文件输出进程号 `PID`，编译后生成 `systemcall` 可执行文件。在子进程中调用 `system_call`,观察输出结果并分析总结。

```

[root@kp-test01 lab1]# ./1-1
parent: pid = 8854
Using system to call system_call:
parent: pid_new = 8853
parent: flag = 2000
This is system_call. My PID is: 8855
Using exec to call system_call:
This is system_call. My PID is: 8854
pid = 8854 program before end: flag = 2000
[root@kp-test01 lab1]#

```

使用 `system()` 调用

1. **父进程 PID：** 父进程的 `PID` (Process ID) 是 8853。这是该进程的唯一标识符。
2. **子进程 PID：** 子进程通过 `system()` 调用 `system_call` 执行文件时，生成了一个新的进程，其 `PID` 是 8855。

3. **函数执行顺序：** 父进程的代码先执行了，之后子进程通过 `system()` 执行了 `system_call`。

使用 `exec()` 调用

1. **子进程替换：** `exec()` 函数替换了子进程（PID 8854）的内容。因此，这个 PID 与父进程中显示的子进程 PID 一致。
2. **程序流：** 由于 `exec()` 替换了子进程的内容，`exec()` 之后的任何代码都不会被执行。

总结

1. **进程独立性：** 使用 `system()` 创建了一个全新的进程（PID 8855）来执行 `system_call`，而父进程（PID 8853）和子进程（PID 8854）都继续执行了剩下的代码。
2. **进程替换：** 使用 `exec()` 替换了子进程的内容，所以新的 `system_call` 运行在原子进程（PID 8854）的上下文中，而没有创建新的进程。
3. **控制流：** 两种方法都在子进程中成功调用了 `system_call`，但 `system()` 允许子进程继续执行其他代码，而 `exec()` 则完全替换了子进程，使得 `exec()` 之后的代码不会被执行。

### 1.1.5 测试数据设计

无需数据测试。

### 1.1.6 程序运行初值及运行结果分析

运行结果已经分析。

### 1.1.7 实验总结

#### 1.1.7.1 实验中的问题与解决过程

1. **问题：隐式函数声明警告**
  - **描述：** 在最初的版本中，使用了 `wait(NULL)` 函数，但没有包含 `<sys/wait.h>` 头文件，导致编译器发出“implicit declaration of function”警告。
  - **解决：** 在代码中加入 `#include <sys/wait.h>` 来解决这个问题。
2. **问题：全局变量的影响**
  - **描述：** 当添加了全局变量后，发现父子进程中全局变量的变化是独立的。

- **解决：** 经研究，明确了 `fork()` 在复制进程时会复制数据段，因此全局变量在父子进程中是独立的。
3. **问题： `system()` 和 `exec()` 的用法**
- **描述：** 在尝试在子进程中调用 `system()` 和 `exec()` 函数时，初次遇到一些困惑和不熟悉的用法。
  - **解决：** 通过查阅文档和测试，理解了这两个函数的基本用法和作用，并成功地在代码中应用了它们。

### 1.1.7.2 实验收获

1. **进程管理理解深化：** 通过这个实验，更加深入地了解 Linux 系统中进程的创建、管理和调度。特别是通过观察 `wait()` 函数的行为，理解了父子进程间同步的重要性。
2. **编程技巧提升：** 这个实验让我更熟悉了 C 语言的编程模式，尤其是涉及到系统级调用和进程管理的函数。对 `fork()`, `wait()`, `system()`, 和 `exec()` 等函数有了更深入的了解。
3. **系统调用与命令行工具：** 实验中涉及到 `system()` 和 `exec()` 系列函数，使我了解了如何在程序中执行系统命令，以及如何用 `exec()` 替换当前进程的执行内容。
4. **多进程编程模型：** 通过在一个程序中创建多个进程，以及管理这些进程的行为和状态，我对多进程编程有了更实际的认识和理解。

### 1.1.7.3 意见与建议

1. **增加更多的进程管理实验：** 当前实验内容虽然涵盖了基础的进程创建和管理，但在实际应用中还有更多高级的用法，比如多进程并发处理，进程通信等，建议加入这部分内容。
2. **提供更详细的函数文档和示例代码：** 尽管实验手册给出了基础框架，但更多具体函数的使用例子和文档将会更有助于理解。
3. **加强对错误处理的教学：** 在实际编程中，错误处理是非常重要的的一环。本次实验虽然有简单的错误处理，但没有详细介绍这方面的最佳实践。

## 1.1.8 附件

### 1.1.8.1 附件 1 程序

1-1.c



*system\_call.c*

### 1.1.8.2 附件 2 Readme

*lab1-1Readme.md*

## 1.2 线程相关编程实验

### 1.2.1 实验目的

探究多线程编程中的线程共享进程信息。在计算机编程中，多线程是一种常见的并发编程方式，允许程序在同一进程内创建多个线程，从而实现并发执行。由于这些线程共享同一进程的资源，包括内存空间和全局变量，因此可能会出现线程共享进程信息的现象。本实验旨在通过创建多个线程并使其共享进程信息，以便深入了解线程共享资源时可能出现的问题。

### 1.2.2 实验内容

- (1) 在进程中给一变量赋初值并成功创建两个线程；
- (2) 在两个线程中分别对此变量循环五千次以上做不同的操作（自行设计）并输出结果；
- (3) 多运行几遍程序观察运行结果，如果发现每次运行结果不同，请解释原因并修改程序解决，考虑如何控制互斥和同步；
- (4) 将任务一中第一个实验调用 `system` 函数和调用 `exec` 族函数改成在线程中实现，观察运行结果输出进程 PID 与线程 TID 进行比较并说明原因。

### 1.2.3 实验思想

本实验旨在通过创建两个线程，它们分别对一个共享的变量进行多次循环操作，并观察在多次运行实验时可能出现的不同结果。在观察到结果不稳定的情况下，引入互斥和同步机制来确保线程间的正确协同操作。

(1) 线程创建与变量操作：首先，在一个进程内创建两个线程，并在进程内部初始化一个共享的变量。这两个线程将并发地对这个共享变量进行循环操作，执行不同的操作。

(2) 竞态条件和不稳定结果：由于线程并发执行，存在竞态条件，即两个线程可能同时读取和修改共享变量。在没有适当的同步措施的情况下，不同线程的操作可能会交叉执行，导致结果不稳定，每次运行可能都会得到不同的结果。

(3) 互斥与同步：为了解决竞态条件带来的问题，可以使用互斥锁（Mutex）来保护共享变量的访问。在每个线程对变量进行操作之前，先获取互斥锁，操作完成后再释放锁。这样一来，每次只有一个线程能够访问变量，从而避免了并发访问带来的不稳定性。

(4) 观察结果与比较：运行多次实验，观察使用互斥锁后的运行结果。应该可以发现，通过互斥锁的保护，不再出现不稳定的结果，每次运行得到的结果都是一致的。

(5) 调用系统函数和线程函数的比较：在任务一中，如果将调用系统函数和调用 `exec` 族函数改成在线程中实现，观察运行结果。可以发现，调用系统函

数和 `exec` 族函数时，会输出进程的 PID（Process ID），而在线程中运行时，会输出线程的 TID（Thread ID）。这是因为线程是进程的子任务，它们共享进程的资源，但有自己的执行流程。

### 1.2.4 实验步骤

步骤一：设计程序，创建两个子线程，两线程分别对同一个共享变量多次操作，观察输出结果。

```
[root@kp-test01 lab1]# gcc 1-2.c -o 1-2 -lpthread
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: -318
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 89
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 0
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 0
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 210
[root@kp-test01 lab1]#
```

必须手动链接pthread库

可以发现，此时输出的结果并不 fixed。

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

int shared_variable = 0;

void *increment() {
    for (int i = 0; i < 5000; i++) {
        shared_variable++;
    }
    return NULL;
}

void *decrement() {
    for (int i = 0; i < 5000; i++) {
        shared_variable--;
    }
}
```

```

    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    if (pthread_create(&thread1, NULL, increment, NULL) != 0) {
        perror("Failed to create thread1");
        exit(1);
    }

    if (pthread_create(&thread2, NULL, decrement, NULL) != 0) {
        perror("Failed to create thread2");
        exit(1);
    }

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Final value of shared variable: %d\n", shared_variable);

    return 0;
}

```

结果不一致是因为程序没有进行线程同步。当多个线程访问和修改同一个共享变量 (*shared\_variable*) 而没有进行任何形式的同步时，程序的行为就变得不可预测。

一个线程正在尝试增加变量的值，而另一个线程正在尝试减少它。这两个操作可能几乎同时发生，因为操作系统的调度器可以随时中断一个线程并转而执行另一个线程。这样的话，两个线程可能会“看到”共享变量几乎同时的不同值，或者同时对它进行修改，从而导致不一致的结果。

考虑一个简单的情境：

1. *shared\_variable* 当前值为 0。
2. 线程 1 读取其值 (0)，准备加 1。

3. 线程 2 被调度，读取其值（还是 0），准备减 1。
4. 线程 1 继续执行，将值加 1，结果为 1。
5. 线程 2 继续执行，减 1（认为之前的值是 0），结果为 -1。

步骤二： 修改程序， 定义信号量 `signal`，使用 PV 操作实现共享变量的访问与互斥。运行程序，观察最终共享变量的值。

同步（Synchronization）在多线程编程中指的是协调多个线程的执行，以确保它们能够正常、可预测地访问共享资源或完成某些任务。互斥（Mutual Exclusion）是同步的一种特殊形式，确保一次只有一个线程能访问某个特定的资源或代码段。互斥通常通过互斥锁（Mutex）来实现，但它也可以通过其他机制来实现，比如信号量（Semaphore）。信号量和互斥锁类似，但更为通用。信号量可以用来解决除了互斥之外的其他同步问题。

PV 操作是信号量（Semaphores）操作的传统术语，源自荷兰语的 *Proberen*（尝试）和 *Verhogen*（增加）。在信号量的上下文中，P 操作通常用于申请或等待资源，而 V 操作用于释放或发出信号。

### **P 操作（也叫 *wait* 或 *down* 或 *sem\_wait*）**

- 当一个线程执行 P 操作时，它会检查信号量的值。
  - 如果信号量的值大于 0，那么它将减少信号量的值（通常是减 1）并继续执行。
  - 如果信号量的值为 0，线程将被阻塞，直到信号量的值变为大于 0。

### **V 操作（也叫 *signal* 或 *up* 或 *sem\_post*）**

- V 操作增加信号量的值（通常是加 1）。
- 如果有线程因执行 P 操作而阻塞在这个信号量上，一个或多个线程将被解除阻塞，并被允许减少信号量的值。

在 C 语言中，使用 POSIX 信号量

- `sem_wait(&semaphore);`: 执行 P 操作。
- `sem_post(&semaphore);`: 执行 V 操作。

其中，`semaphore` 是一个 `sem_t` 类型的变量，代表信号量。

`sem_init` 和 `sem_destroy` 是 POSIX 信号量 (Semaphores) 的初始化和销毁函数，它们用于设置和清理信号量。

### `sem_init`

这个函数用于初始化一个未命名的信号量。

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- **sem**: 一个指向信号量对象的指针。
- **pshared**: 如果这个参数是 0，信号量就是当前进程的局部信号量。如果这个参数非 0，则该信号量在多个进程间共享。
- **value**: 信号量的初始值。

这个函数成功时返回 0，失败时返回 -1。

### `sem_destroy`

这个函数用于销毁一个未命名的信号量，释放其占用的资源。

```
int sem_destroy(sem_t *sem);
```

- **sem**: 一个指向信号量对象的指针。

这个函数成功时返回 0，失败时返回 -1。在使用 `sem_destroy` 之前，确保没有线程被阻塞在该信号量上。

整体的代码为：

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

```

#include <pthread.h>
#include <semaphore.h>

int shared_variable = 0;

sem_t semaphore;

void *increment() {
    for (int i = 0; i < 5000; i++) {
        sem_wait(&semaphore);
        shared_variable++;
        sem_post(&semaphore);
    }
    return NULL;
}

void *decrement() {
    for (int i = 0; i < 5000; i++) {
        sem_wait(&semaphore);
        shared_variable--;
        sem_post(&semaphore);
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    if (sem_init(&semaphore, 0, 1) == -1) {
        perror("Failed to initialize semaphore");
        exit(1);
    }

    if (pthread_create(&thread1, NULL, increment, NULL) != 0) {
        perror("Failed to create thread1");
        exit(1);
    }

    if (pthread_create(&thread2, NULL, decrement, NULL) != 0) {
        perror("Failed to create thread2");
        exit(1);
    }

    pthread_join(thread1, NULL);

```

```

pthread_join(thread2, NULL);

sem_destroy(&semaphore);
printf("Final value of shared variable: %d\n", shared_variable);

return 0;
}

```

```

[root@kp-test01 lab1]# gcc 1-2-2.c -o 1-2 -lpthread -lrt
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 0
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 0
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 0
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 0
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 0
[root@kp-test01 lab1]#

```

步骤三： 在第一部分实验了解了 `system()` 与 `exec` 族函数的基础上，将这两个函数的调用改为在线程中实现，输出进程 PID 和线程的 TID 进行分析。

```

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

void *system_thread_function() {
    printf("System Thread TID: %ld\n", syscall(SYS_gettid));
    system("./system_call");
    return NULL;
}

void *exec_thread_function() {
    printf("Exec Thread TID: %ld\n", syscall(SYS_gettid));
    char *args[] = {"ls", "/usr/src", NULL};
    execvp(args[0], args);
    return NULL;
}

int main() {
    pthread_t system_thread, exec_thread;

```



```

printf("Main thread PID: %d\n", getpid());

if (pthread_create(&system_thread, NULL, system_thread_function,
NULL)) {
    fprintf(stderr, "Error creating system thread\n");
    return 1;
}

if (pthread_create(&exec_thread, NULL, exec_thread_function, NUL
L)) {
    fprintf(stderr, "Error creating exec thread\n");
    return 1;
}

pthread_join(system_thread, NULL);
pthread_join(exec_thread, NULL); // 注意, 如果exec_thread 成功运行
了, 这里实际上不会被执行

return 0;
}

```

```

[root@kp-test01 lab1]# This is system_call. My PID is: 10855
gcc 1-2-3.c -o 1-2 -lpt^C
[root@kp-test01 lab1]# ./1-2
Main thread PID: 10856
System Thread TID: 10857
Exec Thread TID: 10858
debug kernels lab1
[root@kp-test01 lab1]# This is system_call. My PID is: 10859
^C
[root@kp-test01 lab1]# |

```

我们会发现程序不会自动终止, 因为 exec 运行后使得该线程无法返回 NULL 终止。解决方案是在 exec 之前先 fork 一个进程, 在子进程中调用 exec

```

void *system_thread_function() {
    printf("System thread pid: %d\n", getpid());
    printf("System Thread TID: %ld\n", syscall(SYS_gettid));
    system("./system_call");
    return NULL;
}

```

```

void *exec_thread_function() {
    printf("exec thread pid: %d\n", getpid());
    printf("Exec Thread TID: %ld\n", syscall(SYS_gettid));
    pid_t pid = fork();
    if (pid == 0) {
        char *args[] = {"ls", "/usr/src", NULL};
        execvp(args[0], args);
        exit(0);
    } else if (pid > 0) {
        int status;
        waitpid(pid, &status, 0);
    } else {
        perror("fork");
    }
    return NULL;
}

```

```

[root@kp-test01 lab1]# gcc 1-2-3.c -o 1-2 -lpthread -lrt
[root@kp-test01 lab1]# ./1-2
Main thread PID: 11020
System thread pid: 11020
System Thread TID: 11021
exec thread pid: 11020
Exec Thread TID: 11022
debug kernels lab1
This is system_call. My PID is: 11023
[root@kp-test01 lab1]# ls /usr/src
debug kernels lab1
[root@kp-test01 lab1]# ./1-2
Main thread PID: 11026
System thread pid: 11026
System Thread TID: 11027
exec thread pid: 11026
Exec Thread TID: 11028
debug kernels lab1
This is system_call. My PID is: 11029
[root@kp-test01 lab1]# |

```

可以看到线程共享进程的 PID，但会有自身独立的线程 TID，同时 system 在实际过程中会调用 fork 但是这个行为被封装起来了

## 1.2.5 测试数据设计

无需数据测试。

## 1.2.6 程序运行初值及运行结果分析

运行结果已经分析。

## 1.2.7 实验总结

### 1.2.7.1 实验中的问题与解决过程

在本次实验中，我遇到了多个问题，这些问题涵盖了多线程编程、进程与线程的 ID 差异、以及程序中的同步和互斥等方面。

#### 线程创建和编译问题

首先，我试图在 Linux 环境中使用 C 语言创建两个线程，并对一个共享变量进行加一和减一操作。编译的过程中遇到了 *undefined reference to 'pthread\_create'* 等错误。这个问题是因为在链接阶段没有加上 *-lpthread* 标志，这个标志是必需的，因为它告诉编译器需要链接到 Pthreads 库。

#### 输出不一致性问题

接下来，我注意到多次运行同一个程序会产生不一致的输出。这是因为多个线程在访问和修改共享变量时没有进行同步，导致了竞态条件。解决这个问题的一个常见方法是使用互斥锁或信号量进行同步。

#### 信号量与同步机制

如何使用信号量（semaphore）进行同步以及 P（Proberen，测试）和 V（Verhogen，增加）操作的具体用法。信号量是用于控制多个线程对共享资源访问的一个同步原语。通过 PV 操作，我们可以确保在任何时刻只有一个线程能够访问共享变量，从而避免竞态条件。

## System 与 Exec 调用

我分析了如何在多线程环境中使用 `system()` 与 `exec` 族函数，并观察其对进程 ID 和线程 ID 的影响。注意到使用 `system()` 调用的进程 ID 与主线程不同，这是因为 `system()` 内部会使用 `fork()` 创建一个新的子进程来执行命令。

## 程序终止问题

我遇到了一个问题，即程序无法自动终止。这个问题出现在使用 `execvp()` 执行 `ls` 命令的部分。解决方法是在 `fork()` 创建的子进程中调用 `execvp()`，而父进程则通过 `waitpid()` 来等待子进程的完成。

综上所述，这些问题和解决方案都是多线程和多进程编程中常见的问题。每一步都需要仔细地考虑同步机制、系统调用和进程管理，以确保程序的正确性。我对这些细节有了深入的理解，这对我未来在并发编程方面将是非常有价值的经验。

### 1.2.7.2 实验收获

本次实验让我对多线程和多进程编程有了更深入的了解，特别是在 Linux 环境下使用 C 语言。掌握了如何创建和管理线程，以及如何使用信号量进行同步，

学习了 `system()` 和 `exec` 族函数在多线程环境下的行为，这对于理解进程和线程的区别非常有帮助。

我研究了多线程对共享资源访问的问题，尤其是竞态条件和其对程序输出的影响，成功地应用了互斥锁和信号量来解决这些问题，增强了程序的健壮性。

除了技术细节，实验还锻炼了我的问题解决能力。遇到编译错误、不一致的输出和程序终止问题时，都能够分析问题的根本原因，并找到相应的解决方案。这些都是软件开发中非常重要的技能。

### 1.2.7.3 意见与建议

1. **更多的实例操作：**尽管本次实验已经涵盖了多个重要的概念，但实践是检验真理的唯一标准。在未来的实验中，更多的实例操作可能会更有助于理解这些复杂的概念。
2. **代码审查与反思：**完成代码和实验后，进行代码审查和反思总结也是非常有价值的。这不仅能帮助你找出可能的错误或不足，还能促使思考如何优化和改进。
3. **深入理解同步机制：**本实验主要集中在使用信号量进行同步，但还有其他同步机制，如条件变量、读写锁等，也值得进一步研究。
4. **探索更多系统调用：**除了 `system()` 和 `exec` 族函数，还有很多其他有用的系统调用，如 `pipe()`、`dup()` 等，这些都是进程和线程间通信的重要手段。

## 1.2.8 附件

### 1.2.8.1 附件 1 程序

*1-2-1.c*

*1-2-2.c*

*1-2-3.c*

### 1.2.8.2 附件 2 Readme

*lab1-2Readme.md*

## 1.3 自旋锁实验

### 1.3.1 实验目的

自旋锁作为一种并发控制机制，可以在特定情况下提高多线程程序的性能。本实验旨在通过设计一个多线程的实验环境，以及使用自旋锁来实现线程间的同步，从而实现以下目标：

- (1) 了解自旋锁的基本概念：通过研究自旋锁的工作原理和特点，深入理解自旋锁相对于其他锁机制的优势和局限性；
- (2) 实验自旋锁的应用：在一个多线程的实验环境中，设计一个竞争资源的场景，让多个线程同时竞争对该资源的访问；
- (3) 实现自旋锁的同步：使用自旋锁来保护竞争资源的访问，确保同一时间只有一个线程可以访问该资源，避免数据不一致和竞态条件；

### 1.3.2 实验内容

- (1) 在进程中给一变量赋初值并成功创建两个线程；
- (2) 在两个线程中分别对此变量循环五千次以上做不同的操作（自行设计）并输出结果；
- (3) 使用自旋锁实现互斥和同步；

### 1.3.3 实验思想

自旋锁是一种基于忙等待（busy-waiting）的同步机制，用于在线程竞争共享资源时，不断尝试获取锁，而不是阻塞等待。它的工作原理可以简单地概括为以下几个步骤：

（1）初始化锁：自旋锁的开始是一个共享的标志变量（flag），最初为未锁定状态（0）。这个标志变量用于表示资源是否已被其他线程占用。

（2）获取锁：当一个线程尝试获取锁时，它会循环检查标志变量的状态。如果发现标志变量是未锁定状态（0），那么该线程将通过原子操作将标志变量设置为锁定状态（1），从而成功获取锁。如果标志变量已经是锁定状态，线程会一直在循环中等待，直到标志变量变为未锁定状态为止。

（3）释放锁：当持有锁的线程完成对共享资源的操作后，它会通过原子操作将标志变量设置回未锁定状态（0），从而释放锁，允许其他等待的线程尝试获取锁。

自旋锁的工作原理中关键的部分在于“自旋”这一概念，即等待获取锁的线程会循环忙等待，不断检查标志变量的状态，直到能够成功获取锁。这种方式在锁的占用时间很短的情况下可以减少线程切换的开销，提高程序性能。

### 1.3.4 实验步骤

步骤一：根据实验内容要求，编写模拟自旋锁程序代码 spinlock.c，待补充主函数的示例代码如下：

```
/**spinlock.c*in xjtu*2023.8  
*/
```

```

#include <stdio.h>#include <pthread.h>// 定义自旋锁结构体 typedef struc
t {
int flag;} spinlock_t;
// 初始化自旋锁
void spinlock_init(spinlock_t *lock) {lock->flag = 0;
}
// 获取自旋锁
void spinlock_lock(spinlock_t *lock) {
while (__sync_lock_test_and_set(&lock->flag, 1)) {// 自旋等待
}}
// 释放自旋锁
void spinlock_unlock(spinlock_t *lock) {__sync_lock_release(&lock->f
lag);
}
// 共享变量int shared_value = 0;
// 线程函数
void *thread_function(void *arg) {spinlock_t *lock = (spinlock_t *)ar
g;for (int i = 0; i < 5000; ++i) {
spinlock_lock(lock);shared_value++;spinlock_unlock(lock);
}
return NULL;}
int main() {
pthread_t thread1, thread2;
spinlock_t lock;// 输出共享变量的值
// 初始化自旋锁
// 创建两个线程
// 等待线程结束
// 输出共享变量的值return 0;
}

```

完整代码为：

```

/**spinlock.c*in xjtu*2023.8
*/
#include <pthread.h> // 定义自旋锁结构体
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int flag;
} spinlock_t;
// 初始化自旋锁
void spinlock_init(spinlock_t *lock) { lock->flag = 0; }
// 获取自旋锁

```



```

void spinlock_lock(spinlock_t *lock) {
    while (__sync_lock_test_and_set(&lock->flag, 1)) { // 自旋等待
    }
}
// 释放自旋锁
void spinlock_unlock(spinlock_t *lock) { __sync_lock_release(&lock->flag); }
// 共享变量
int shared_value = 0;
// 线程函数
void *thread_function(void *arg) {
    spinlock_t *lock = (spinlock_t *)arg;
    for (int i = 0; i < 5000; ++i) {
        spinlock_lock(lock);
        shared_value++;
        spinlock_unlock(lock);
    }
    return NULL;
}
int main() {
    pthread_t thread1, thread2;
    spinlock_t lock; // 输出共享变量的值
    spinlock_init(&lock);

    if (pthread_create(&thread1, NULL, thread_function, &lock) != 0) {
        perror("Failed to create thread1");
        exit(1);
    }
    if (pthread_create(&thread2, NULL, thread_function, &lock) != 0) {
        perror("Failed to create thread2");
        exit(1);
    }

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("shared value is %d\n", shared_value);

    return 0;
}

```

补充完成代码后，编译并运行程序，分析运行结果

```
[root@kp-test01 lab1]# gcc 1-3-1.c -o 1-3 -lpthread
[root@kp-test01 lab1]# ./1-3
shared value is 10000
[root@kp-test01 lab1]# ./1-3
shared value is 10000
[root@kp-test01 lab1]# ./1-3
shared value is 10000
[root@kp-test01 lab1]# |
```

自旋锁设定成功，每个线程都正确地对 `shared_value` 进行了修改

### 1.3.5 测试数据设计

无

### 1.3.6 程序运行初值及运行结果分析

`shared_value` 初始值为 0

运行后获得了正确的结果：10000，这说明自旋锁设定有效

### 1.3.7 实验总结

#### 1.3.7.1 实验中的问题与解决过程

在实验过程中，我遇到了多个问题。其中一个重要的问题是 Segmentation fault（段错误）。这一问题是由于在创建线程时未将自旋锁作为参数传入线程函数，导致线程函数试图访问一个空指针，从而触发了段错误。经过仔细分析代码和逻辑，发现应将自旋锁的地址作为 `pthread_create` 的第四个参数传入，解决了这一问题。

另一个问题是编译警告，关于 `exit` 函数未声明。这个问题的解决相对简单，仅需要包含相应的头文件 `<stdlib.h>` 即可。

### 1.3.7.2 实验收获

1. **提高了问题解决能力：**遇到的问题和解决过程锻炼了分析问题和调试代码的能力，尤其是如何定位和修复内存访问相关的错误。
2. **加强了实际操作能力：**不仅仅是理论知识，还有实际将理论应用到代码中的能力也得到了加强。

### 1.3.7.3 意见与建议

1. **注意代码的健壮性：**在编写代码时，应考虑各种边界条件和异常情况，例如空指针、非法输入等。
2. **充分利用编译器：**使用编译器的警告和错误信息作为调试的线索，而不仅仅是编译的障碍。
3. **注重代码可读性：**代码是给人看的，其次才是给机器执行。应确保代码结构清晰，命名规范，同时也要充分注释。
4. **持续学习和实践：**技术是日新月异的，通过不断的学习和实践，才能跟上技术的步伐，更好地应对各种问题。

## 1.3.8 附件

### 1.3.8.1 附件 1 程序

*1-3.c*

### 1.3.8.2 附件 2 Readme

*lab1-3Readme.md*

## 实验二 进程通信与内存管理

### 2.1 进程的软中断通信

#### 2.1.1 实验目的

编程实现进程的创建和软中断通信，通过观察、分析实验现象，深入理解进程及进程在调度执行和内存空间等方面的特点，掌握在 POSIX 规范中系统调用的功能和使用。

#### 2.1.2 实验内容

(1) 使用 man 命令查看 fork 、 kill 、 signal、 sleep、 exit 系统调用的帮助手册。

(2) 根据流程图（如图 2.1 所示） 编制实现软中断通信的程序： 使用系统调用 fork()创建两个子进程，再用系统调用 signal()让父进程捕捉键盘上发出的中断信号（即 5s 内按下 delete 键或 quit 键），当父进程接收到这两个软中断的某一个后，父进程用系统调用 kill()向两个子进程分别发出整数值为 16 和 17 软中断信号，子进程获得对应软中断信号，然后分别输出下列信息后终止：

Child process 1 is killed by parent !! Child process 2 is killed by parent !!

父进程调用 wait()函数等待两个子进程终止后，输出以下信息，结束进程执行： Parent process is killed!!

注： delete 会向进程发送 SIGINT 信号， quit 会向进程发送 SIGQUIT 信号。 ctrl+c 为 delete， ctrl+\为 quit 。

(3) 多次运行所写程序, 比较 5s 内按下 Ctrl+\或 Ctrl+Delete 发送中断, 或 5s 内不进行任何操作发送中断, 分别会出现什么结果? 分析原因。

(4) 将本实验中通信产生的中断通过 14 号信号值进行闹钟中断, 体会不同中断的执行样式, 从而对软中断机制有一个更好的理解

### 2.1.3 实验前准备

实验相关 UNIX 系统调用介绍:

(1)fork(): 创建一个子进程。

创建的子进程是 fork 调用者进程(即父进程)的复制品,即进程映像.除了进程标识数以及与进程特性有关的一些参数外,其他都与父进程相同,与父进程共享文本段和打开的文件,并都受进程调度程序的调度。

如果创建进程失败,则 fork()返回值为 -1,若创建成功,则从父进程返回值是子进程号,从子进程返回的值是 0。

因为 FORK 会将调用进程的所有内容原封不动地拷贝到新创建的子进程中去,而如果之后马上调用 exec,这些拷贝的东西又会马上抹掉,非常不划算.于是设计了一种叫作“写时拷贝”的技术,使得 fork 结束后并不马上复制父进程的内容,而是到了真正要用的时候才复制。

(2)exec(): 装入并执行相应文件。

(3)wait():父进程处于阻塞状态,等待子进程终止,其返回值为所等待子进程的进程号。

(4)exit():进程自我终止,释放所占资源,通知父进程可以删除自己,此时它的状态变为 P\_state= SZOMB,即僵死状态.如果调用进程在执行 exit 时其父进程正在等待它的中止,则父进程可立即得到该子进程的 ID 号。

(5)getpid():获得进程号。

(6)lockf(files,function,size):用于锁定文件的某些段或整个文件。本函数适用的头文件为: #include<unistd.h>,

参数定义: int lockf(files,function,size) int files,function;

long size;

files 是文件描述符, function 表示锁状态, 1 表示锁定, 0 表示解锁;

size 是锁定或解锁的字节数, 若为 0 则表示从文件的当前位置到文件尾。

(7)kill(pid,sig): 一个进程向同一用户的其他进程 pid 发送一中断信号。

(8)signal(sig,function): 捕捉中断信号 sig 后执行 function 规定的操作。

头文件为: #include <signal.h>

参数定义: signal(sig,function) int sig;

void (\*func) ();其中 sig 共有 19 个值

注意: signal 函数会修改进程对特定信号的响应方式。

(9)pipe(fd);

int fd[2];

其中 fd[1]是写端，向管道中写入， fd[0]是读端，从管道中读出。

(10)暂停一段时间 sleep;

调用 sleep 将在指定的时间 seconds 内挂起本进程。

其调用格式为：“unsigned sleep(unsigned seconds);”；返回值为实际的挂起时间。

(11)暂停并等待信号 pause;

调用 pause 挂起本进程以等待信号，接收到信号后恢复执行。当接收到中止进程信号时，该调用不再返回。

其调用格式为“int pause(void);”

在 linux 系统下，我们可以输入 kill -l 来观察所有的信号以及对应的编号

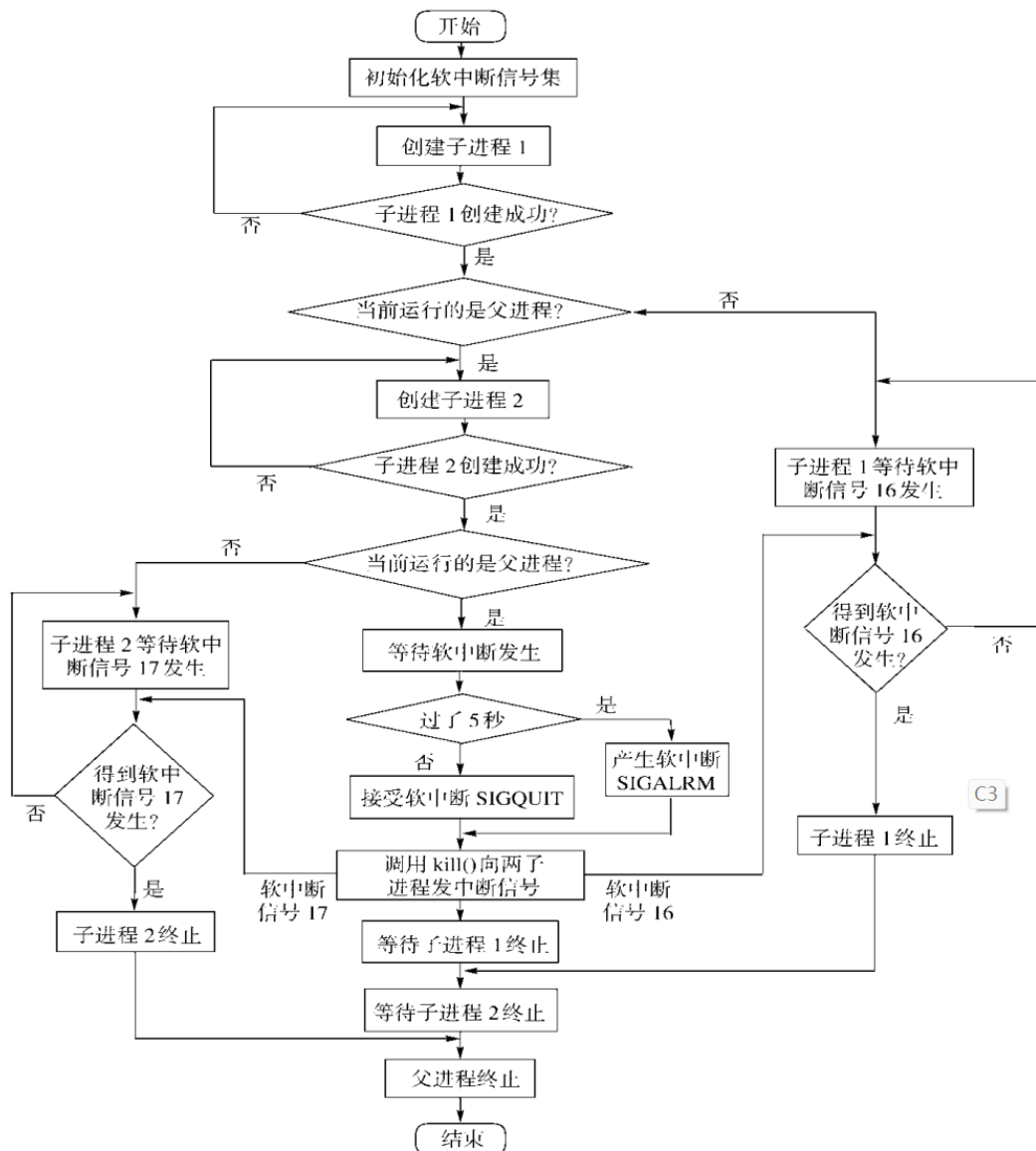


图 2.1 软中断通信程序流程图

编写实验代码需要考虑的问题：

- (1)父进程向子进程发送信号时，如何确保子进程已经准备好接收信号？
- (2)如何阻塞住子进程，让子进程等待父进程发来信号？



## 2.1.4 实验步骤

了解 kill、wait 等系统函数，编写代码，编译运行，观察输出。

## 2.1.5 程序运行初值及运行结果分析

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

void signal_handler(int signal_num) {
    // printf("Received interrupt signal: %d\n", signal_num);
}

void child_signal_handler(int signal_num) {
    if (signal_num == 16) {
        printf("Child process 1 is killed by parent !!\n");
    }
    if (signal_num == 17) {
        printf("Child process 2 is killed by parent !!\n");
    }
    exit(0);
}

int main() {
    pid_t child1, child2;
    signal(SIGINT, signal_handler); // (Ctrl+C)
    signal(SIGQUIT, signal_handler); // (Ctrl+\)

    child1 = fork();
    if (child1 == 0) {
        signal(16, child_signal_handler);
        while (1) {
        }
    }

    child2 = fork();
    if (child2 == 0) {
        signal(17, child_signal_handler);
        while (1) {
        }
    }
}
```

```

}

if (child1 > 0 && child2 > 0) {
    alarm(5);
    pause();

    kill(child1, 16);
    kill(child2, 17);

    // wait two child process to exit
    wait(NULL);
    wait(NULL);
    printf("Parent process is killed!!\n");
}

return 0;
}

```

```

cold@ubuntu:~/Desktop/xjtuOSlab$ ./2.1.1
^CChild process 1 is killed by parent !!
Child process 2 is killed by parent !!
Parent process is killed!!
cold@ubuntu:~/Desktop/xjtuOSlab$ ./2.1.1
^\Child process 2 is killed by parent !!
Child process 1 is killed by parent !!
Parent process is killed!!
cold@ubuntu:~/Desktop/xjtuOSlab$ ./2.1.1
Alarm clock
cold@ubuntu:~/Desktop/xjtuOSlab$ █

```

观察三次输出可以看到，子进程结束的顺序并不固定，如果超时会通过 alarm 来退出。这符合我一开始对于程序的设计。因为超时没有传递中断或者退出子进程就无法被结束。（最重要的是 alarm 信号没有被处理。）现在我希望改进代码，我增加一个 alarm\_handle:

```

void signal_handler(int signal_num) {
    if (signal_num == SIGALRM) {
        printf("the signal is timeout proc stopped auto!!\n");
    }
}

```

```
}  
// printf("Received interrupt signal: %d\n", signal_num);  
}
```

```
Parent process is killed!!  
cold@ubuntu:~/Desktop/xjtuOSlab$ gcc 2.1.1.c -o 2.1.1  
cold@ubuntu:~/Desktop/xjtuOSlab$ ./2.1.1  
the signal is timeout proc stopped auto!!  
Child process 1 is killed by parent !!  
Child process 2 is killed by parent !!  
Parent process is killed!!
```

此时我发现实验手册要求我使用闹钟中断——alarm，我在前面已经实现完毕。

只能说：无巧不成书！

kill 命令我只调用了两次。调用后一方面是中断了对应的子进程，另一方面发出了一个 signal，输出相关信息。

（5）使用 kill 命令可以在进程的外部杀死进程。进程怎样能主动退出？这两种退出方式哪种更好一些？

对于这个问题，我认为可以在进程内调用 exit(0)、return 等方式来退出。后者更好一些。kill 我理解的类似于 Windows 平台设备管理器的“结束进程”，或者说强制关机，这是一个具有风险的行为，尤其是对于易失性的信息，很可能直接丢失。因此 kill 不到万不得已不太应该使用。

## 2.1.6 实验总结

### 2.1.6.1 实验中的问题与解决过程

不了解 kill、signal 相关的调用。查询相关文档学习。

参考代码复杂且难懂，难以完成，最终通过自主重构完成。

### 2.1.6.2 实验收获

我们总是会遇到很多以目前知识无法立刻解决的难题，但是只要保持自信，勇敢挑战自己，不断学习，就能解决难题。

### 2.1.6.3 意见与建议

增加一些验收形式。目前的实验验收形式过于落后。效率极低。在目前实验本身价值有限的情况下，浪费了同学更浪费了老师和助教的大量时间。

## 2.1.7 附件

### 2.1.7.1 附件 1 程序

*2.1.1.c*

### 2.1.7.2 附件 2 Readme

*lab2-1Readme.md*

## 2.2 进程的管道通信

### 2.2.1 实验目的

编程实现进程的管道通信，通过观察、分析实验现象，深入理解进程管道通信的特点，掌握管道通信的同步和互斥机制。



程), 则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的, 故又称为管道通信。这种方式首创于 UNIX 系统, 由于它能有效地传送大量数据, 因而又被引入到许多其它操作系统中。

为了协调双方的通信, 管道机制必须提供以下三方面的协调能力:

①互斥, 即当一个进程正在对 pipe 执行读/写操作时, 其它(另一)进程必须等待。

②同步, 指当写(输入)进程把一定数量(如 4KB)的数据写入 pipe, 便去睡眠等待, 直到读(输出)进程取走数据后, 再把他唤醒。当读进程读一空 pipe 时, 也应睡眠等待, 直至写进程将数据写入管道后, 才将之唤醒。

③确定对方是否存在, 只有确定了对方已存在时, 才能进行通信。

管道是进程间通信的一种简单易用的方法。管道分为匿名管道和命名管道两种。下面首先介绍匿名管道。

匿名管道只能用于父子进程之间的通信, 它的创建使用系统调用 `pipe()`: `int pipe(int fd[2])`

其中的参数 `fd` 用于描述管道的两端, 其中 `fd[0]`是读端, `fd[1]`是写端。两个进程分别使用

读端和写端, 就可以进行通信了。

一个父子进程间使用匿名管道通信的例子。

匿名管道只能用于父子进程之间的通信，而命名管道可以用于任何管道之间的通信。命名管道实际上就是一个 FIFO 文件，具有普通文件的所有性质，用 ls 命令也可以列表。但是，它只是一块内存缓冲区。

```
/*管道通信实验程序残缺版 */
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
int pid1, pid2; // 定义两个进程变量
int main() {
    int fd[2];
    char InPipe[1000]; // 定义读缓冲区
    char c1 = '1', c2 = '2';
    pipe(fd); // 创建管道
    while ((pid1 = fork()) == -1)
        ; // 如果进程 1 创建不成功,则空循环
        // 如果子进程 1 创建成功,pid1 为进程号
    if (pid1 == 0) {
        // 补充:锁定管道
        // 补充:分 2000 次每次向管道写入字符'1'
        sleep(5); // 等待读进程读出数据
        // 补充:解除管道的锁定
        exit(0); // 结束进程 1
    } else {
        while ((pid2 = fork()) == -1)
            ; // 若进程 2 创建不成功,则空循环
        if (pid2 == 0) {
            lockf(fd[1], 1, 0);
            // 补充:分 2000 次每次向管道写入字符'2'
            sleep(5);
            lockf(fd[1], 0, 0);
            exit(0);
        } else {
            // 补充:等待子进程 1 结束
            wait(0); // 等待子进程 2 结束
            // 补充:从管道中读出 4000 个字符
            // 补充:加字符串结束符
            printf("%s\n", InPipe); // 显示读出的数据
            exit(0); // 父进程结束
        }
    }
}
```

## 2.2.4 实验步骤

根据残缺代码完善。基于我自己的知识：

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int pid1, pid2;

int main() {
    int fd[2];
    char InPipe[1000];
    char c1 = '1', c2 = '2';
    pipe(fd);

    while ((pid1 = fork()) == -1)
        ; // 如果进程 1 创建不成功,则空循环
        // 如果子进程 1 创建成功,pid1 为进程号
    if (pid1 == 0) {
        lockf(fd[1], 1, 0); // 锁定管道 0 para means lock until the end
        for (int i = 0; i < 2000; i++) {
            write(fd[1], &c1, 1);
        } // 分 2000 次每次向管道写入字符'1'
        sleep(5); // 等待读进程读出数据
        lockf(fd[1], 0, 0); // 解除管道的锁定
        exit(0); // 结束进程 1
    } else {
        while ((pid2 = fork()) == -1)
            ; // 若进程 2 创建不成功,则空循环
        if (pid2 == 0) {
            lockf(fd[1], 1, 0);
            for (int i = 0; i < 2000; i++) {
                write(fd[1], &c2, 1);
            } // 分 2000 次每次向管道写入字符'2'
            sleep(5);
            lockf(fd[1], 0, 0);
            exit(0);
        } else {
            wait(NULL); // 等待子进程 1 结束
        }
    }
}
```



[illegible][illegible]

## 2.2.5 运行结果分析

linux.die.net/man/3/lockf

### lockf(3) - Linux man page

---

**Name**

lockf - apply, test or remove a POSIX lock on an open file

**Synopsis**

```
#include <unistd.h>

int lockf(int fd, int cmd, off_t len);
```

Feature Test Macro Requirements for glibc (see [feature test macros\(7\)](#)):

```
lockf():
    _BSD_SOURCE || _SVID_SOURCE ||
    _XOPEN_SOURCE >= 500 ||
    _XOPEN_SOURCE &&
    _XOPEN_SOURCE_EXTENDED
```

**Description**

Apply, test or remove a POSIX lock on a section of an open file. The file is specified by *fd*, a file descriptor open for writing, the action by *cmd*, and the section consists of byte positions *pos*..*pos+len-1* if *len* is positive, and *pos-len*..*pos-1* if *len* is negative, where *pos* is the current file position, and if *len* is zero, the section extends from the current file position to infinity, encompassing the present and future end-of-file positions. In all cases, the section may extend past current end-of-file.

On Linux, **lockf()** is just an interface on top of **fcntl(2)** locking. Many other systems implement **lockf()** in this way, but note that POSIX.1-2001 leaves the relationship between **lockf()** and **fcntl(2)** locks unspecified. A portable application should probably avoid mixing calls to these interfaces.

Valid operations are given below:

lockf 是一种 pthread\_mutex lock，它可以保证锁定的管道只能被当前 thread 操作，如果一个 thread 尝试调用一个锁定的管道，它会进入等待状态。

## 2.2.6 实验总结

### 2.2.6.1 实验中的问题与解决过程

无

### 2.2.6.2 实验收获

5. **互斥锁的重要性**：通过这次实验，明确了互斥锁在多进程同步中的重要性。

6. **管道的同步与互斥**：理解了如何通过管道来进行进程间的通信，并通过互斥锁来保证管道的同步与互斥。

### 2.2.6.3 意见与建议

7. **更多的实践**：虽然本次实验提供了对管道和进程同步的基础理解，但更复杂的场景需要进一步的实验来探索。
8. **深入理解 API 函数**：*lockf*, *wait*, *pipe* 等都是非常有用的系统调用，值得深入理解其背后的工作机制。

## 2.2.7 附件

### 2.2.7.1 附件 1 程序

*2.2.c*

### 2.2.7.2 附件 2 Readme

*lab2-2Readme.md*

## 2.3 内存的分配与回收

### 2.3.1 实验目的

通过设计实现内存分配管理的三种算法（FF，BF，WF），理解内存分配及回收的过程及实现思路，理解如何提高内存的分配效率和利用率。

### 2.3.2 实验内容

- （1）理解内存分配 FF，BF，WF 策略及实现的思路。
- （2）参考给出的代码思路，定义相应的数据结构，实现上述 3 种算法。每种算法要实现内存分配、回收、空闲块排序以及合并、紧缩等功能。

(3) 充分模拟三种算法的实现过程，并通过对比，分析三种算法的优劣。

### 2.3.3 实验前准备

关于 OS 的连续内存管理知识

#### (1) 系统区与用户区

内存管理的基础是对内存的划分。最简单的划分就是每部分仅包含连续的内存区域，这样的区域称为分区。内存一般被划分为两部分：操作系统分区和用户进程分区。当系统中同时运行多个进程时，每个进程都需要有自己独占的分区，这样就形成了多个用户进程分区。每个分区都是连续的，作为进程访问的物理内存，其位置和大小可以由基址寄存器和界限寄存器唯一确定。通过这两个寄存器，系统可以将一个进程的逻辑地址空间映射到其物理地址空间，整个系统仅需要一对这样的寄存器用于当前正在执行的进程。进程分区的起址和大小是保存在进程控制块中的，仅在进程运行时才会装载到基址和界限寄存器中。

操作系统内存管理的任务就是为每个进程分配内存分区，并将进程代码和数据装入分区。每当进程被调度，执行前由操作系统为其设置好基址和界限寄存器。进程在执行过程中 CPU 会依据这两个寄存器的值进行地址转换，得到要访问的物理地址。进程执行结束并退出内存后，操作系统回收进程所占的分区。

下面讨论三种连续内存管理的特点及管理方法。

#### (2) 连续内存管理方法

##### 1) 单一连续区分配

这种分配方式仅适用于单用户单任务的系统，它把内存分为系统区和用户区。系统区仅供 OS 使用，用户区供用户使用，任意时刻内存中只能装入一道程序。

## 2) 固定分区分配

固定分区分配将用户内存空间划分为若干个固定大小的区域，在每个用户分区中可以装入一个用户进程。内存的用户区被划分成几个分区，便允许几个进程驻留内存。操作系统为了完成对固定分区的管理，必须定义一个记录用户分区大小、分区起始地址及分区是否空闲的数据结构。

## 3) 动态分区分配

这种分配方式根据用户进程的大小，动态地对内存进行划分，根据进程需要的空间大小分配内存。内存中分区的大小和数量是变化的。动态分区方式比固定分区方式显著地提高了内存利用率。

操作系统刚启动时，内存中仅有操作系统分区和一个空闲分区。随着进程不断运行和退出，原始的空闲分区被分割成了大量的进程分区和不相邻的空闲分区。当一个新的进程申请内存时，系统为其分配一个足够大的空闲分区，当一个进程结束时，系统回收进程所占内存。采用动态分区分配方式时，通常可以建立一个空闲分区链以管理空闲的内存区域。

一般不会存在一个空闲分区，其大小正好等于需装入的进程的大小。操作系统不得不把一个大的空闲分区进行拆分后分配给新进程，剩下的放入空闲分区

表。空闲分区表中可能有多个大于待装入进程的分区，应该按照什么策略选择分区，会影响内存的利用率。

### （3）常见的动态分区分配算法

#### 1) 首次适应算法。

在采用空闲分区链作为数据结构时，该算法要求空闲分区链表以地址递增的次序链接。在进行内存分配时，从链首开始顺序查找，直至找到一个能满足进程大小要求的空闲分区为止。然后，再按照进程请求内存的大小，从该分区中划出一块内存空间分配给请求进程，余下的空闲分区仍留在空闲链中。

#### 2) 循环首次适应算法。

该算法是由首次适应算法演变而形成的，在为进程分配内存空间时，从上次找到的空闲分区的下一个空闲分区开始查找，直至找到第一个能满足要求的空闲分区，并从中划出一块与请求的大小相等的内存空间分配给进程。

#### 3) 最佳适应算法。

将空闲分区链表按分区大小由小到大排序，在链表中查找第一个满足要求的分区。

#### 4) 最差匹配算法。

将空闲分区链表按分区大小由大到小排序，在链表中找到第一个满足要求的空闲分区。

### （4）动态分区的回收

内存分区回收的任务是释放被占用的内存区域，如果被释放的内存空间与其它空闲分区在地址上相邻接，还需要进行空间合并，分区回收流程如下：

- 1) 释放一块连续的内存区域。
- 2) 如果被释放区域与其它空闲区间相邻，则合并空闲区。
- 3) 修改空闲分区链表。

如果被释放的内存区域（回收区）与任何其它的空闲区都不相邻，则为该回收区建立一个空闲区链表的结点，使新建结点的起始地址字段等于回收区起始地址，空闲分区大小字段等于回收区大小，根据内存分配程序使用的算法要求（按地址递增顺序或按空闲分区大小由小到大排序），把新建结点插入空闲分区链表的适当位置。

如果被释放区域与其它空闲区间相邻，需要进行空间合并，在进行空间合并时需要考虑以下三种不同的情况：

- 1) 仅回收区的前面有相邻的空闲分区。如图 3-7a 所示，把回收区与空闲分区 R1 合并成一个空闲分区，把空闲链表中与 R1 对应的结点的分区起始地址作为新空闲区的起始地址，将该结点的分区大小字段修改为空闲分区 R1 与回收区大小之和。
- 2) 仅回收区的后面有相邻的空闲分区。如图 3-7b 所示，把回收区与空闲分区 R2 合并成一个空闲分区，把空闲链表中与 R2 对应的结点的分区起始地址改为回收区起始地址，将该结点的分区大小字段修改为空闲分区 R2 与回收区大小之和。

3) 回收区的前、后都有相邻的空闲分区。如图 3-7c 所示，把回收区与空闲分区 R1、R2 合并成一个空闲分区，把空闲链表中与 R1 对应的结点的分区起始地址作为合并后新空闲分区的起始地址，将该结点的分区大小字段修改为空闲分区 R1、R2 与回收区三者大小之和，删去与 R2 分区对应的空闲分区结点。当然，也可以修改分区 R2 对应的结点，而删去 R1 对应的结点。还可以为新合并的空闲分区建立一个新的结点，插入空闲分区链表，删除 R1 和 R2 对应的分区结点。

#### (5) 内存碎片

一个空闲分区被分配给进程后，剩下的空闲区域有可能很小，不可能再分配给其他的进程，这样的小空闲区域称为内存碎片。最坏情况下碎片的数量会与进程分区的数量相同。大量碎片会降低内存的利用率，因此如何减少碎片就成为分区管理的关键问题。内存中的碎片太多时，可以通过移动分区将碎片集中，形成大的空闲分区。这种方法的系统开销显然很大，而且随着进程不断运行或退出，新的碎片很快就会产生。当然，回收分区时合并分区也会消除一些碎片。

### 2.3.4 实验步骤

- 明确主要功能
  - 1 - Set memory size (default=1024)
  - 2 - Select memory allocation algorithm
  - 3 - New process
  - 4 - Terminate a process
  - 5 - Display memory usage
  - 0 - Exit



通过键盘输入选择。

- 主要的基础数据结构

#### 1) 内存空闲分区的描述

```
/*描述每一个空闲块的数据结构*/ struct free_block_type {  
    int size;  
    int start_addr;  
  
    struct free_block_type *next;  
};  
  
/*指向内存中空闲块链表的首指针*/ struct free_block_type *free_block;
```

#### 2) 描述已分配的内存块

```
/*每个进程分配到的内存块的描述*/ struct allocated_block {  
    int pid;  
    int size;  
  
    int start_addr;  
  
    char process_name[PROCESS_NAME_LEN];  
    struct allocated_block *next;  
};  
  
/*进程分配内存块链表的首指针*/  
  
struct allocated_block *allocated_block_head = NULL;
```

#### 3) 常量定义

```
#define PROCESS_NAME_LEN 32 /*进程名长度*/  
#define MIN_SLICE 10 /*最小碎片的大小*/  
#define DEFAULT_MEM_SIZE 1024 /*内存大小*/  
#define DEFAULT_MEM_START 0 /*起始位置*/  
/* 内存分配算法 */  
#define MA_FF 1  
#define MA_BF 2  
#define MA_WF 3  
int mem_size = DEFAULT_MEM_SIZE; /*内存大小*/  
int ma_algorithm = MA_FF; /*当前分配算法*/  
static int pid = 0; /*初始 pid*/  
int flag = 0; /*设置内存大小标志*/
```

- 主要模块

```
int main() {
    char choice;
    pid = 0;
    free_block = init_free_block(mem_size); // 初始化空闲区
    while (1) {
        display_menu();
        fflush(stdin);
        choice = getchar();
        switch (choice) {
            case '1':
                set_mem_size();
                break;
            case '2':
                set_algorithm();
                flag = 1;
                break;
            case '3':
                new_process();
                flag = 1;
                break;
            case '4':
                kill_process();
                flag = 1;
                break;
            case '5':
                display_mem_usage();
                flag = 1;
                break;
            case '0':
                do_exit();
                exit(0);
            default:
                break;
        }
    }
}
```

*/初始化空闲块, 默认为一块, 可以指定大小及起始地址/*

```
struct free_block_type *init_free_block(int mem_size) {
    struct free_block_type *fb;
    fb = (struct free_block_type *)malloc(sizeof(struct free_block_type));
}
```

```

    if (fb == NULL) {
        printf("No mem\n");
        return NULL;
    }
    fb->size = mem_size;
    fb->start_addr = DEFAULT_MEM_START;
    fb->next = NULL;
    return fb;
}

/显示菜单/

void display_menu() {
    printf("\n");
    printf("1 - Set memory size (default=%d)\n", DEFAULT_MEM_SIZ
E);
    printf("2 - Select memory allocation algorithm\n");
    printf("3 - New process \n");
    printf("4 - Terminate a process \n");
    printf("5 - Display memory usage \n");
    printf("0 - Exit\n");
}

/设置内存的大小/

void set_mem_size() {
    int size;
    if (flag != 0) {
        printf("Cannot set memory size again\n");
        return;
    }
    printf("set memory_size to: ");
    scanf("%d", &size);
    if (size > 0) {
        mem_size = size;
        free_block->size = mem_size;
    }
    flag = 1;
}

/* 设置当前的分配算法 */

void set_algorithm() {
    int algorithm;
    printf("\t1 - First Fit\n");
    printf("\t2 - Best Fit \n");

```

```

printf("\t3 - Worst Fit \n");
scanf("%d", &algorithm);
if (algorithm >= MA_FF && algorithm <= MA_WF) {
    ma_algorithm = algorithm;
}
rearrange(ma_algorithm);
}

/按指定的算法整理内存空闲块链表/

void rearrange(int algorithm) {
    switch (algorithm) {
        case MA_FF:
            rearrange_FF();
            break;
        case MA_BF:
            rearrange_BF();
            break;
        case MA_WF:
            rearrange_WF();
            break;
        default:
            printf("Invalid algorithm.\n");
    }
}

/按 FF 算法重新整理内存空闲块链表/

/按 BF 算法重新整理内存空闲块链表/

/按 WF 算法重新整理内存空闲块链表/

/创建新的进程，主要是获取内存的申请数量/

/分配内存模块/

/删除进程，归还分配的存储空间，并删除描述该进程内存分配的节点/

/释放 ab 数据结构节点/

/* 显示当前内存的使用情况，包括空闲区的情况和已经分配的情况 */

void display_mem_usage() {
    struct free_block_type *fbt = free_block;
    struct allocated_block *ab = allocated_block_head;

    if (fbt == NULL) {
        printf("No free memory blocks\n");
    }
}

```

```

        return;
    }

    printf("-----\n");
    printf("Free Memory:\n");
    printf("%20s %20s\n", "start_addr", "size");
    while (fbt != NULL) {
        printf("%20d %20d\n", fbt->start_addr, fbt->size);
        fbt = fbt->next;
    }
    printf("\nUsed Memory:\n");
    printf("%10s %20s %10s %10s\n", "PID", "ProcessName", "start_
addr", "size");
    while (ab != NULL) {
        printf("%10d %20s %10d %10d\n", ab->pid, ab->process_name,
ab->start_addr,
        ab->size);
        ab = ab->next;
    }

    printf("-----\n");
}

```

完善补充各个模块代码，最终构建成功正确的项目代码。

### 2.3.5 测试数据设计

不断地 rearrange 算法以及创建和 terminate proc 即可。

### 2.3.6 程序运行初值及运行结果分析

```
● cold@ubuntu:~/Desktop/xjtuOSlab$ gcc 2.3.c -o 2.3
○ cold@ubuntu:~/Desktop/xjtuOSlab$ ./2.3
```

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
3
Memory for PROCESS-01: 406
```

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
3
Memory for PROCESS-02: 512
```

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
3
Memory for PROCESS-03: 50
```

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
4
Kill Process, pid=2
```

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
5
-----
Free Memory:
      start_addr      size
          968          56
          406          512

Used Memory:
      PID      ProcessName start_addr      size
        3      PROCESS-03      918          50
        1      PROCESS-01         0          406
-----
```

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
3
Memory for PROCESS-04: 500
```

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
5
```

```
-----
Free Memory:
```

start_addr	size
968	56
906	12

```
Used Memory:
```

PID	ProcessName	start_addr	size
4	PROCESS-04	406	500
3	PROCESS-03	918	50
1	PROCESS-01	0	406

```
-----
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
3
Memory for PROCESS-05: 68
```



```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
5
No free memory blocks
```

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
0
cold@ubuntu:~/Desktop/xjtuOSlab$
```

## 2.3.7 实验总结

### 2.3.7.1 实验中的问题与解决过程

我没有遇到问题。理解问题的实质后就只是 coding 问题。

### 2.3.7.2 实验收获

- **First Fit (FF)**

算法思想

在 First Fit 算法中，分配器会扫描空闲块列表，找到第一个足够大的空闲块，并将其分配出去。如果该空闲块大于请求的内存，则会分割该块。

优缺点

- **优点:** 查找速度相对较快, 因为一旦找到合适的空闲块, 立即停止搜索。
- **缺点:** 容易导致内存碎片, 特别是外部碎片。

提高查找性能

在本次实验中好像并无。

- **Best Fit (BF)**

算法思想

Best Fit 算法在所有的空闲块中找到与请求最匹配 (即最接近) 的空闲块。这通常需要扫描整个空闲块列表。

优缺点

- **优点:** 最小化了每次分配后的剩余空间, 从而减少了浪费。
- **缺点:** 查找性能较差, 因为通常需要扫描整个列表; 可能导致更多小的空闲块, 进而增加碎片。

提高查找性能

维护一个按照大小排序的空闲块列表来提高查找性能。

- **Worst Fit (WF)**

算法思想

Worst Fit 算法选择最大的可用空闲块进行分配, 假设剩下的空间可能更有用。

优缺点

- **优点:** 理论上, 选择最大的空闲块应该能减少碎片。
- **缺点:** 在实践中, 通常并不比其他算法好, 查找性能也通常是最差的。

提高查找性能

维护一个按大小排序的空闲块列表来提高查找性能。

对于 FF 的空闲块排序 (事实上是不需要排序的, 因为无论如何排列, 只要找到第一个可用的就行), 对于 BF、WF 我采用了相同的排序算法——归并排序, 这是一种稳定的且速度极快的排序算法, 对于提升内存分配速率有很大作用。

- 内碎片

由于我们对于最小碎片长度有要求，一个空闲块的大小 enough 但是并不 sufficient。比如在我们的实验中最小的碎片长度为 10，一个空闲块大小为 28，现在要给他分配一个 20 大小的 proc，只能剩下 8，如果被切出，不符合要求，因此 28 要一次性分配给 proc。（此时 proc 的 size 是否应该保持为 request\_size（20）而不是 28？我个人认为要记录为 28，无论这 8 个空间是否被 proc 使用，但是其所“占用”的就是这么大，对于其他 proc 的影响也是这么大）

- 外碎片

由于我们的多次申请空间释放空间导致的长度小于最小值的碎片。例如经过一系列行为，导致 701—705 的空间为 free 的，但是显然这部分空间永远无法被使用到。

内存紧缩解决的就是这种碎片。

- 回收内存时。空闲块合并主要是通过 insertion sort 实现，根据空闲块的地址排序，如果空闲块的内存是连续的，那就将其合并为一个块。

### 2.3.7.3 意见与建议

本次实验具有一定的意义。继续保留，我认为可以调整为必做实验而非选做实验。

## 2.3.8 附件

### 2.3.8.1 附件 1 程序

*2.3.h*

*2.3.c*

### 2.3.8.2 附件 2 Readme

*lab2-3Readme.md*

## 实验三 Linux 动态模块与设备驱动

### 3.1 实验目的

本实验旨在深入理解 Linux 操作系统的内核模块和设备驱动程序的工作原理和实现方法。通过本实验，我们将能够：

9. 掌握 Linux 内核模块的动态加载和卸载机制。内核模块是 Linux 内核的重要组成部分，能够在系统运行时动态加载和卸载，从而提供了一种灵活和高效的方式来扩展内核功能，而无需重新编译整个内核。
10. 理解和实践字符设备驱动程序的编写。字符设备驱动程序是操作系统与字符设备交互的关键组件，能够处理如键盘、鼠标等设备的输入和输出。通过编写和测试一个简单的字符设备驱动程序，学生将学会如何管理和控制这类设备。
11. 学习设备文件的创建和使用。设备文件在用户空间提供了一种与设备驱动程序交互的机制。本实验将指导学生如何创建和使用这些文件，以便与编写的设备驱动程序进行通信。
12. 增强对 Linux 内核编程的实践经验。通过亲自编写内核模块和设备驱动程序，学生将获得宝贵的实践经验，这对于理解更复杂的内核机制和未来的系统级编程至关重要。

通过完成这些任务，不仅能够获得对 Linux 内核模块和设备驱动程序工作原理的深刻理解，还能够在实际编程中应用这些理论知识，为未来的高级系统编程打下坚实的基础。

### 3.2 实验内容

本实验内容涉及以下几个核心部分：

13. **编写基本的 Linux 内核模块：**首先，学生将编写一个简单的 Linux 内核模块。这个模块包括初始化（加载）和清理（卸载）两个基本函数。模块的目的是在加载时向系统日志输出一条信息，并在卸载时输出另一条信息，以此来验证模块的动态加载和卸载机制。
14. **编译和加载内核模块：**需要编写 Makefile 来编译内核模块，并使用 Linux 系统的内核模块工具（如 *insmod* 和 *rmmod*）来加载和卸载模块。这一步骤让学生了解内核模块编译的过程以及如何在运行中的 Linux 系统中添加和移除模块。

15. **编写字符设备驱动程序**: 将编写一个字符设备驱动程序, 该程序能够创建一个字符设备, 并对其执行基本操作, 如打开、读取、写入和关闭。这一部分旨在理解字符设备驱动程序的结构和工作原理, 并学会如何控制和管理字符设备。
16. **创建设备文件并测试驱动程序**: 学生将学习如何使用 `mknod` 命令在 `/dev` 目录下创建设备文件, 使用户空间的程序能够通过这个文件与字符设备驱动程序进行交互。接着, 通过编写测试脚本或直接在命令行中使用 `echo` 和 `cat` 命令, 将测试字符设备的读写功能。
17. **分析和调试**: 在整个实验过程中, 将使用 `dmesg` 命令来查看和分析内核日志信息, 以调试和验证内核模块和设备驱动程序的功能。这不仅帮助理解 Linux 内核的日志机制, 也是提高问题解决能力的重要环节。

通过完成这些内容, 将获得对 Linux 内核模块和设备驱动程序的深入理解, 并在实际的系统编程中积累宝贵的经验。

### 3.3 实验思想

本实验的核心思想是通过实践学习来深入理解 Linux 内核模块和设备驱动程序的原理和工作机制。具体而言, 实验的思想包括以下几个方面:

18. **理论与实践相结合**: 通过将理论知识和实践操作相结合, 学生能够更好地理解 Linux 内核模块的概念、生命周期以及设备驱动程序的工作原理。这种结合方式使得学生不仅能够阅读和理解理论知识, 还能通过实际编程来应用这些理论。
19. **动态学习过程**: 学生将在编写、编译、加载和测试内核模块的过程中逐步掌握 Linux 内核编程的技能。每一步都是学习过程的一部分, 错误和调试环节尤其重要, 因为它们提供了反馈, 帮助学生理解和修正问题。
20. **系统级编程理解**: 通过编写设备驱动程序, 学生将学会如何与 Linux 操作系统的核心部分交互。这提供了一个深入了解系统级编程的机会, 特别是在与硬件交互和处理并发操作方面。
21. **强调安全性和稳定性**: 内核编程需要特别注意安全性和稳定性。学生将学习如何编写既不会崩溃系统也不会引入安全漏洞的代码, 这是任何系统级程序员必须掌握的关键技能。
22. **综合应用能力培养**: 此实验不仅需要学生理解和实现内核模块和设备驱动程序, 还要求他们能够综合运用 Linux 命令行工具、编程技巧和调试方法。这种综合应用能力对于计算机科学和工程领域的学生来说至关重要。

总的来说，本实验旨在通过一系列具体任务，培养系统级编程能力，加深对 Linux 内核工作原理的理解，并为未来的高级系统编程和操作系统研究奠定基础。

## 3.4 实验步骤

### 3.4.1 编写和测试基本的 "Hello World" 内核模块

**目的：**熟悉内核模块的基本结构和生命周期。

**步骤：**

23. **创建模块代码：**编写一个简单的内核模块，包含初始化和退出函数。初始化函数在模块加载时执行，退出函数在模块卸载时执行。
24. **编译模块：**创建一个 Makefile 来编译内核模块。Makefile 指定了编译内核模块所需的命令和参数。
25. **加载和卸载模块：**使用 **insmod** 命令加载编译好的模块，使用 **rmmod** 命令卸载模块。
26. **检查输出：**使用 **dmesg** 命令查看内核日志，验证模块是否正确加载和卸载，并打印了相应的消息。

```
#include <linux/module.h>           // 引入支持动态添加和移除模块的必需头文件
#include <linux/kernel.h>           // 引入包含内核中常用功能的头文件，比如打印信息的 KERN_INFO

// 初始化函数，当模块加载时调用
static int __init hello_start(void)
{
    printk(KERN_INFO "Loading hello module...\n"); // 打印信息到内核日志
    printk(KERN_INFO "Hello world\n");           // 再次打印信息
    return 0; // 返回 0 表示模块加载成功
}

// 清理函数，当模块卸载时调用
static void __exit hello_end(void)
{
```

```

    printk(KERN_INFO "Goodbye Mr.\n"); // 在卸载时打印信息到内核日志
}

module_init(hello_start); // 指定模块初始化时调用的函数
module_exit(hello_end);   // 指定模块卸载时调用的函数

MODULE_LICENSE("GPL");           // 指明许可证类型
MODULE_AUTHOR("XHCuteDog");      // 指明作者
MODULE_DESCRIPTION("A simple Hello World Module"); // 描述模块功能
MODULE_VERSION("1.0");           // 指明模块版本

```

#### 27. 创建模块代码：

- 模块包括两个函数：**hello\_start** 和 **hello\_end**。
- **hello\_start** 在模块加载时调用，**hello\_end** 在卸载时调用。
- **printk** 函数用于向内核日志输出信息，**KERN\_INFO** 指定日志级别。

#### 28. 编译模块：

- 创建一个 Makefile，指定编译内核模块所需的指令和参数。
- 使用 **make** 命令编译模块，生成 **.ko** 文件。

#### 29. 加载和卸载模块：

- 使用 **sudo insmod hello.ko** 命令加载模块。
- 使用 **sudo rmmod hello** 命令卸载模块。

#### 30. 检查输出：

- 使用 **dmesg | tail** 命令查看内核日志。
- 确认加载时打印了 "Loading hello module..." 和 "Hello world"。
- 确认卸载时打印了 "Goodbye Mr."。

makefile:

```
obj-m += hello.o
```

```
all:
```

```
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

```
clean:
```

```
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

- **obj-m += hello.o**

- **obj-m** 指定要构建的目标是一个模块。
- **+=** 表示添加到现有的对象列表中。
- **hello.o** 是要编译的目标，对应于我们的模块源文件 **hello.c**。
- **all** :
  - 使用 **make -C** 更改到 Linux 内核源代码目录并执行 make 命令。
  - **/lib/modules/\$(shell uname -r)/build** 定位到当前运行内核的构建目录。
  - **M=\$(PWD)** 指定我们的 Makefile 所在目录。
  - **modules** 指示 make 构建模块。
- **clean** :
  - 用于清理编译产生的中间文件和目标文件。
  - 使用与 **all** 目标相同的路径和目录设置。

### 3.4.2 编写和测试字符设备驱动程序 **char\_dever\*\***

```
#include <linux/cdev.h>      // 字符设备结构体
#include <linux/device.h>    // 设备类
#include <linux/fs.h>        // 文件操作
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/uaccess.h>   // 包含 copy_to_user 函数

#define DEVICE_NAME "XH_DEVICE" // 设备名称
#define CLASS_NAME  "XH_DEVICE_CLASS" // 设备类名称

static int    majorNumber;           // 主设备号
static struct class*  exampleClass = NULL; // 设备类
static struct device* exampleDevice = NULL; // 设备
static char    message[256] = {0};   // 内存中的设备字符串
static short   size_of_message;      // 设备字符串的长度

// 设备打开函数
static int dev_open(struct inode *inodep, struct file *filep){
    printk(KERN_INFO "Example: Device has been opened\n");
    return 0;
}
```



```

// 设备读取函数
static ssize_t dev_read(struct file *filep, char *buffer, size_t len,
loff_t *offset){
    int error_count = 0;

    // 如果位置偏移已经到达或超过了消息的长度, 那么返回0表示已经读到文件末尾
    if (*offset >= size_of_message) {
        return 0;
    }

    // 如果读取的长度超过了消息的长度, 调整读取的长度
    if (*offset + len > size_of_message) {
        len = size_of_message - *offset;
    }

    // 将数据从内核空间复制到用户空间
    error_count = copy_to_user(buffer, message + *offset, len);

    if (error_count == 0) { // 如果成功复制所有数据
        printk(KERN_INFO "Example: Sent %ld characters to the user\n",
len);
        *offset += len; // 更新偏移位置
        return len; // 返回传输的字节数
    } else {
        printk(KERN_ERR "Example: Failed to send %d characters to the user\n", error_count);
        return -EFAULT; // 返回失败
    }
}

static ssize_t dev_write(struct file *filep, const char *buffer, size_t len, loff_t *offset){
    if (len > sizeof(message) - 1)
        len = sizeof(message) - 1;

    if (copy_from_user(message, buffer, len) != 0)
        return -EFAULT;

    message[len] = '\0'; // 确保字符串以空字符结束
    size_of_message = strlen(message); // 更新消息长度
    printk(KERN_INFO "Example: Received %zu characters from the user\n", len);
    return len;
}

```

```

}

// 设备关闭函数
static int dev_release(struct inode *inode, struct file *file){
    printk(KERN_INFO "Example: Device successfully closed\n");
    return 0;
}

// 文件操作结构体
static struct file_operations fops =
{
    .open = dev_open,
    .read = dev_read,
    .write = dev_write,
    .release = dev_release,
};

// 模块初始化函数
static int __init dever_init(void){
    printk(KERN_INFO "Example: Initializing the Example LKM\n");
    // KERN_INFO 定义了消息的重要性级别

    // 动态分配主设备号
    majorNumber = register_chrdev(0, DEVICE_NAME, &fops);
    // 调用 register_chrdev 函数来注册一个字符设备。
    // 0 作为第一个参数意味着系统将动态分配一个主设备号。
    // DEVICE_NAME 是我们设备的名称, &fops 是一个指向前面定义的 file_operations 结构的指针,
    // 它告诉内核哪些驱动程序函数应该被调用以响应相应的文件操作。

    if (majorNumber<0){
        // 如果小于0, 意味着注册设备号失败。函数打印一条警告信息并返回错误代码。
        printk(KERN_ALERT "Example failed to register a major number\n");
        return majorNumber;
    }
    printk(KERN_INFO "Example: registered correctly with major number %d\n", majorNumber);

    // 注册设备类
    exampleClass = class_create(THIS_MODULE, CLASS_NAME);
    // THIS_MODULE 宏引用当前的模块

```

```

    if (IS_ERR(exampleClass)){
        // 如果创建失败, 则注销前面注册的设备号, 并返回错误。
        unregister_chrdev(majorNumber, DEVICE_NAME);
        printk(KERN_ALERT "Failed to register device class\n");
        return PTR_ERR(exampleClass);
    }
    printk(KERN_INFO "Example: device class registered correctly\n");

    // 注册设备驱动
    exampleDevice = device_create(exampleClass, NULL, MKDEV(majorNumber, 0), NULL, DEVICE_NAME);
    // 创建一个设备, 它将出现在 /dev 目录下。device_create 函数将设备与前面创建的类关联起来。

    if (IS_ERR(exampleDevice)){
        class_destroy(exampleClass);
        unregister_chrdev(majorNumber, DEVICE_NAME);
        printk(KERN_ALERT "Failed to create the device\n");
        return PTR_ERR(exampleDevice);
    }
    printk(KERN_INFO "Example: device class created correctly\n");

    return 0;
}

// 模块退出函数
static void __exit dever_exit(void){
    device_destroy(exampleClass, MKDEV(majorNumber, 0));
    class_unregister(exampleClass);
    class_destroy(exampleClass);
    unregister_chrdev(majorNumber, DEVICE_NAME);
    printk(KERN_INFO "Example: Goodbye from the LKM!\n");
}

module_init(dever_init);
module_exit(dever_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("XHCuteDog");
MODULE_DESCRIPTION("A simple example Linux char driver");
MODULE_VERSION("0.1");

```

文件操作结构体 *fops*

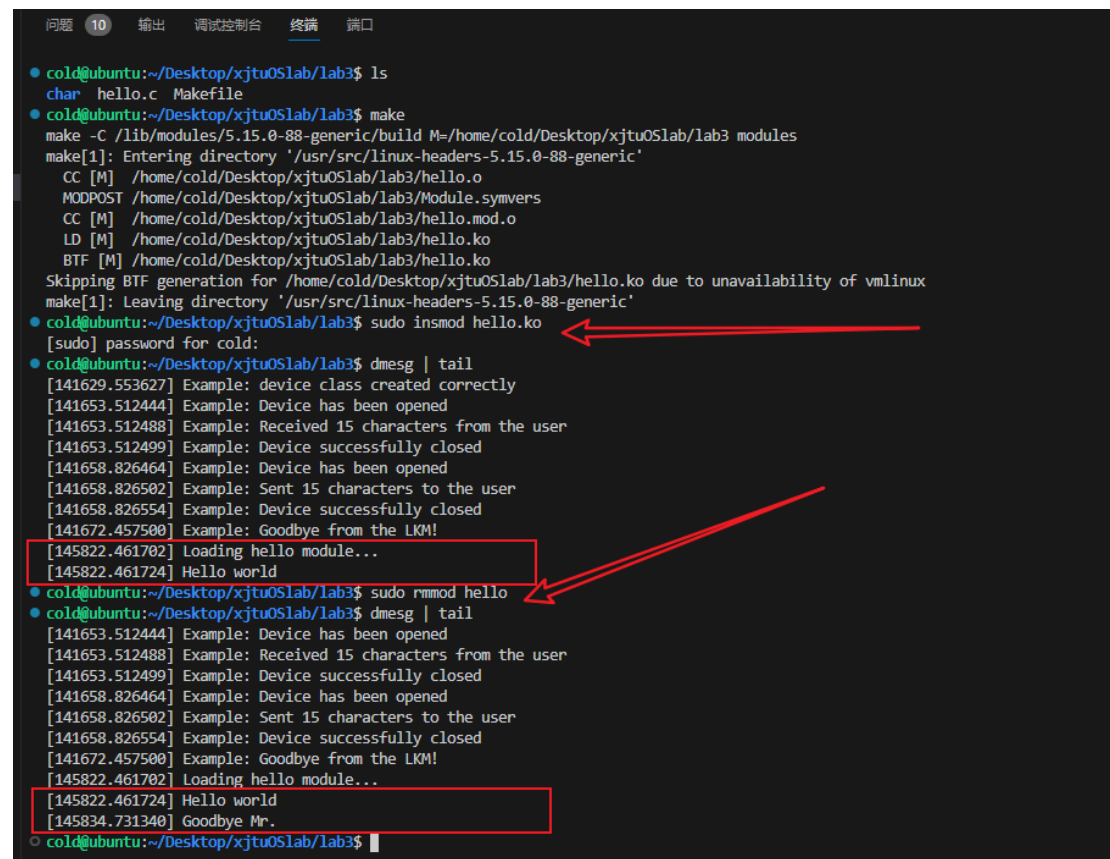
**file\_operations** 结构体是驱动程序与用户空间交互的关键。它将文件操作

(如读、写、打开、关闭) 映射到相应的驱动程序函数。在我们的例子中:

- **.open = dev\_open**: 当用户打开设备文件时调用。
- **.read = dev\_read**: 从设备中读取数据时调用。
- **.write = dev\_write**: 向设备写入数据时调用。
- **.release = dev\_release**: 当用户关闭设备文件时调用。

### 3.5 程序运行初值及运行结果分析

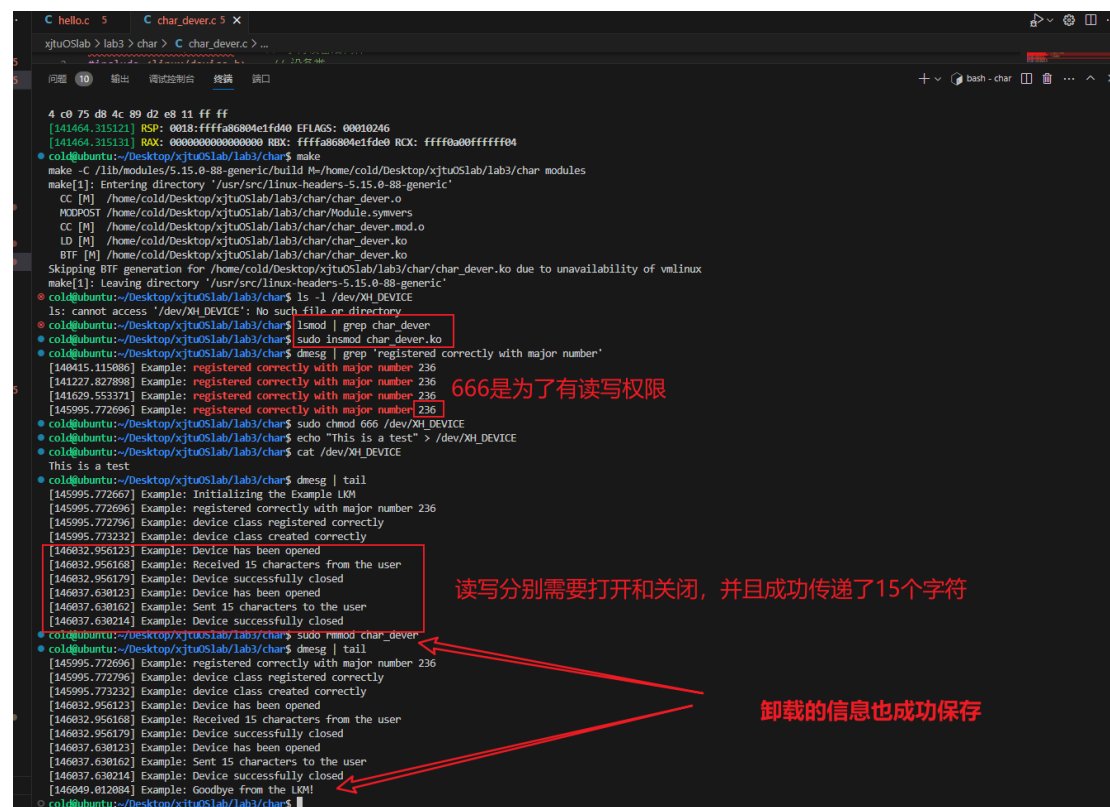
首先看简单版本:



```
冷@ubuntu:~/Desktop/xjtuOSlab/lab3$ ls
char hello.c Makefile
冷@ubuntu:~/Desktop/xjtuOSlab/lab3$ make
make -C /lib/modules/5.15.0-88-generic/build M=/home/cold/Desktop/xjtuOSlab/lab3 modules
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-88-generic'
  CC [M] /home/cold/Desktop/xjtuOSlab/lab3/hello.o
  MODPOST /home/cold/Desktop/xjtuOSlab/lab3/Module.symvers
  CC [M] /home/cold/Desktop/xjtuOSlab/lab3/hello.mod.o
  LD [M] /home/cold/Desktop/xjtuOSlab/lab3/hello.ko
  BTF [M] /home/cold/Desktop/xjtuOSlab/lab3/hello.ko
Skipping BTF generation for /home/cold/Desktop/xjtuOSlab/lab3/hello.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-88-generic'
冷@ubuntu:~/Desktop/xjtuOSlab/lab3$ sudo insmod hello.ko
[sudo] password for cold:
冷@ubuntu:~/Desktop/xjtuOSlab/lab3$ dmesg | tail
[141629.553627] Example: device class created correctly
[141653.512444] Example: Device has been opened
[141653.512488] Example: Received 15 characters from the user
[141653.512499] Example: Device successfully closed
[141658.826464] Example: Device has been opened
[141658.826502] Example: Sent 15 characters to the user
[141658.826554] Example: Device successfully closed
[141672.457500] Example: Goodbye from the LKM!
[145822.461702] Loading hello module...
[145822.461724] Hello world
冷@ubuntu:~/Desktop/xjtuOSlab/lab3$ sudo rmmod hello
冷@ubuntu:~/Desktop/xjtuOSlab/lab3$ dmesg | tail
[141653.512444] Example: Device has been opened
[141653.512488] Example: Received 15 characters from the user
[141653.512499] Example: Device successfully closed
[141658.826464] Example: Device has been opened
[141658.826502] Example: Sent 15 characters to the user
[141658.826554] Example: Device successfully closed
[141672.457500] Example: Goodbye from the LKM!
[145822.461702] Loading hello module...
[145822.461724] Hello world
[145834.731340] Goodbye Mr.
冷@ubuntu:~/Desktop/xjtuOSlab/lab3$
```

可以正常看到载入和卸载的信息!!! 成功。

来看下一个~



```
4 c0 75 d8 4c 89 d2 e8 11 ff ff
[141464.315121] RSP: 0018:ffffa86804e1fd40 EFLAGS: 00010246
[141464.315131] RAX: 0000000000000000 RBX: fffffa86804e1fde9 RCX: fffffa0000000004
cold@ubuntu:~/Desktop/xjtuOSlab/lab3/char$ make
make -C /lib/modules/5.15.0-88-generic/build M=/home/cold/Desktop/xjtuOSlab/lab3/char modules
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-88-generic'
CC [M] /home/cold/Desktop/xjtuOSlab/lab3/char/char_dever.o
WPOPT /home/cold/Desktop/xjtuOSlab/lab3/char/module.symvers
CC [M] /home/cold/Desktop/xjtuOSlab/lab3/char/char_dever.mod.o
LD [M] /home/cold/Desktop/xjtuOSlab/lab3/char/char_dever.ko
BTF [M] /home/cold/Desktop/xjtuOSlab/lab3/char/char_dever.ko
Skipping BTF generation for /home/cold/Desktop/xjtuOSlab/lab3/char/char_dever.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-88-generic'
cold@ubuntu:~/Desktop/xjtuOSlab/lab3/char$ ls -l /dev/XH_DEVICE
ls: cannot access '/dev/XH_DEVICE': No such file or directory
cold@ubuntu:~/Desktop/xjtuOSlab/lab3/char$ lsmod | grep char_dever
cold@ubuntu:~/Desktop/xjtuOSlab/lab3/char$ sudo insmod char_dever.ko
cold@ubuntu:~/Desktop/xjtuOSlab/lab3/char$ dmesg | grep 'registered correctly with major number'
[140415.115086] Example: registered correctly with major number 236
[141227.827898] Example: registered correctly with major number 236
[141629.553371] Example: registered correctly with major number 236
[145995.772696] Example: registered correctly with major number 236
cold@ubuntu:~/Desktop/xjtuOSlab/lab3/char$ sudo chmod 666 /dev/XH_DEVICE
cold@ubuntu:~/Desktop/xjtuOSlab/lab3/char$ echo "This is a test" > /dev/XH_DEVICE
cold@ubuntu:~/Desktop/xjtuOSlab/lab3/char$ cat /dev/XH_DEVICE
This is a test
cold@ubuntu:~/Desktop/xjtuOSlab/lab3/char$ dmesg | tail
[145995.772667] Example: Initializing the Example UOM
[145995.772696] Example: registered correctly with major number 236
[145995.772796] Example: device class registered correctly
[145995.773232] Example: device class created correctly
[146032.956123] Example: Device has been opened
[146032.956168] Example: Received 15 characters from the user
[146032.956179] Example: Device successfully closed
[146037.630123] Example: Device has been opened
[146037.630162] Example: Sent 15 characters to the user
[146037.630214] Example: Device successfully closed
cold@ubuntu:~/Desktop/xjtuOSlab/lab3/char$ sudo rmmod char_dever
cold@ubuntu:~/Desktop/xjtuOSlab/lab3/char$ dmesg | tail
[145995.772696] Example: registered correctly with major number 236
[145995.772796] Example: device class registered correctly
[145995.773232] Example: device class created correctly
[146032.956123] Example: Device has been opened
[146032.956168] Example: Received 15 characters from the user
[146032.956179] Example: Device successfully closed
[146037.630123] Example: Device has been opened
[146037.630162] Example: Sent 15 characters to the user
[146037.630214] Example: Device successfully closed
[146049.012084] Example: Goodbye from the UOM!
```

666是为了有读写权限

读写分别需要打开和关闭，并且成功传递了15个字符

卸载的信息也成功保存

## 3.6 实验总结

### 3.6.1 实验中的问题与解决过程

我遇到了一系列挑战和问题，以下是其中一些主要的问题及其解决过程：

31. **Makefile 错误**：最初的 Makefile 编写时出现了“missing separator”错误。问题在于 Makefile 严格要求使用制表符而不是空格作为命令行的前缀。通过替换空格为制表符解决了这个问题。
32. **模块加载错误**：在尝试加载 **char\_dever** 模块时遇到“File exists”错误。这是因为同名模块已经加载在内核中。通过先卸载已存在的模块再进行加载解决了此问题。
33. **内核崩溃**：在测试阶段，向设备写入数据时遇到了终端崩溃的问题。经过调查，发现是 **dev\_write** 函数中未正确处理用户空间传来的数据，导致内核空间的缓冲区溢出。通过在写入操作中添加足够的检查和边界保护解决了这个问题。

## 3.6.2 实验收获

提供了宝贵的学习机会，特别是在以下方面：

34. **Linux 内核模块的理解**：通过实际编写和加载模块，深入理解了 Linux 内核模块的工作原理，包括它们的生命周期、如何与内核其他部分交互等。
35. **字符设备驱动开发**：学习了如何编写字符设备驱动程序，并通过实际的代码实现来理解设备文件的创建和操作。
36. **调试技能的提升**：在解决编译和运行时遇到的问题过程中，提高了使用诸如 ***dmesg*** 等工具进行调试的能力。
37. **系统级编程经验**：增强了在系统级别进行编程的实践经验，这对于理解更复杂的系统概念和结构至关重要。

## 3.6.3 意见与建议

- **更多实践机会**：建议在教学过程中提供更多类似的实践机会，以帮助学生更好地理解理论知识。
- **文档和社区资源**：鼓励学生利用现有的文档和社区资源，如 Linux 内核源代码和在线论坛，以便于更好地理解和解决问题。
- **安全性和稳定性的重视**：在进行内核级编程时，强调代码的安全性和稳定性，避免潜在的系统崩溃和安全漏洞。

## 3.7 附件

### 3.7.1 附件 1 程序

[hello.c](#)

[char\\_dever.c](#)

### 3.7.2 附件 2 Readme