

**操作系统专题实验报告**

班级： 计算机2102

学号： 2216515195

姓名： 申程宇

2023年12月5日

目 录

[1. openEuler系统环境实验 1](#_Toc6793)

[1.1 实验目的 1](#_Toc11669)

[1.2 实验内容 1](#_Toc26181)

[1.3 实验思想 2](#_Toc29359)

[1.4 实验步骤 2](#_Toc17973)

[1.5 程序运行初值及运行结果分析 3](#_Toc19653)

[1.6 实验总结 19](#_Toc13147)

[1.6.1 实验中的问题与解决过程 21](#_Toc22457)

[1.6.2 实验收获 21](#_Toc19822)

[1.6.3 意见与建议 21](#_Toc16984)

[1.7 附件 21](#_Toc9774)

[2. 进程通信与内存管理 23](#_Toc9439)

[2.1 实验目的 23](#_Toc5213)

[2.2 实验内容 23](#_Toc1182)

[2.3 实验思想 24](#_Toc13720)

[2.4 实验步骤及结果分析 25](#_Toc7347)

[2.5 回答问题 43](#_Toc4700)

[2.5.1 软中断通信 43](#_Toc4902)

[2.5.2 管道通信 46](#_Toc871)

[2.6 实验总结 47](#_Toc21237)

[2.6.1 实验中的问题与解决过程 49](#_Toc23244)

[2.6.2 实验收获 49](#_Toc6511)

[2.7 附件 50](#_Toc28159)

[2.7.1 附件1 程序 50](#_Toc17943)

[2.7.2 附件2 ReadMe 50](#_Toc26342)

[3. 文件系统 51](#_Toc18914)

[3.1 实验目的 51](#_Toc20138)

[3.2 实验内容 51](#_Toc32077)

[3.3 实验思想 51](#_Toc10206)

[3.4 文件系统架构 51](#_Toc31853)

[3.4.1 结构图 51](#_Toc23275)

[3.4.2 主要数据结构的定义 52](#_Toc24606)

[3.4.3 主要函数及设计思路 54](#_Toc3822)

[3.5 实验总结 57](#_Toc9719)

[3.5.1实验中的问题与解决过程 58](#_Toc6754)

[3.5.2 实验收获 59](#_Toc25786)

[3.6 Q&A 59](#_Toc21077)

[3.7 运行结果 60](#_Toc6479)

[3.8附件 63](#_Toc29681)

[3.8.1 附件1 程序 63](#_Toc30861)

[3.8.2 附件2 ReadMe 63](#_Toc928)

# openEuler系统环境实验

## 实验目的

（1）熟悉基于鲲鹏架构弹性云服务器ECS上openEuler操作系统基本系统环境；

（2）理解进程的创建、进程地址空间的概念、父子进程资源的关系；

（3）观察进程调度，了解进程调度的过程，了解孤儿进程和僵尸进程的区别；

（4）理解线程与进程的关系，对等线程间的关系。

（5）理解并运用信号量及其PV操作、自旋锁实现线程间的同步互斥。

## 实验内容

（1）在华为云上搭建openEuler操作系统环境，通过运行shell命令查看系统信息以了解并熟悉openEuler操作系统。

（2）熟悉操作命令、编辑、编译、运行程序。完成给定程序（图1.1所示的程序）的运行验证，多运行程序几次观察结果；去除wait后再观察结果并进行理论分析。

（2）在所给程序中添加一个全局变量，在父进程和子进程中分别对这个变量做不同操作并输出操作结果，对结果进行观察并解释，同时输出两种变量的地址观察并分析。

（3）修改程序，在子进程中调用system函数和在子进程中调用exec族函数以执行自己写的一段程序，在此程序中输出进程PID，进行比较分析。

（4）线程实验:在进程中给一变量赋初值并创建两个线程，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并输出结果 ；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；在线程中调用system函数/exec族函数以执行自己写的一段程序，输出进程PID和线程TID, 进行比较分析。

（5）自旋锁实验：在进程中给一变量赋初值并成功创建两个线程；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；使用自旋锁实现互斥和同步。

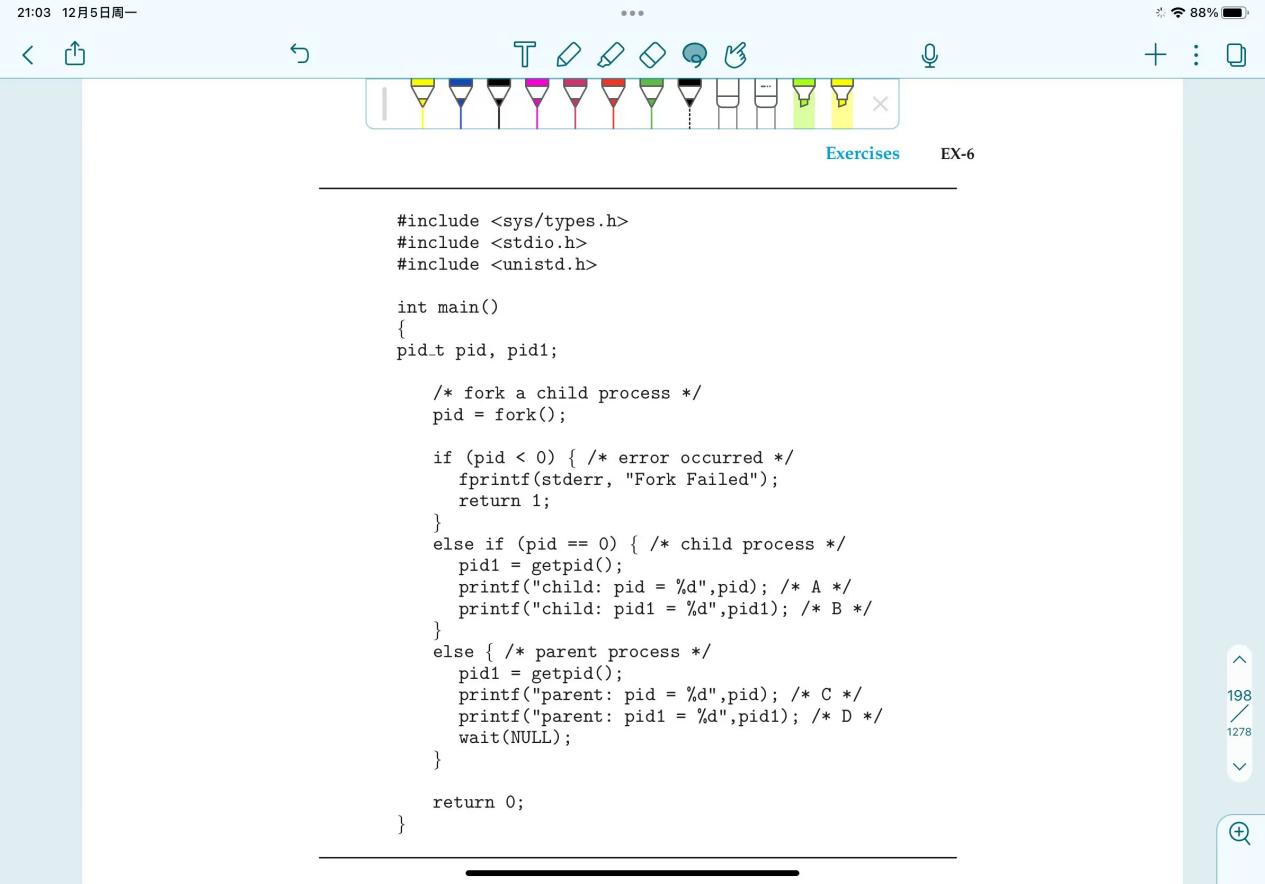


图1.1 给定程序

## 实验思想

openEuler基于Linux内核，支持多种架构，支持多种虚拟化技术，支持多种容器技术，支持多种云计算技术。openEuler采用Linux内核，通过了解openEuler系统下的shell命令和系统命令后，所学习的知识可以轻松迁移到其他Linux发行版本中，如Ubuntu。

通过配置基于鲲鹏架构弹性云服务器ECS上openEuler操作系统基本系统环境，运行shell命令查看系统信息以达到了解和使用openEuler操作系统的目的。

通过进程相关编程实验，编写和运行简单的进程、线程相关程序，理解进程与线程概念、进程空间与物理内存空间、进程调度、进程间变量管理、进程调用其他程序、如何实现正确的并发（同步互斥）等方面在操作系统中的实际操作。具体来讲，通过输出子进程和父进程的PID，观察进程调度，了解进程调度的过程，了解孤儿进程和僵尸进程的区别。观察并发进程中的全局变量改变，输出父子进程共享变量地址以了解物理地址与虚地址概念，从而理解进程地址空间的概念，了解关于地址绑定的基础知识。创建两个线程运行后体会线程与进程的关系、对等线程间的资源共享以及同步与互斥的知识。

## 实验步骤

（1）登录华为云,搭建openEuler操作系统环境。

（2）编辑、编译、运行给定的程序，多次运行观察实验结果。

（2）去除wait()后观察结果并进行理论分析, 比较程序在有无wait()函数时的运行结果，分析wait()函数的作用。

（3）添加一个全局变量，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，同时输出两种变量的地址并进行分析。

（4）在子进程中调用system函数执行自己写的一段程序，在此程序中输出进程PID和其父进程PID进行比较分析。

（5）在子进程中调用exec函数执行自己写的一段程序，在此程序中输出进程PID和其父进程PID进行比较分析。

（6）修改给定程序，给一变量赋初值并创建两个线程，在两个线程中分别对此变量循环5000次做不同的操作并输出结果。

（7）多运行几遍程序观察运行结果。

（8）完成上述过程的同步和互斥操作。

（9）在两个线程中调用system函数执行自己写的一段程序，在此程序中输出线程TID及进程PID,进行分析。

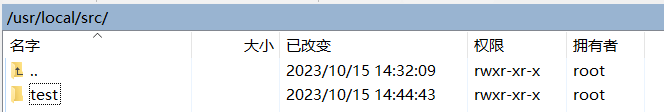
（10）在两个线程中调用exec函数执行自己写的一段程序，在此程序中输出线程TID及进程PID,进行分析。

（11）参考实验指导书，编写模拟自旋锁程序代码，补充主函数代码，用自旋锁实现线程间的同步。编译并运行程序，分析运行结果

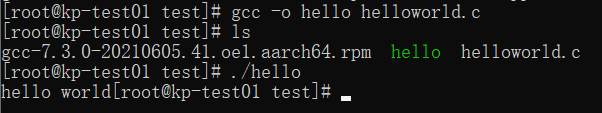
## 程序运行初值及运行结果分析

云环境测试：

服务端新建一个test文件夹

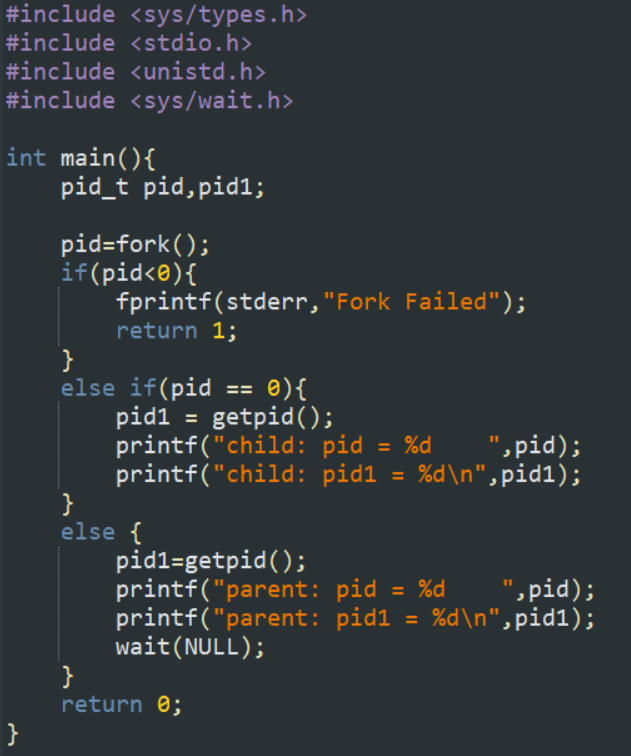


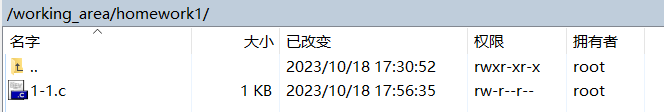
编译结束执行exe文件输出如下



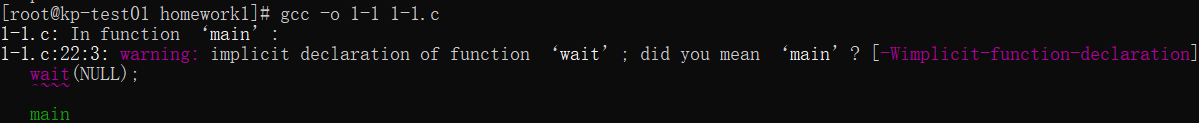
1. *进程相关实验*
2. 运行1-1代码

进入我们创建的本次作业的代码路径，上传c文件（为方便观察，将代码每个进程执行换行'\n'）





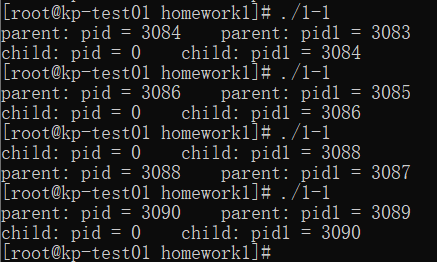
运行如下



出现该问题是源代码缺少头文件

#include **<sys/wait.h>**

添加后，多次运行如下



首先分析fork()不同返回值的含义如下：

fork()调用一次，但返回两次，一次在父进程中，一次在子进程中。返回值有不同的含义，具体如下：

在父进程中：

返回新创建子进程的进程ID（PID）。

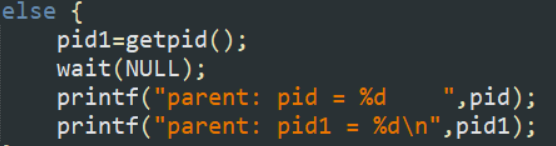
如果返回值为负数，表示创建子进程失败。

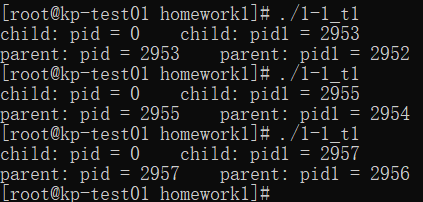
在子进程中：

返回0，表示当前进程是子进程。

如果返回值为负数，表示创建子进程失败。

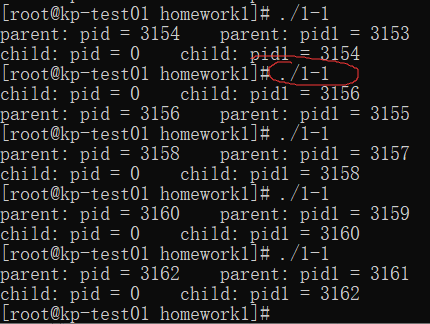
分析： pid值按运行顺序分配，父子的执行顺序是随机的，均可以先执行。然而这样并不能看出wait的效果，理论上wait(NULL)会让父进程进入阻塞（等待子进程结束），然而这段代码的输出在wait之前因此看不出差异。因此尝试修改代码，将wait移到print之前，如下





这次便可以明显看到parent的wait阻塞效果，使得child执行后才会执行wait后续代码。

1. 删去wait()函数



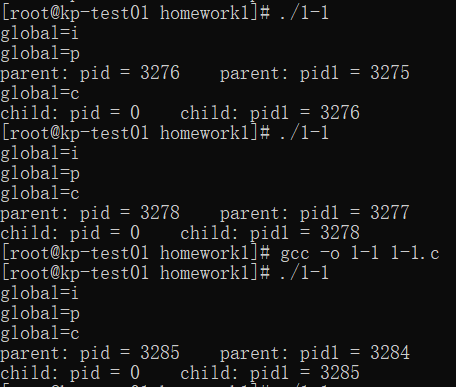
pid值有中断,说明这一段时间有其他进程诞生,并且此时父子的执行顺序是完全随机的。

1. 增加全局变量并进行不同操作

修改代码如下

#include **<sys/types.h>**  
#include **<stdio.h>**  
#include **<unistd.h>**  
char global=0; *//全局变量定义*   
int main(){  
 pid\_t pid,pid1;  
 global='i'; *//i*   
 printf("global=%c\n",global);   
 pid=fork();  
 **if**(pid<0){  
 global='z'; *//z*  
 printf("global=%c\n",global);  
 fprintf(stderr,"Fork Failed");  
 **return** 1;  
 }  
 **else** **if**(pid == 0){  
 global='c';  
 printf("global=%c\n",global);  
 pid1 = getpid();  
 printf("child: pid = %d ",pid);  
 printf("child: pid1 = %d\n",pid1);  
 }  
 **else** {  
 global='p';  
 printf("global=%c\n",global);  
 pid1=getpid();  
 printf("parent: pid = %d ",pid);  
 printf("parent: pid1 = %d\n",pid1);  
 }  
 **return** 0;  
}

定义全局变量global=0,在三种状态下分别修改值为z, c, p运行如下



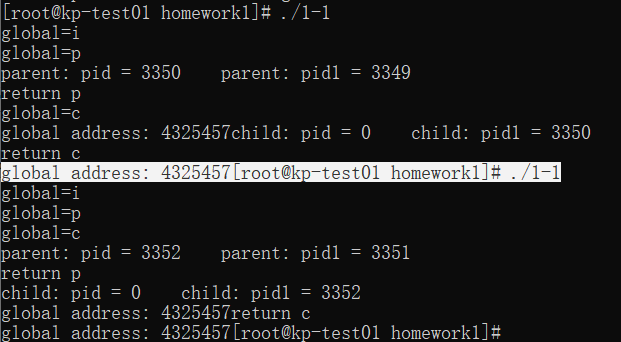
i先输出，父子进程是完全独立且随机的执行过程，因此存在交错输出的情况.

1. return前操作global

鉴于程序基本不会fork失败，因此只在return 0前增加语句

printf("return %c\n",global);  
printf("global address: %d",&global);

运行如下



此处父进程先结束，然而其global仍然为p，即使在时间上在子进程修改为c，说明父子进程的global值在同一时刻不同，然而观察其地址却指向同一地址。查阅后得知

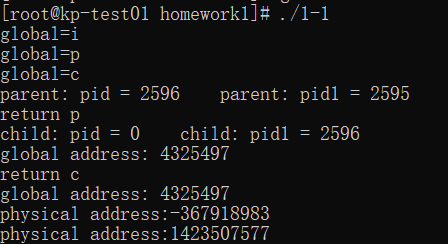
父子进程没有修改global前均共享同一虚拟地址（非物理地址），当某一方修改global后，会copy成另一个副本,修改后再通过内存管理映射回进程。

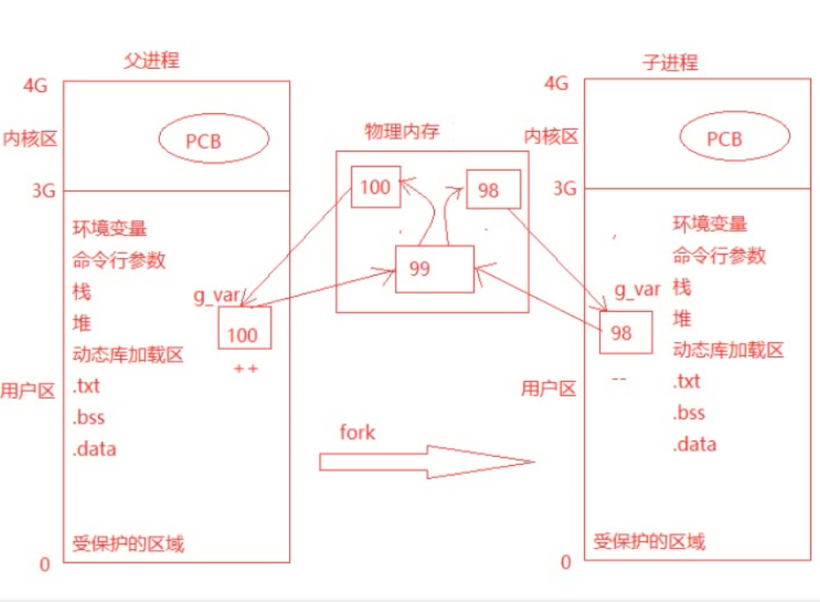
【增加】

获取物理地址的代码

#include **<sys/types.h>**  
#include **<stdio.h>**  
#include **<unistd.h>**  
#include **<sys/stat.h>**  
#include **<stdlib.h>**   
#include **<fcntl.h>**  
#include **<stdint.h>**  
void mem\_addr(unsigned long vaddr, unsigned long \*paddr)  
{  
 int pageSize = getpagesize();*//调用此函数获取系统设定的页面大小*  
  
 unsigned long v\_pageIndex = vaddr / pageSize;*//计算此虚拟地址相对于0x0的经过的页面数*  
 unsigned long v\_offset = v\_pageIndex \* **sizeof**(uint64\_t);*//计算在/proc/pid/page\_map文件中的偏移量*  
 unsigned long page\_offset = vaddr % pageSize;*//计算虚拟地址在页面中的偏移量*  
 uint64\_t item = 0;*//存储对应项的值*  
  
 int fd = open("/proc/self/pagemap", O\_RDONLY);*// 以只读方式打开/proc/pid/page\_map*  
 **if**(fd == -1)*//判断是否打开失败*  
 {  
 printf("open /proc/self/pagemap error\n");  
 **return**;  
 }  
  
 **if**(lseek(fd, v\_offset, SEEK\_SET) == -1)*//将游标移动到相应位置，即对应项的起始地址且判断是否移动失败*  
 {  
 printf("sleek error\n");  
 **return**;   
 }  
  
 **if**(read(fd, &item, **sizeof**(uint64\_t)) != **sizeof**(uint64\_t))*//读取对应项的值，并存入item中，且判断读取数据位数是否正确*  
 {  
 printf("read item error\n");  
 **return**;  
 }  
  
 **if**((((uint64\_t)1 << 63) & item) == 0)*//判断present是否为0*  
 {  
 printf("page present is 0\n");  
 **return** ;  
 }  
  
 uint64\_t phy\_pageIndex = (((uint64\_t)1 << 55) - 1) & item;*//计算物理页号，即取item的bit0-54*  
  
 \*paddr = (phy\_pageIndex \* pageSize) + page\_offset;*//再加上页内偏移量就得到了物理地址*  
}  
  
char global=0; *//全局变量定义*   
int main(){  
 pid\_t pid,pid1;  
 global='i'; *//i*   
 printf("global=%c\n",global);   
 pid=fork();  
 **if**(pid<0){  
 global='z'; *//z*  
 printf("global=%c\n",global);  
 fprintf(stderr,"Fork Failed");  
 **return** 1;  
 }  
 **else** **if**(pid == 0){  
 global='c';  
 printf("global=%c\n",global);  
 pid1 = getpid();  
 printf("child: pid = %d ",pid);  
 printf("child: pid1 = %d\n",pid1);  
 }  
 **else** {  
 global='p';  
 printf("global=%c\n",global);  
 pid1=getpid();  
 printf("parent: pid = %d ",pid);  
 printf("parent: pid1 = %d\n",pid1);  
 }  
 printf("return %c\n",global);  
 printf("global address: %d\n",&global);  
 long phy;  
 mem\_addr((unsigned long)&global, &phy);  
 printf("physical address:　%d\n",phy);   
 **return** 0;  
}

运行结果





***小结:***

**进程地址空间的概念（父子进程资源的关系、为什么同一个变量会有不同的值）**

在创建子进程时，父进程和子进程会共享相同的内存映像（memory image），这包括代码段（text segment）、数据段（data segment）和堆（heap）。但由于操作系统使用了虚拟内存技术，父子进程实际上拥有各自独立的虚拟地址空间，这使得它们之间的内存空间互相隔离。

因此，虽然父进程和子进程在开始时共享相同的内存映像，但它们之后可以独立地修改各自的数据。如果一个进程修改了自己的数据，不会影响到另一个进程。这是因为虚拟内存将物理内存分割成页，而每个进程都有自己的页表，将虚拟地址映射到实际的物理地址。因此，同一个变量在不同的进程中可以有不同的值。

这也解释了为什么同一个变量在父进程和子进程中的值可能会不同。当子进程被创建时，它拥有父进程的内存映像的副本，但之后对这些数据的修改不会影响到父进程的对应数据。这样的设计使得进程之间能够独立工作，提高了系统的安全性和稳定性。

**wait()的作用**

等待子进程终止： 当父进程调用wait()时，它会被挂起，直到有一个子进程终止。父进程可以通过wait()获取子进程的退出状态以及一些关于子进程终止的其他信息。

防止孤儿进程： wait()的另一个作用是防止孤儿进程的产生。孤儿进程是指父进程先于子进程退出，导致子进程成为孤儿，被init进程（通常是进程ID为1的进程）收养。通过在父进程中使用wait()，父进程可以等待子进程终止，从而避免子进程成为孤儿。

**关于孤儿进程：**

如果父进程在子进程之前退出，子进程将由init进程接管。init进程会自动调用wait()来获取已终止子进程的退出状态。

**关于并发进程的调度：**

进程调度： 操作系统使用调度算法来决定在多个进程之间分配CPU时间。不同的操作系统有不同的调度算法，例如先来先服务（FCFS）、最短作业优先（SJF）、轮转调度（Round Robin）等。

并发执行： 操作系统可以在多个进程之间实现并发执行，即同时执行多个进程。这通过在不同的时间片内切换不同的进程来实现。调度器负责决定在给定时刻哪个进程应该运行。

进程优先级： 操作系统通常给进程分配优先级，这影响了进程被调度的顺序。高优先级的进程可能更频繁地获得CPU时间。

1. system()与exec族函数

编写system\_call.c文件

#include **<sys/types.h>**  
#include **<stdio.h>**  
#include **<unistd.h>**  
  
int main(){  
 pid\_t p;  
 p=getpid();  
 printf("\n system\_call, PID:%d\n",p);  
 **return** 0;  
}

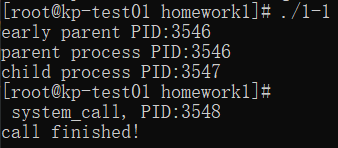
编译（可执行文件才能被进程调用）

gcc -o system\_call system\_call.c

修改1-1代码

#include **<sys/types.h>**
  
#include **<stdio.h>**
  
#include **<unistd.h>**
  
#include **<stdlib.h>**
  
int main(){
  
 pid\_t pid,pid1;
  
 pid=getpid();
  
 printf("early parent PID:%d\n",pid);
  
 pid=fork();
  
 **if**(pid<0){
  
 fprintf(stderr,"Fork Failed");
  
 **return** 1;
  
 }
  
 **else** **if**(pid == 0){ *//child*  
 pid1 = getpid();
  
 printf("child process PID:%d\n",pid1);
  
 system("/working\_area/homework1/system\_call"); *//执行调用*  
 printf("call finished!\n");
  
 }
  
 **else** {
  
 pid1=getpid();
  
 printf("parent process PID:%d\n",pid1);
  
 }
  
 **return** 0;
  
}

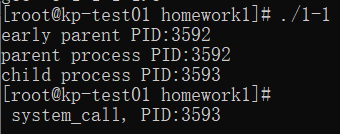
运行如图



子进程调用system\_call后，PID从3547变成3548,说明system调用新建了一个进程。而在调用后的printf()语句成功执行，说明system调用后仍然会返回原中断点继续执行该进程。

接着修改1-1，system语句改为execl()函数

execl("/working\_area/homework1/system\_call",NULL);



system调用后所在进程的PID和原子进程PID相同，并且最后的printf语句不再执行，因此是直接执行并覆盖原进程的行为。

***小结：***

system 创建一个新的进程来执行命令，而 exec 函数族在当前进程中执行新程序。

system 函数返回命令的退出状态，而 exec 函数族不返回，除非出现错误。

system 函数会阻塞当前进程，等待命令完成，而 exec 函数族在执行成功后不会返回。

*2. 线程相关实验*

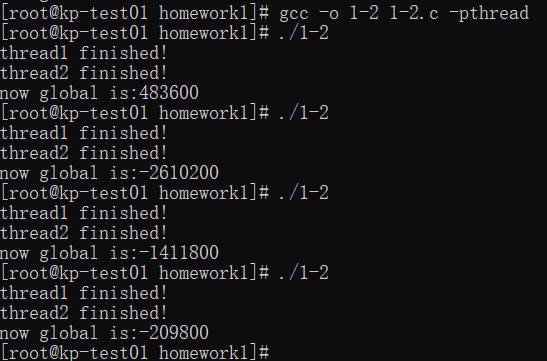
1. 双线程操作共享变量

创建双线程并修改共享变量的代码如下

#include **<stdio.h>**  
#include **<unistd.h>**  
#include **<pthread.h>**  
int global = 0;  
  
void \*runner1(){  
 **for**(int i=1;i<=1000000;i++)  
 global+=100;  
 printf("thread1 finished!\n");  
 pthread\_exit(0);  
}  
void \*runner2(){  
 **for**(int i=1;i<=1000000;i++)  
 global-=100;  
 printf("thread2 finished!\n");  
 pthread\_exit(0);  
}  
int main(){  
 pthread\_t tid1,tid2;  
 pthread\_create(&tid1,NULL,runner1,NULL);*//创建子线程1*  
 pthread\_create(&tid2,NULL,runner2,NULL);   
 pthread\_join(tid1,NULL);  
 pthread\_join(tid2,NULL);*//阻塞，避免提前输出*  
 printf("now global is:%d\n",global);  
 **return** 0;  
}

由于pthread并非Linux系统默认库，因此编译要添加-pthread，即

gcc -o 1-2 1-2.c -pthread



这里的观察到两个线程finished信号，因此可以判断两个线程的确都结束了，理论上全局变量经过加减操作应该是0，然而global值是介于-1000000\*100到+1000000\* 100之间的非零数，说明两个线程均在各自并发执行对global变量修改的时候出现了冲突，导致结果不可预测。[这里加了pthread\_join()语句为了保证两个线程彻底结束]

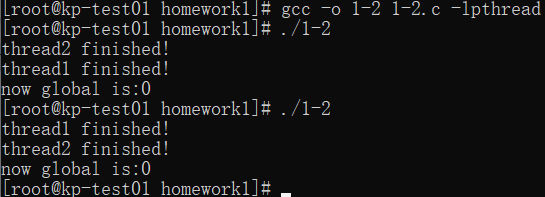
1. 信号量PV操作控制访问

增加信号量signal,初始化操作如下

sem\_t sem; *//新增信号量*   
int global = 0;  
sem\_init(&sem,0,1);*//设置信号量，仅本进程访问且初值为1*

修改runner1函数(runner2同理)

void \*runner1(){  
 **for**(int i=1;i<=1000000;i++){  
 sem\_wait(&sem); *//P操作*  
 global+=100;  
 sem\_post(&sem); *//V操作*  
 }  
 printf("thread1 finished!\n");  
 pthread\_exit(0);  
}



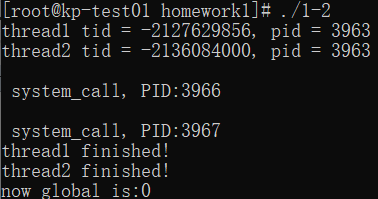
运行两次结果都是0，说明此时通过PV操作，使得共享变量的访问没有竞争变得稳定，故输出的值是可预测的。

1. 线程中调用system或exec簇函数

修改代码1-2如下

#include **<stdio.h>**  
#include **<unistd.h>**  
#include **<pthread.h>**  
#include **<semaphore.h>**  
#include **<stdlib.h>**  
sem\_t sem; *//新增信号量*   
int global = 0;  
  
void \*runner1(){  
 pthread\_t tid11;  
 tid11==pthread\_self();  
 pid\_t pid1=getpid();  
 printf("thread1 tid = %d, pid = %d\n",tid11,pid1);  
 system("/working\_area/homework1/system\_call");  
 printf("thread1 finished!\n");  
 pthread\_exit(0);  
}  
void \*runner2(){  
 pthread\_t tid22;  
 tid22=pthread\_self();  
 pid\_t pid2=getpid();  
 printf("thread2 tid = %d, pid = %d\n",tid22,pid2);  
 system("/working\_area/homework1/system\_call");  
 printf("thread2 finished!\n");  
 pthread\_exit(0);  
}  
int main(){  
 sem\_init(&sem,0,1);*//设置信号量，仅本进程访问且初值为1*   
 pthread\_t tid1,tid2;  
 pthread\_create(&tid1,NULL,runner1,NULL);*//创建子线程1*  
 pthread\_create(&tid2,NULL,runner2,NULL);   
 pthread\_join(tid1,NULL);  
 pthread\_join(tid2,NULL);  
 printf("now global is:%d\n",global);  
 **return** 0;  
}

代码中在两个线程的runner函数各自调用system，并在前后都输出了相应标记。

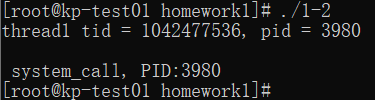


首先，两个thread的tid是不同的，但是是同一进程(pid=3963)，两个线程的系统调用原理和1-1相同，均是新建进程，故PID不同于原进程中的线程。最后两个线程正常结束，这里两个system\_call是密切相邻先后执行的，即是两个并发线程在时间上以紧密的顺序先后调用了system\_call。

接着修改代码，使用execl函数，将system调用语句修改为

execl("/working\_area/homework1/system\_call",NULL);

编译运行如下



观察可知，此时，第一个线程一旦执行了excel调用system\_call后，PID不变，进程（包含这两个线程）直接被新内容代替，执行system\_call中的内容后，没有原进程中的finished!标记，因此不会返回原断点继续执行。

***system()和exec簇函数小结***

二者都是从指定目录打开文件进行执行，区别是，system()会创建新进程来执行对应代码，并在执行后返回原中断处继续向下执行源代码，而exec簇函数会直接执行对应代码，处于同一进程，且不会返回断点处。

如果调用失败时，execl会设置errno并返回-1，然后从原程序的调用点接着往下执行。而对于system，如果执行成功返回0，失败则返回非零值。

***线程（Thread）和进程（Process）的区别：***

定义：

进程： 进程是一个独立的执行单位，拥有自己的地址空间、文件描述符、系统资源等。进程之间相互独立，通信需要通过进程间通信（IPC）机制。

线程： 线程是进程中的一个执行单元，共享相同的地址空间和文件描述符。线程之间更轻量级，通信可以直接通过共享内存等方式进行。

资源开销：

进程： 创建、撤销一个进程的开销较大，因为需要分配独立的地址空间等资源。

线程： 创建、撤销一个线程的开销相对较小，因为它们共享大部分资源。

通信：

进程： 进程之间通信需要使用专门的IPC机制，如管道、消息队列、共享内存等。

线程： 线程之间通信更直接，可以通过共享内存、全局变量等方式进行。

独立性：

进程： 进程是相互独立的执行单元。

线程： 线程是进程内的执行单元，共享进程的资源。

关于线程间的同步与互斥：

同步： 同步是指协调线程之间的执行顺序，以避免竞态条件（Race Condition）和其他并发问题。常见的同步机制有信号量、条件变量等。

互斥： 互斥是指一次只允许一个线程访问共享资源，以防止数据的不一致性。常见的互斥机制有互斥锁（Mutex）。

*3. 自旋锁实验*

1. 编写模拟自旋锁代码

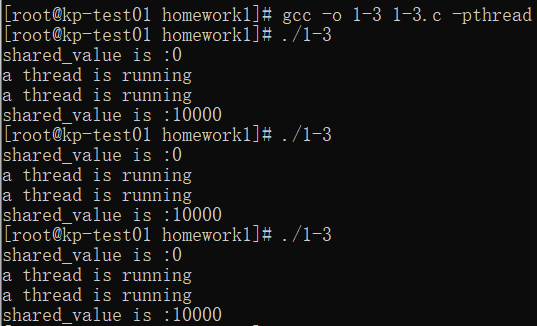
补充缺失的主函数，完整代码如下

#include **<stdio.h>**  
#include **<pthread.h>**  
*// 定义自旋锁结构体*  
**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;  
 printf("a thread is running\n");  
 **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*  
 printf("shared\_value is :%d\n",shared\_value);  
 *// 初始化自旋锁*  
 spinlock\_init(&lock);*//传地址，修改锁*   
 *// 创建两个线程*  
 pthread\_create(&thread1,NULL,(void \*)thread\_function,(void \*)&lock);  
 pthread\_create(&thread2,NULL,(void \*)thread\_function,(void \*)&lock);  
 *// 等待线程结束*  
 pthread\_join(thread1,NULL);  
 pthread\_join(thread2,NULL);  
 *// 输出共享变量的值*  
 printf("shared\_value is :%d\n",shared\_value);  
 **return** 0;  
}

由于我没有创建thread默认的输出，因此这里手动printf()表示线程运行状态

1. 运行并分析

多次运行如下



这里的结果和上文中经过PV操作的结果类似，此处通过spinlock机制，在对共享数据shared\_value访问之前，都进行了如下操作

spinlock\_lock(lock);  
 shared\_value++;  
 spinlock\_unlock(lock);

即，先将启用自旋锁，若可以访问则以原子方式修改变量，然后解锁。若锁已经被占用则原地等待，从而达到了同时只有一个线程对共享变量修改，因此这里的输出结果：在双方访问之前是0，而经过无竞争的随机先后访问后，shared\_value值最终变成可预料的2\*5000=10000

## 实验总结

1. 进程相关实验

1.1 进程创建和同步

fork()函数： 通过fork()创建子进程，子进程继承父进程的内存映像。在父子进程中，可以通过返回值的不同区分它们的执行流。

wait()函数： 用于父进程等待子进程的终止，避免子进程成为孤儿进程。

共享变量和地址空间： 父子进程之间共享初始的地址空间，但之后的修改不会相互影响。全局变量在父子进程中有相同的虚拟地址，但可能对应不同的物理地址。

1.2 系统调用和exec簇函数

system()函数： 创建一个新的进程来执行指定的命令，并在执行完成后返回到原中断点。

exec簇函数： 在当前进程中执行新的程序，不会返回到原中断点，而是直接执行新程序。

2. 线程相关实验

2.1 线程创建和同步

pthread库： 用于创建和管理线程。

共享变量和互斥： 线程之间的共享变量需要通过互斥锁进行保护，避免竞态条件。

线程的创建和结束： 使用pthread\_create()创建线程，pthread\_exit()结束线程。

2.2 系统调用和exec簇函数在线程中的应用

在线程中调用system和exec： 在线程中调用这两类函数会有不同的行为。system()会创建一个新的进程来执行，而exec簇函数会在当前线程中直接执行，并覆盖原有的程序。

3. 自旋锁实验

自旋锁： 通过自旋锁（spinlock）实现对共享资源的互斥访问。自旋锁是一种忙等待的锁，线程会一直循环等待直到获得锁。

\_\_sync\_lock\_test\_and\_set()和\_\_sync\_lock\_release()： 这两个内建函数用于实现原子操作，确保在多线程环境中的互斥性。

4. 实验总结

进程和线程： 进程和线程是操作系统中用于执行程序的基本单位，它们之间有一些关键的区别，包括资源开销、通信方式等。

同步和互斥： 在多线程或多进程的环境中，通过同步和互斥机制来协调各个执行单元，防止竞态条件和数据不一致性。

系统调用和exec簇函数： 理解这两类函数的差异和使用场景，能够灵活地选择适合的方式进行程序执行。

自旋锁： 了解自旋锁的基本原理和使用方式，以及它与其他锁的区别，能够在需要时实现简单的互斥保护。

总体而言，本次实验涵盖了操作系统中的基本概念和编程技巧，加深了对进程、线程、同步、互斥等概念的理解，并通过实际编码体验了不同的操作系统编程场景。

### 实验中的问题与解决过程

问题： 在使用fork()创建子进程时，输出的结果出现了意外的交错现象，导致难以观察wait()函数的效果。

解决过程： 调整代码结构，将wait()函数移到输出之前，确保父进程在子进程执行完毕之前会被阻塞等待。这样可以清晰地观察到wait()函数的效果，以及父进程在子进程执行结束后继续执行的过程。

问题： 在修改全局变量的值并观察父子进程之间的关系时，发现父子进程的全局变量值虽然在同一虚拟地址上，但修改不会相互影响。

解决过程： 通过了解操作系统的内存管理机制，发现父子进程在修改全局变量前共享同一虚拟地址，但一旦有一方修改，操作系统会将其复制为另一个副本，修改后再映射回进程。这解释了为什么修改不会相互影响。

### 实验收获

深入理解进程和线程： 通过实际编码实验，更深入地理解了进程和线程的概念，以及它们在操作系统中的基本原理。

掌握同步与互斥机制： 通过使用互斥锁、信号量、自旋锁等机制，加深了对多线程编程中同步与互斥的理解，以及如何应用这些机制来保证共享资源的正确访问。

熟悉系统调用和exec簇函数： 实验中对system()和exec函数族的使用进行了实践，理解了它们的不同行为和适用场景。

理解共享变量的管理： 通过在多线程环境中修改共享变量，并使用自旋锁进行保护，学习了如何避免竞态条件和确保数据一致性。

### 意见与建议

实验初期，课程内的进度不足，建议实验多写一些可能涉及到的知识点，哪怕仅仅是个链接供参考，否则实验参考书本身书写的语言有些简练，硬挖掘可能存在的问题是不合理的。建议实验参考书能多写一些可以拓展的关键词供学生探索。

## 附件

ReadMe文档



代码



# 进程通信与内存管理

## 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。

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。

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。

## 实验内容

**软中断通信**

（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!!

（3）多次运行所写程序，比较5s内按下Ctrl+\或Ctrl+ c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。

**管道通信**

（1）学习man 命令的用法，通过它查看管道创建、同步互斥系统调用的在线帮助，并阅读参考资料。

（2）根据流程图（如图2.2所示）和所给管道通信程序，按照注释里的要求把代码补充完整，运行程序，体会互斥锁的作用，比较有锁和无锁程序的运行结果，分析管道通信是如何实现同步与互斥的。

**内存分配与回收**

（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。

## 实验思想

1. 软中断通信

1.1 Fork()函数的理解

目的： 通过使用fork()函数创建子进程，理解新进程的产生和父子进程之间的关系。

实现思路： fork()函数创建新进程，新进程是原进程的副本。通过获取fork()的返回值，可以在父子进程中分别执行不同的代码。

1.2 信号处理和软中断

目的： 了解信号处理机制，通过使用signal()函数设置信号处理函数，捕捉键盘发出的中断信号。

实现思路： 使用signal()函数捕捉键盘中断信号（如delete键或quit键）。当捕捉到中断信号后，通过kill()函数向两个子进程发送特定的软中断信号。

1.3 多进程通信与等待

目的： 学习多进程间的通信和等待机制。

实现思路： 父进程通过wait()函数等待两个子进程的结束，获取子进程的终止状态。子进程在接收到软中断信号后输出特定信息并终止。

1.4 信号处理与闹钟中断

目的： 通过使用14号信号值进行闹钟中断，理解不同中断的执行样式。

实现思路： 使用signal()函数捕捉14号信号，并在父进程中发送14号信号。观察父进程和子进程对14号信号的不同响应，深入理解信号处理的机制。

2. 管道通信

2.1 管道的创建与使用

目的： 学习管道的创建和使用，理解管道通信的同步和互斥机制。

实现思路： 使用pipe()函数创建管道，使用fork()函数创建子进程。在父子进程中通过管道进行通信，通过互斥锁保证管道通信的同步和互斥。

2.2 互斥锁的作用

目的： 了解互斥锁的作用和使用方式。

实现思路： 在管道通信的实验中，使用互斥锁保证对共享资源的互斥访问。通过比较有锁和无锁程序的运行结果，分析互斥锁是如何实现同步与互斥的。

3. 内存分配与回收

3.1 内存分配算法的理解

目的： 深入理解内存分配算法，包括首次适应（FF）、最佳适应（BF）、最差适应（WF）等算法。

实现思路： 定义相应的数据结构，实现内存分配算法的关键功能，包括分配、回收、空闲块排序、合并和紧缩等。

3.2 模拟三种算法的实现过程

目的： 通过模拟实现三种不同的内存分配算法，深入理解它们的工作原理。

实现思路： 分别实现首次适应、最佳适应、最差适应算法，通过模拟分配和回收过程，观察它们在不同场景下的表现。

3.3 优劣分析

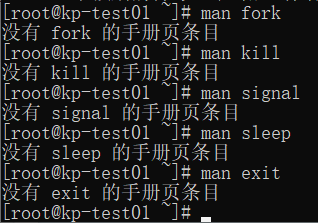
目的： 对比三种算法的优劣，了解它们在不同情境下的性能表现。

实现思路： 通过对比分析每种算法的内存分配效率、内存利用率等指标，总结它们的优缺点，为选择合适的算法提供依据。

## 实验步骤及结果分析

*2.1 进程的软中断通信*

**(1)** **man命令**

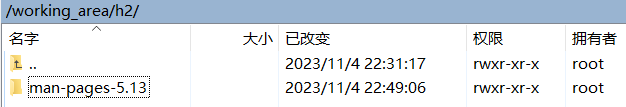


服务器上的库里什么都没有，需要安装C语言库函数的man手册页

先在该网页下载源码

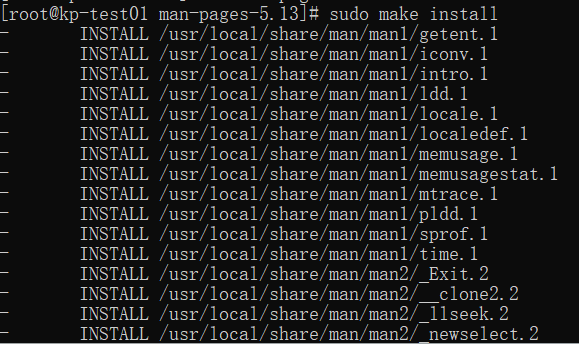
<http://www.kernel.org/pub/linux/docs/man-pages/man-pages-5.13.tar.xz>

上传至服务器



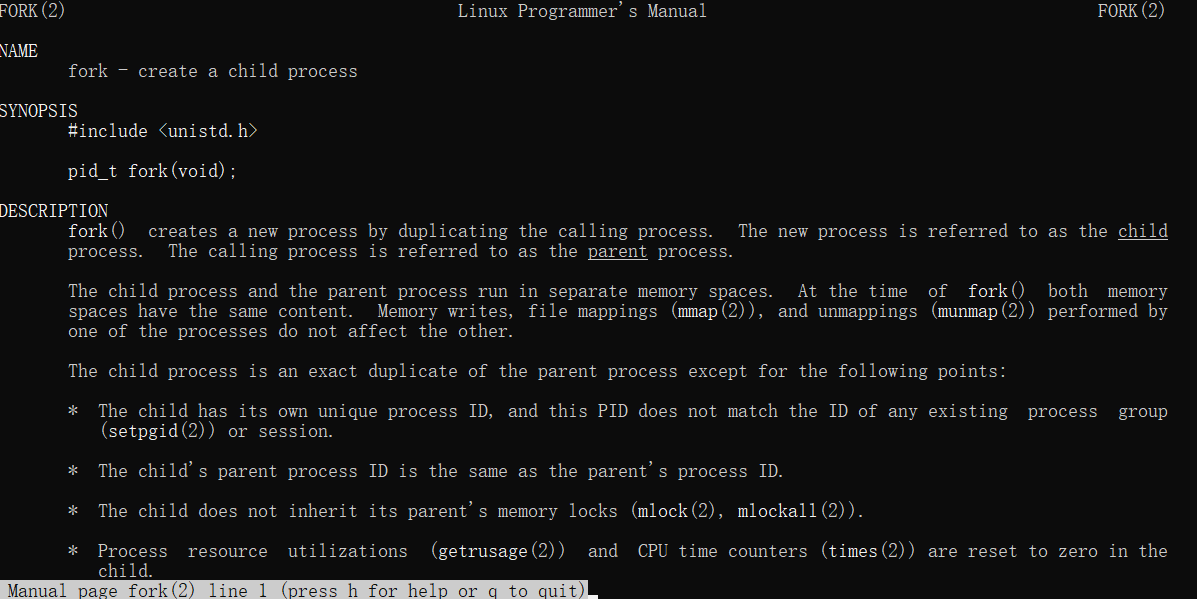
将man手册安装到/user/local/share/man目录下

sudo make install

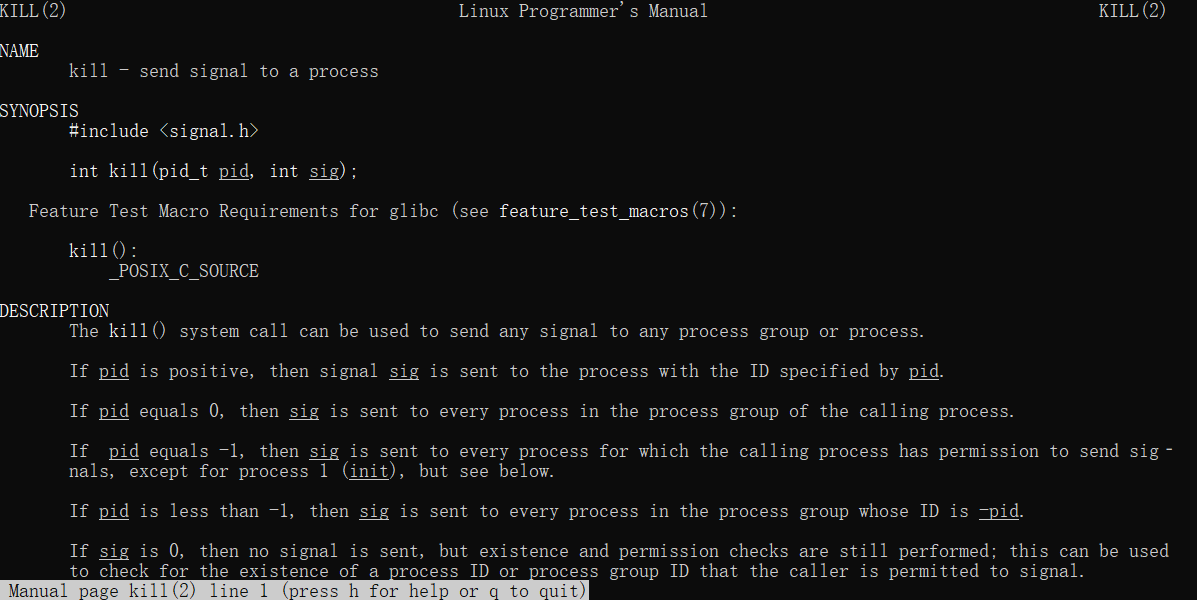


再次尝试man命令查询

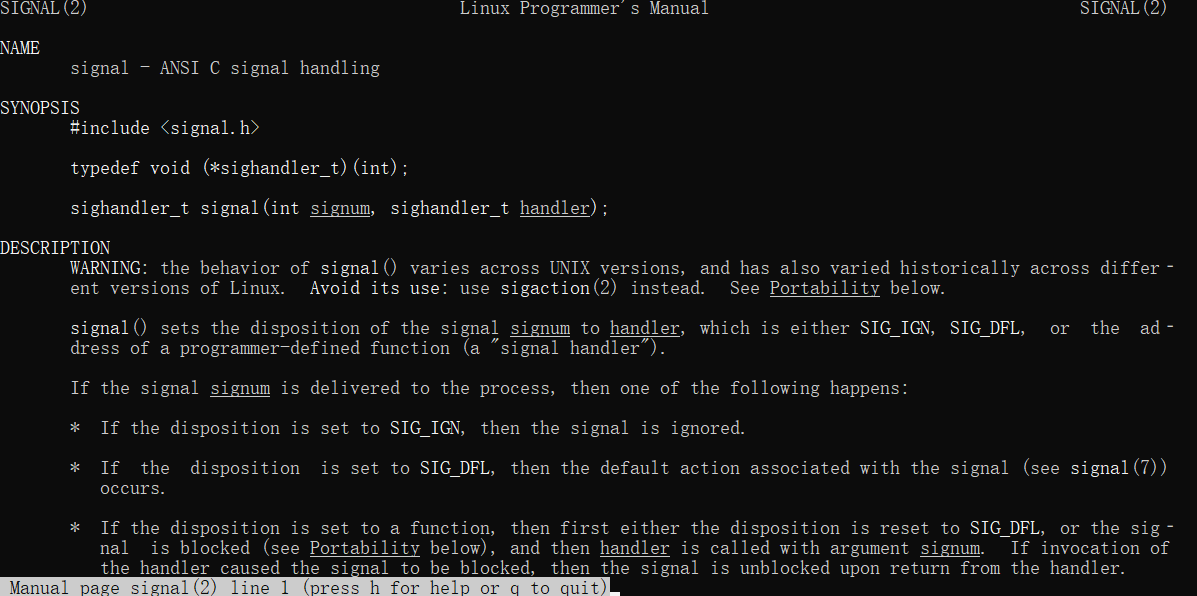
man fork



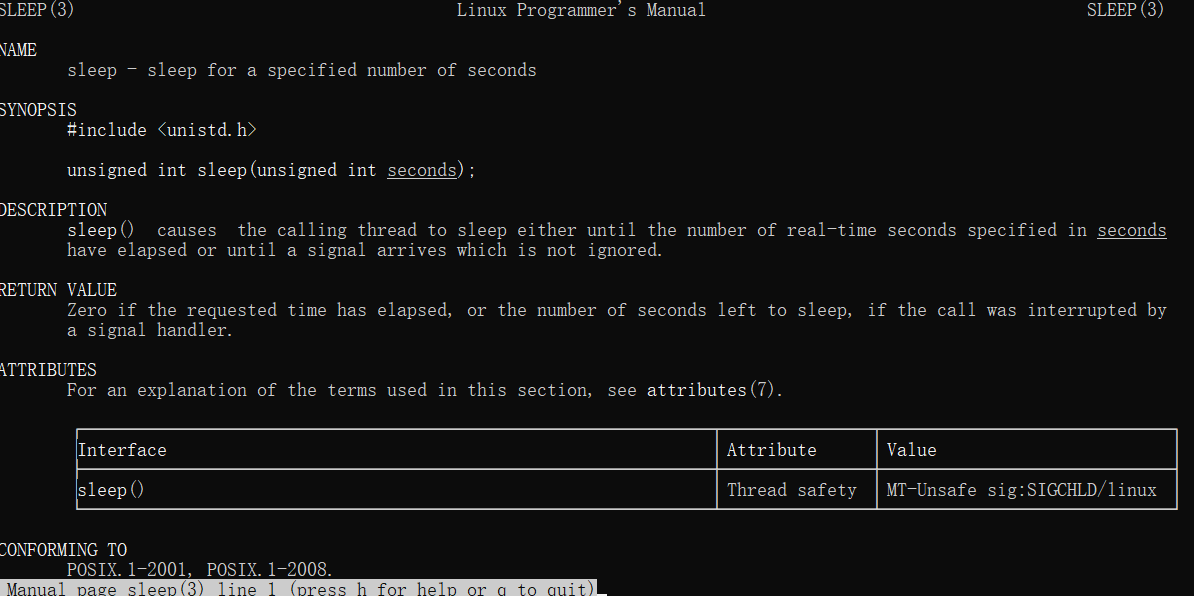
man kill



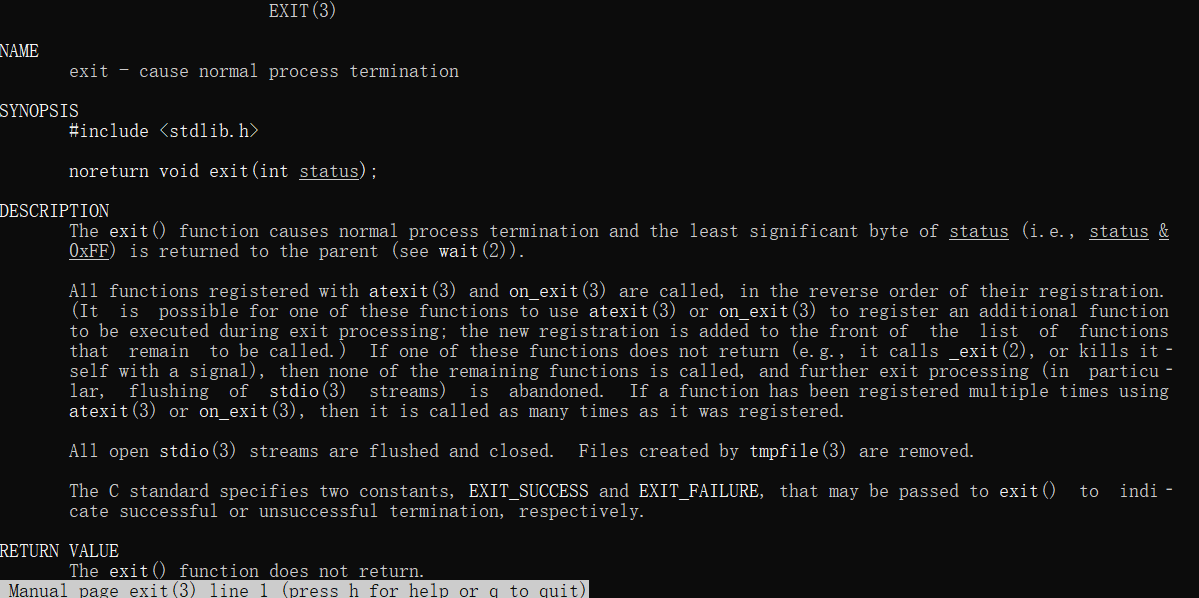
man signal



man sleep

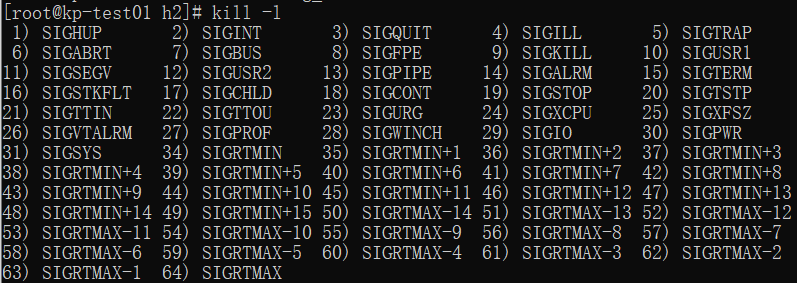


man exit



**(2).** 软中断通信

首先使用kill -l查看信号对应编号



父进程接收的中断SIGINT和SIGQUIT分别是2和3，向子进程传递的中断信号是16和17，其中16和17号信号含义如下：

* *SIGSTKFLT：这是一个信号名，STKFLT是stack fault的缩写，对应的数值为161。当设备内存耗尽，特别是在底层的Linux操作系统层中使用malloc()调用时，可能会发生SIGSTKFLT2。当操作系统无法在malloc中返回失败时，它会生成这种类型的致命信号2。*
* *SIGCHLD：当一个进程正常或异常终止时，内核就会向其父进程发送SIGCHLD信号3。这是因为子进程是一个异步事件，所以这种信号也是内核向父进程发送的异步通知3。*

完整代码如下

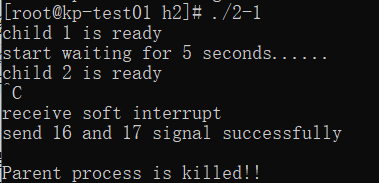
#include **<stdio.h>**  
#include **<unistd.h>**  
#include **<sys/wait.h>**  
#include **<stdlib.h>**  
#include **<signal.h>**  
int flag=0;  
pid\_t pid1=-1, pid2=-1;  
void inter\_handler() {  
 printf("\nreceive soft interrupt\n");  
 kill(pid1,16); *//事实上子进程也执行此处，但PID=0,空操作*  
 kill(pid2,17);  
 printf("send 16 and 17 signal successfully\n");  
 sleep(2); *//给子进程处理信号的时间*  
}  
void handler1(){  
 printf("子进程1收到中断信号\n");  
 flag=1;  
}  
void handler2(){  
 printf("子进程2收到中断信号\n");  
 flag=1;  
}  
void waiting(){  
 **while**(!flag)sleep(1);  
}  
int main() {  
 *// 五秒之后或接收到两个信号*  
 **while** (pid1 == -1)pid1=fork(); *//创建子进程1*  
 **if** (pid1 > 0) {*//父进程*  
 **while** (pid2 == -1)pid2=fork(); *//创建子进程2*  
 **if** (pid2 > 0) { *// 父进程，等待软中断*  
 signal(SIGINT,inter\_handler); *//捕获SIGINT*  
 signal(SIGQUIT,inter\_handler); *//捕获SIGQUIT*  
 printf("start waiting for 5 seconds......\n");  
 sleep(5);  
 **while**(wait(NULL)!=-1)sleep(1);  
 printf("\nParent process is killed!!\n");  
 }   
 **else** { *// 子进程 2*  
 signal(17,handler2);  
 printf("child 2 is ready\n");  
 waiting(); *//阻塞子进程*  
 printf("\nChild process2 is killed by parent!!\n");  
 sleep(1);  
 **return** 0;  
 }  
 }  
 **else** { *// 子进程 1*  
 signal(16,handler1);  
 printf("child 1 is ready\n");  
 waiting();  
 printf("\nChild process1 is killed by parent!!\n");  
 sleep(1);  
 **return** 0;  
 }  
 **return** 0;  
}

其中，

* 为了确保子进程准备好接收信号，在子进程signal设置完毕后输出ready信号。
* 为了阻塞住子进程，调用pause函数或者调用waiting(通过flag信号不断while循环），让子进程处于忙等待软中断信号发生

ctrl + c 为delete(SIGINT)， ctrl + \为 quit(SIGQUIT)

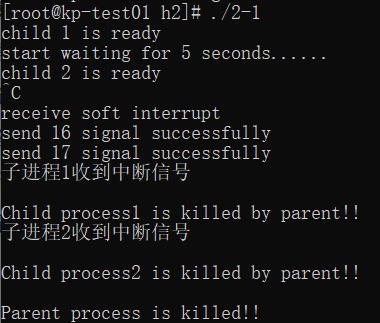
运行程序



发现子进程根本不会输出，重新思考后，发现是我按下SIGINT信号后父子进程都会收到这个信号，理论上父进程收到该信号是可预料的，然而子进程在收到该信号后会直接结束进程，因此需要让子进程具备接受该信号的能力，索性将父进程的接受signal直接放到创建进程之前，子进程执行inter\_handler实则进行空操作，无影响。修改后代码如下：

#include **<stdio.h>**  
#include **<unistd.h>**  
#include **<sys/wait.h>**  
#include **<stdlib.h>**  
#include **<signal.h>**  
int flag=0;  
pid\_t pid1=-1, pid2=-1;  
void inter\_handler() {  
 **if**(pid1<=0||pid2<=0)**return**; *//子进程空操作*  
 printf("\nreceive soft interrupt\n");  
 kill(pid1,16);   
 printf("send 16 signal successfully\n");  
 kill(pid2,17);  
 printf("send 17 signal successfully\n");  
}  
void handler1(){  
 printf("子进程1收到中断信号\n");  
 flag=1;  
}  
void handler2(){  
 printf("子进程2收到中断信号\n");  
 flag=1;  
}  
void waiting(){  
 **while**(!flag);  
}  
int main() {  
 *// 五秒之后或接收到两个信号*  
 signal(SIGINT,inter\_handler); *//捕获SIGINT*  
 signal(SIGQUIT,inter\_handler); *//捕获SIGQUIT*  
 **while** (pid1 == -1)pid1=fork(); *//创建子进程1*  
 **if** (pid1 > 0) {*//父进程*  
 **while** (pid2 == -1)pid2=fork(); *//创建子进程2*  
 **if** (pid2 > 0) { *// 父进程，等待软中断*  
 printf("start waiting for 5 seconds......\n");  
 sleep(5);  
 **while**(wait(NULL)!=-1)sleep(1);  
 printf("\nParent process is killed!!\n");  
 }   
 **else** { *// 子进程 2*  
 signal(17,handler2);  
 printf("child 2 is ready\n");  
 waiting(); *//阻塞子进程*  
 printf("\nChild process2 is killed by parent!!\n");  
 exit(0);  
 }  
 }  
 **else** { *// 子进程 1*  
 signal(16,handler1);  
 printf("child 1 is ready\n");  
 waiting();  
 printf("\nChild process1 is killed by parent!!\n");  
 exit(0);  
 }  
 **return** 0;  
}

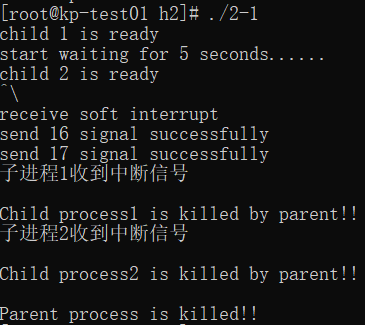
重新运行后如图：



在child 1和child 2就绪后，发送SIGINT信号，父进程发出16和17信号，子进程收到后分别输出shut down语句，最后父进程结束。

**(3)** 比较发送中断与不操作

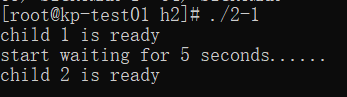
* 按下Ctrl+\或Ctrl+c（delete)



和（2）中的结果保持一致，即父进程发出16，17信号，子进程接收后进行输出，然后父进程等待子进程结束，最后父进程结束。

* 5秒内不进行任何操作

**A.** 如果没有添加超时处理且父进程有阻塞状态，在源代码上运行后结果如图



由于父进程必须等待子进程结束，而子进程始终在等待信号，因此不会继续推进，原地踏步。

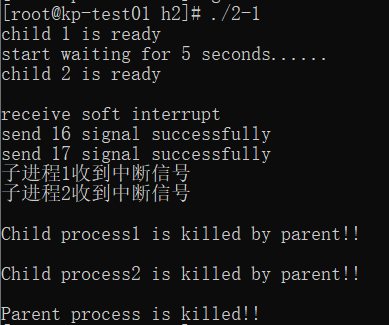
**B.** 如果添加了超时软中断SIGALRM，这时需要使用alarm函数，如果是接收该信号，即为（4）中需要设计的，如果没有接收而仅仅使用延迟，导致超时后等待子进程结束，则又处于阻塞状态。

**(4)** 将sleep语句修改为alarm语句，并添加中断的处理函数，代码如下

#include **<stdio.h>**  
#include **<unistd.h>**  
#include **<sys/wait.h>**  
#include **<stdlib.h>**  
#include **<signal.h>**  
int flag=0;  
pid\_t pid1=-1, pid2=-1;  
void inter\_handler() {  
 **if**(pid1<=0||pid2<=0)**return**; *//子进程空操作*  
 printf("\nreceive soft interrupt\n");  
 kill(pid1,16);   
 printf("send 16 signal successfully\n");  
 kill(pid2,17);  
 printf("send 17 signal successfully\n");  
}  
  
void handler1(){  
 printf("子进程1收到中断信号\n");  
 flag=1;  
}  
void handler2(){  
 printf("子进程2收到中断信号\n");  
 flag=1;  
}  
void waiting(){  
 **while**(!flag);  
}  
int main() {  
 *// 五秒之后或接收到两个信号*  
 signal(SIGALRM,inter\_handler); *//捕获SIGALRM*  
 signal(SIGINT,inter\_handler); *//捕获SIGINT*  
 signal(SIGQUIT,inter\_handler); *//捕获SIGQUIT*  
 **while** (pid1 == -1)pid1=fork(); *//创建子进程1*  
 **if** (pid1 > 0) {*//父进程*  
 **while** (pid2 == -1)pid2=fork(); *//创建子进程2*  
 **if** (pid2 > 0) { *// 父进程，等待软中断*  
 printf("start waiting for 5 seconds......\n");  
 alarm(5); *//设置定时器5秒*  
 pause();  
 **while**(wait(NULL)!=-1)sleep(1);  
 printf("\nParent process is killed!!\n");  
 }   
 **else** { *// 子进程 2*  
 signal(17,handler2);  
 printf("child 2 is ready\n");  
 waiting(); *//阻塞子进程*  
 printf("\nChild process2 is killed by parent!!\n");  
 exit(0);  
 }  
 }  
 **else** { *// 子进程 1*  
 signal(16,handler1);  
 printf("child 1 is ready\n");  
 waiting();  
 printf("\nChild process1 is killed by parent!!\n");  
 exit(0);  
 }  
 **return** 0;  
}

超时信号由SIGALRM表示，此时仍然需要父进程向子进程发送相同的信号，因此signal参数中仍然是inter\_handler即可。

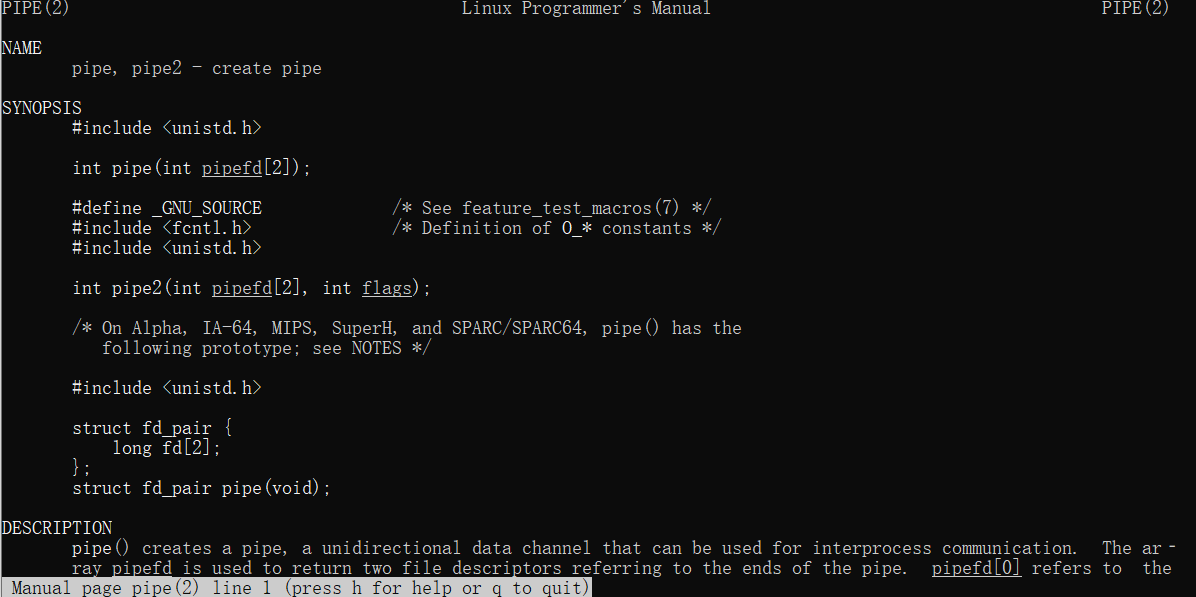
运行如图



可以观察到此时5秒内，没有任何键盘输入中断信号，alarm(5)超时后，发出SIGALRM信号，执行kill语句，之后重复（2）的过程。

***2.2 进程的管道通信***

1.使用man命令查看管道创建和同步互斥系统调用



先声明一下对管道使用读写的方式：

* [write() 函数1](https://www.tutorialandexample.com/write-function-in-c)：write() 函数用于将缓冲区的内容写入到一个已声明的输出或流中。它的原型如下：
* ssize\_t write(int fd, const void \*buf, size\_t count);
* 其中，fd 是文件描述符，buf 是指向要写入数据的缓冲区的指针，count 是要写入的字节数。此函数会返回实际写入的字节数。如果返回值与 count 不同，通常表示发生了错误。
* [read() 函数](https://www.tutorialandexample.com/write-function-in-c)[2](https://man7.org/linux/man-pages/man2/read.2.html)：read() 函数用于从文件或流中读取数据。它的原型如下：
* ssize\_t read(int fd, void \*buf, size\_t count);
* 其中，fd 是文件描述符，buf 是指向用于存储读取数据的缓冲区的指针，count 是要读取的字节数。此函数会返回实际读取的字节数。如果返回值与 count 不同，可能是因为已经到达了文件末尾，或者在读取过程中发生了错误。

2.2管道通信程序

由于此处使用的是匿名管道的通信方式，因此确认对方的问题不再考虑，通过程序创建进程是否成功来判定，不再赘述。后续根据是否考虑互斥问题和同步问题分别描述。

2.2.1 首先是不写互斥锁，不处理同步问题

代码如下

#include **<stdio.h>**  
#include **<unistd.h>**  
#include **<sys/wait.h>**  
#include **<stdlib.h>**  
#include **<signal.h>**  
#include **<string.h>**  
int pid1,pid2; *// 定义两个进程变量*  
int main(){  
 int fd[2];   
 char InPipe[10000]={0}; *// 定义读缓冲区,并初始化*  
 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*  
 **if**(pid1 == 0) { *// 如果子进程 1 创建成功,pid1 为进程号*  
 *//补充； // 锁定管道*  
 **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*  
 *//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);*   
 **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** {   
 waitpid(pid1,NULL,0); *// 等待子进程 1 结束*  
 waitpid(pid2,NULL,0); *// 等待子进程 2 结束*  
 read(fd[0],InPipe,4000); *// 从管道中读出 4000 个字符*  
 int len = strlen(InPipe); *//统计写入字符个数*  
 InPipe[len]='\0'; *// 加字符串结束符*  
 printf("\nword counts = %d\n",len);  
 printf("%s\n",InPipe); *// 显示读出的数据*  
 exit(0); *// 父进程结束*  
 }   
 }  
}

运行结果如下



分析：首先读和写是没有冲突的，因为父进程使用waitpid()等待两个子进程的写入，因此，读取数据过程时完整的。其次，字符总个数是正确的（4000），然而输出却是1和2的交替，说明，在一个子进程写入一个字符后，由于没有互斥锁的存在，另一个进程可以直接写入，因此不是我们预期的全1和全2阵列先后出现。

**2.2.2 添加互斥锁，不处理同步问题**

使用lockf函数，定义如下

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,founction;

long size;

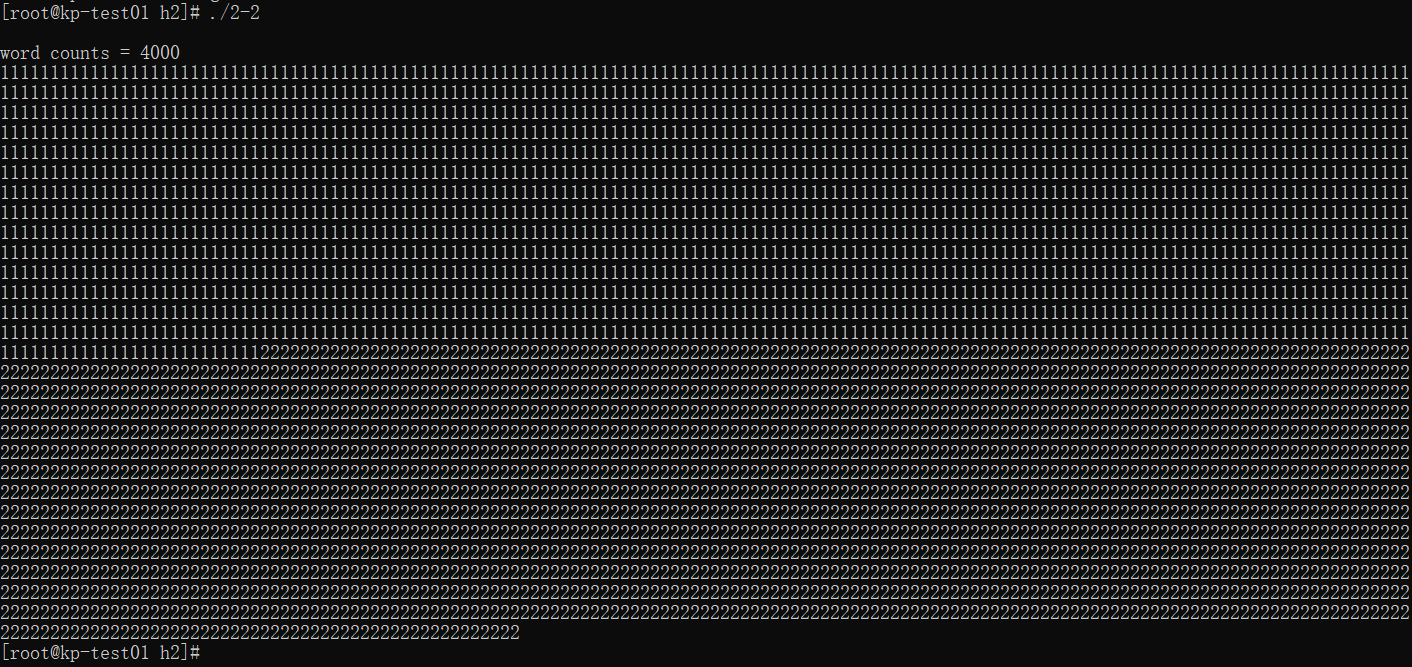
files 是文件描述符，function 表示锁状态，1 表示锁定，0 表示解锁；size 是锁定或解锁的字节数，若为 0 则表示从文件的当前位置到文件尾。

添加互斥锁后代码如下

#include **<stdio.h>**  
#include **<unistd.h>**  
#include **<sys/wait.h>**  
#include **<stdlib.h>**  
#include **<signal.h>**  
#include **<string.h>**  
int pid1,pid2; *// 定义两个进程变量*  
int main(){  
 int fd[2];   
 char InPipe[10000]={0}; *// 定义读缓冲区,并初始化*  
 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*  
 **if**(pid1 == 0) { *// 如果子进程 1 创建成功,pid1 为进程号*  
 lockf(fd[1],1,0);*//补充； // 锁定管道*  
 **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);*//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);   
 **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** {   
 waitpid(pid1,NULL,0); *// 等待子进程 1 结束*  
 waitpid(pid2,NULL,0); *// 等待子进程 2 结束*  
 read(fd[0],InPipe,4000); *// 从管道中读出 4000 个字符*  
 int len = strlen(InPipe); *//统计写入字符个数*  
 InPipe[len]='\0'; *// 加字符串结束符*  
 printf("\nword counts = %d\n",len);  
 printf("%s\n",InPipe); *// 显示读出的数据*  
 exit(0); *// 父进程结束*  
 }   
 }  
}

lockf可以实现互斥访问，通过对管道上锁，当另一进程试图访问时，会阻塞在该处，类似于wait()操作，当互斥锁被释放后进行访问。

运行结果如下



正是我们所预料到的，两个子进程以先后次序，写入2000个1以及2000个2，而没有在时间上交叉，这正是互斥锁保证的。

**2.2.3 添加互斥锁，且保证同步**

在2.2的基础上，为了实现同步，事实上可以采用实验一中的信号量semaphore,通过信号量的P,V操作，实现对有限管道的先后写读访问，此处由于不需要使用同步（因为父进程在waitpid()操作后才执行读操作，因此不会出现次序混乱，如果要实现可以参考实验一，此处不再赘述。

*2.3 内存的分配与回收*

2.3.1 内存分配思路

* **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。核心就是要将空闲块链表按照地址顺序排序。
* **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，此时必然找到最佳适应的最小可用空闲区域。
* **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。

2.3.2 实现算法

空闲链表结构体

**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;

已分配链表结构体

*/\*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;

函数列表

**struct** free\_block\_type\* init\_free\_block(int mem\_size) *//空闲链表初始化*  
void display\_menu() *//菜单*  
int set\_mem\_size() *//设置存储大小*  
void rearrange\_FF() *//FF算法*  
void rearrange\_BF() *//BF算法*  
void rearrange\_WF() *//WF算法*  
void rearrange(int algorithm) *//根据算法分配内存（调用上述3种）*  
void set\_algorithm() *//设置算法*  
int new\_process() *//新建进程*  
void compact\_memory() *//紧缩技术*  
int allocate\_mem(**struct** allocated\_block \*ab) *//分配进程内存*  
**struct** allocated\_block\* find\_process *//寻找进程内存区域*  
void kill\_process() *//结束进程*  
void merge\_free\_blocks() *//合并*  
int free\_mem(**struct** allocated\_block \*ab) *//释放分配块*  
int dispose(**struct** allocated\_block \*free\_ab) *//释放ab数据结构节点*  
int display\_mem\_usage() *//显示内存使用情况*  
void do\_exit() *//删除内存区*

2.3.3 模拟三种算法并对比

首先模拟如下场景

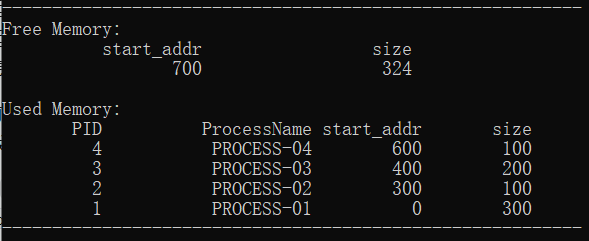
总计1024的内存空间，先后由4个进程（分别占据300,100,200,100)占据一定空间，接着，进程1，3结束，释放空间，得到右上角图所示状态。

接着，遇到新的进程Px，大小为190，按照不同算法，会有不同的装填策略，接下来分别叙述。（设外碎片为不足10的空间大小）

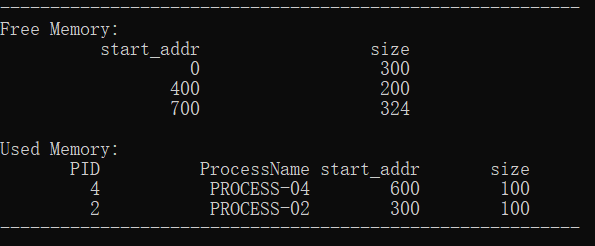
* FF算法：依据该算法，直接找到第一块300大小的内存，进行分割，分割后剩下110大小的内存，模拟如下图所示

选择算法FF

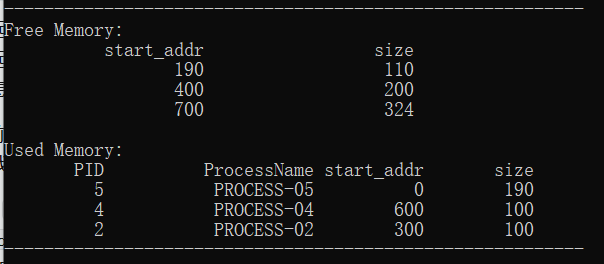
依次创建进程达到初始状态



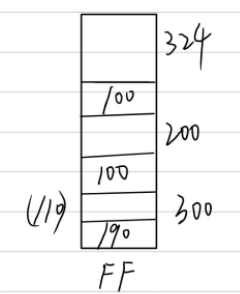
删去1，3进程



按照FF方式装入P5=190



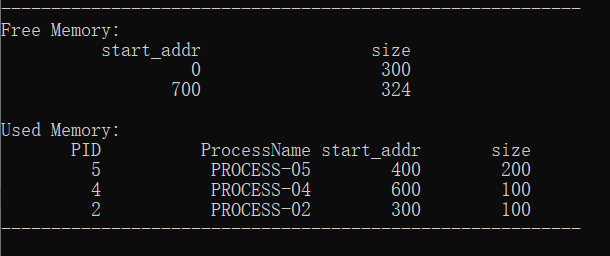
显然，这里的确对应了下图的装填方式



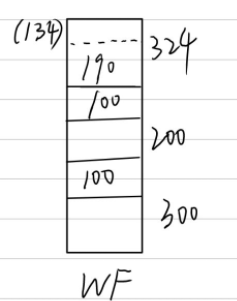
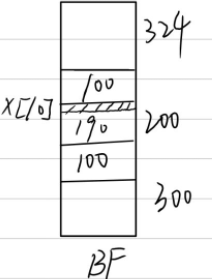
得以验证。

* BF算法：

装入P1-P4进程以及删除P1,P3过程不再重复，接下来直接装入P5，如图

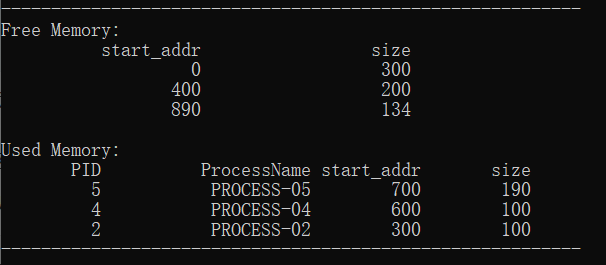


与下图相符，P5=190被装入200大小的空间中，且产生了10大小的不可用碎片空间，在Free Memory中，由于外碎片过小，所以是一起分配的策略，PID=5的size=190+10而非190，且空闲区直接舍弃该碎片，符合预期。



* WF算法：

装入P1-P4进程以及删除P1,P3过程不再重复，接下来直接装入P5，如图



与下图相符，P5直接占据的是起始地址为700的最大的空闲区，并且300-134>10，采用分割的方式，该空闲区大小变成134，符合预期。

## 回答问题

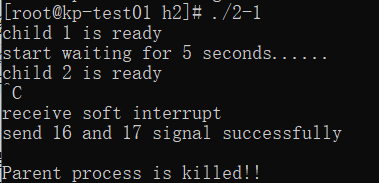
### 2.5.1 软中断通信

1. **你最初认为运行结果会怎么样？写出你猜测的结果。**

最初我认为会按照预期一样，父进程在5秒内收到中断信号后，子进程会受到父进程通过kill传送的信号，因此中断自己的waiting操作，从而进行输出以及结束子进程，当两个子进程结束后父进程结束。

1. **实际的结果什么样？有什么特点？在接收不同中断前后有什么差别？请将5秒内中断和5秒后中断的运行结果截图，并对产生该现象的原因进行分析。**

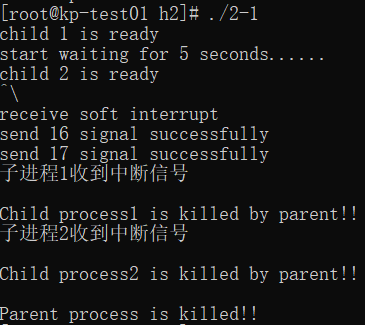
然而事实上遇到了一些问题如图



特点是子进程不会由任何输出，好像从ready后不知道有信号的传送过程一样。

子进程根本不会输出，重新思考后，发现是我按下SIGINT信号后父子进程都会收到这个信号，理论上父进程收到该信号是可预料的，然而子进程在收到该信号后会直接结束进程，因此需要让子进程具备接受该信号的能力，索性将父进程的接受signal直接放到创建进程之前，子进程执行inter\_handler实则进行空操作，无影响。

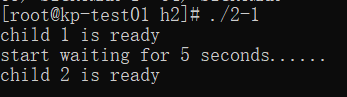
按下Ctrl+\或Ctrl+c（delete)



和（2）中的结果保持一致，即父进程发出16，17信号，子进程接收后进行输出，然后父进程等待子进程结束，最后父进程结束。

5秒内不进行任何操作

A. 如果没有添加超时处理且父进程有阻塞状态，在源代码上运行后结果如图



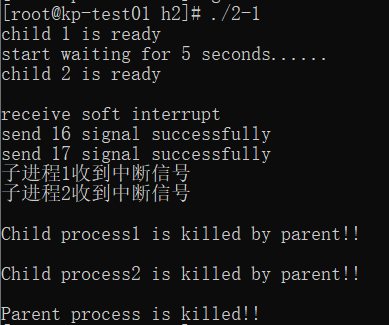
由于父进程必须等待子进程结束，而子进程始终在等待信号，因此不会继续推进，原地踏步。

B. 如果添加了超时软中断SIGALRM，这时需要使用alarm函数，如果是接收该信号，即为（4）中需要设计的，如果没有接收而仅仅使用延迟，导致超时后等待子进程结束，则又处于阻塞状态。

1. **改为闹钟中断后，程序运行的结果是什么样子？与之前有什么不同？**

超时信号由SIGALRM表示，此时仍然需要父进程向子进程发送相同的信号，因此signal参数中仍然是inter\_handler即可。

运行如图



可以观察到此时5秒内，没有任何键盘输入中断信号，alarm(5)超时后，发出SIGALRM信号，执行kill语句，之后重复（2）的过程。

1. **kill 命令在程序中使用了几次？每次的作用是什么？执行后的现象是什么？**

在没有用到alarm之前，只需要用到2次kill命令，在使用alarm信号后，需要4次kill命令，分别如下：

使用alarm前：

父进程向子进程kill两个信号（16和17），此时子进程的ready是通过printf显示出来的结果，因为是手动的发送控制信号，因此无影响。

使用alarm后：

前两个信号同上，还需要子进程向父进程发送两个额外的信号，用来告诉父进程子进程的ready状态。

**（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？**

进程可以通过调用 exit 函数来主动退出。这是一种正常退出的方式，它会执行一些清理工作、释放资源等。相比之下，使用 kill 命令是一种外部强制终止的方式，可能导致进程无法完成一些必要的清理工作，可能会引起资源泄露或不正常的退出。

因此，一般情况下，推荐使用 exit 函数来主动退出进程。这样能够确保程序在退出时执行必要的清理工作，释放占用的资源，保证程序的正常运行。外部强制终止进程的方式一般应该被视为一种紧急手段，只在必要的情况下使用，因为它可能会导致一些不可预测的问题。

### 2.5.2 管道通信

**(1)你最初认为运行结果会怎么样？**

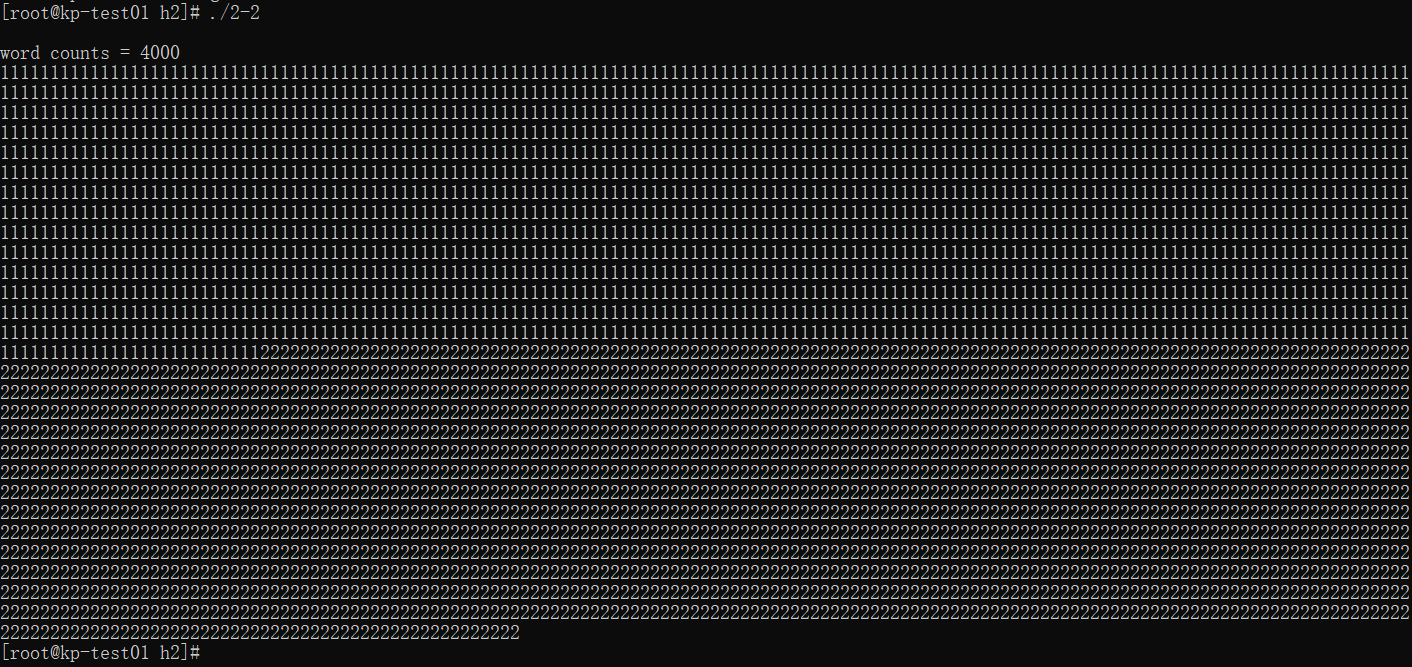
最初我认为，在没有实现互斥的情况下，写入的内容应该是乱序的，并且还可能发生同时写入操作的冲突导致发生字符缺失等问题。在实现互斥后，1和2是有先后顺序的。

1. **实际的结果什么样？有什么特点？试对产生该现象的原因进行分析。**

不处理同步和互斥时：

首先读和写是没有冲突的，因为父进程使用waitpid()等待两个子进程的写入，因此，读取数据过程时完整的。其次，字符总个数是正确的（4000），然而输出却是1和2的交替，说明，在一个子进程写入一个字符后，由于没有互斥锁的存在，另一个进程可以直接写入，因此不是我们预期的全1和全2阵列先后出现。

添加互斥锁后：运行结果如下



正是我们所预料到的，两个子进程以先后次序，写入2000个1以及2000个2，而没有在时间上交叉，这正是互斥锁保证的。

**(3)实验中管道通信是怎样实现同步与互斥的？如果不控制同步与互斥会发生什么后果？**

在实验中，管道通信的同步与互斥是通过操作系统提供的系统调用和文件锁定机制来实现的。主要的系统调用有pipe()、read()、write()等，而文件锁定机制则可以使用flock()、lockf()等。

同步机制的作用是确保进程按照一定的顺序进行读写操作，以保证数据的正确性。如果多个进程同时向管道写入数据或同时从管道读取数据，可能会导致数据混乱，不符合预期的结果。

互斥机制的作用是防止多个进程同时进行写操作，以避免竞争条件。如果多个进程同时向管道写入数据，可能会导致数据覆盖或丢失。

如果不控制同步与互斥，可能会导致以下问题：

数据混乱： 多个进程同时写入数据时，由于操作系统调度的不确定性，数据的顺序可能不一致，导致读取时数据混乱。

竞争条件： 多个进程同时进行写操作时，由于没有互斥机制，可能发生竞争条件，导致数据不一致或错误。

数据丢失： 如果多个进程同时向管道写入数据，可能会导致部分数据被覆盖，造成数据丢失。

为了解决这些问题，需要在并发读写的情况下使用同步和互斥机制，确保数据的正确性和一致性。在实验中，通过使用文件锁定机制（如lockf()）或其他同步工具，可以有效地控制进程的读写操作，从而实现同步与互斥。

## 实验总结

**内存的分配与回收**

1. **对涉及的3个算法进行比较，包括算法思想、算法的优缺点、在实现上如何提高算法的查找性能。**

首次适应算法（FF）:

算法思想: 从空闲分区链表的首部开始查找，找到第一个满足大小的空闲分区进行分配。

优点: 简单，易于实现。

缺点: 可能导致外碎片，影响后续大空间进程的分配。性能较差，容易产生碎片。

提高查找性能: 可以考虑使用空闲分区链表按照地址递增次序排列。

最佳适应算法（BF）:

算法思想: 从空闲分区链表中找到第一个能满足进程大小要求的最小空闲分区进行分配。

优点: 尽量减少外碎片，相对于FF更能保持空间的连续性。

缺点: 容易产生较小的外碎片，需要搭配紧缩技术。

提高查找性能: 空闲分区链表按分区大小由小到大排序。

最差匹配算法（WF）:

算法思想: 从空闲分区链表中找到第一个能满足进程大小要求的最大空闲分区进行分配。

优点: 避免了外碎片，每次占据最大空间。

缺点: 会导致大空间的进程无法加载，可能效率较低。

提高查找性能: 空闲分区链表按分区大小由大到小排序。

总结比较：

FF算法简单，但容易产生外碎片，性能相对较差。

BF算法在空间利用率上相对较好，但容易产生小碎片，需要额外处理。

WF算法能最大程度上避免外碎片，但可能导致大进程无法加载，效率相对较低。

分析三种算法的优劣

事实上，三种算法都不能保证任何情况下的全局最优解，但各自在一定情况下具有一定优势，而在其他情况下可能处理方式比较糟糕，下面分别进行描述：

FF算法：最朴素的方式，按照地址的方式查询是否可用，优点是实现简单，并且当程序是按照顺序存储时表现一般比较好，但缺点是，当比较极端的存储空间大小（如极大）提前出现，会直接占据大空间，可能不是最fit的，造成后续的大空间进程无法加载进入内存，这时就类似于退化成WF算法。

BF算法：该算法是最符合直觉的近似最优分配方式，直接找最小且满足的区域进行分配，优点是在大多数情况下可以保证对空间足够的节约，不会强行分割大空间，而是优先利用小资源。缺点是，对每一个看似恰好合适的空间，更容易产生无法使用的外碎片，需要搭配紧缩技术。

WF算法：虽然名字叫worst fit，但不代表wf算法是最糟糕的，每一次使用最大的空间，在空间比较充足的情况下，可以避免像BF算法带来的外碎片，因为空间几乎每次都是可分的。但是缺点也是十分明显的，每次占据大空间，会导致原本可以装入内存的大进程无法加载，极端情况下可能会导致装入的进程个数远远小于理论值。

1. **3种算法的空闲块排序分别是如何实现的。**

首次适应算法（FF）: 空闲分区链表按照地址递增次序排列。

最佳适应算法（BF）: 空闲分区链表按分区大小由小到大排序。

最差匹配算法（WF）: 空闲分区链表按分区大小由大到小排序。

1. **结合实验，举例说明什么是内碎片、外碎片，紧缩功能解决的是什么碎片。**

内碎片: 一般只出现在固定分配的情况下，当分配的内存块大于进程实际需要的大小时，多余的部分就是内碎片。例如，一个进程需要分配100个字节，而分配的内存块大小是120个字节，那么就有20个字节的内碎片。

外碎片: 一般出现在动态分配的情况中，有“孔”的概念，当空闲内存块无法被利用，无法容纳新进程或大进程时，产生的零散、不连续的内存空间就是外碎片。

紧缩功能解决的是外碎片: 当进程终止并释放了一块内存时，如果这块内存后面的空间是被占用的，就会产生外碎片。紧缩功能的目的是通过将已分配的内存块移动到一起，从而合并外碎片，使得整体内存空间变得连续。

1. **在回收内存时，空闲块合并是如何实现的。**

在回收内存时，空闲块的合并可以通过以下步骤实现：

释放内存块： 当一个进程终止，释放了一块内存块时，将该内存块对应的节点从已分配链表中移除。

合并相邻空闲块： 检查被释放的内存块前后是否有相邻的空闲块。如果有相邻的空闲块，就可以将它们合并成一个更大的空闲块。

更新空闲链表： 将合并后的空闲块加入到空闲链表中。确保空闲链表中的空闲块按照一定的顺序（如地址递增）排列，以方便后续的内存分配。

通过这样的合并操作，可以降低外碎片的产生，提高内存的利用效率。

### 2.6.1 实验中的问题与解决过程

man手册安装： 通过从 [http://www.kernel.org/pub/linux/docs/man-pages/](http://www.kernel.org/pub/linux/docs/man-pages/" \t "https://chat.openai.com/c/_new) 下载源码，并使用正确的configure、make、sudo make install流程，成功在服务器上安装了C语言库函数的man手册页。

软中断通信： 通过使用kill命令发送软中断信号(SIGINT和SIGQUIT)，并在父子进程中设置信号处理函数，成功实现了进程间的通信。通过修改信号处理函数，子进程得以接收并处理相应的中断信号。

超时处理和信号处理冲突： 通过将信号处理函数的设置提前到进程创建之前，解决了子进程接收信号的能力问题。此外，通过使用alarm函数和信号处理函数，成功实现了在一定时间后发送SIGALRM信号，以及子进程接收并处理该信号的功能。

进程间管道通信： 成功实现了使用管道进行父子进程间的通信，通过pipe函数创建管道，并通过read和write实现了进程间的数据传输。添加了互斥锁（lockf函数）确保了对管道的互斥访问。

内存分配与回收： 实现了三种内存分配算法（首次适应、最佳适应、最差匹配），并通过模拟场景演示了它们的不同行为。成功处理了新建进程、结束进程、释放内存等操作，以及使用互斥锁和紧缩技术的内存管理问题。

### 2.6.2 实验收获

深入理解Linux系统： 通过实验，我深入了解了Linux系统的一些基本概念和操作，包括man手册的安装、进程间通信、信号处理机制、管道通信，以及内存的分配与回收。

信号处理机制： 实践中，我学到了如何使用信号处理机制在父子进程之间进行通信。掌握了信号的发送和接收，以及如何设置信号处理函数，处理各种中断事件。

管道通信和互斥访问： 通过使用管道进行进程间通信，我学到了管道的基本原理和使用方法。在此过程中，我引入了互斥锁（lockf函数），确保了对管道的互斥访问，避免了竞态条件。

内存管理算法： 实现了三种内存分配算法（首次适应、最佳适应、最差匹配），并通过模拟场景演示了它们的不同行为。深入理解了这些算法的优缺点，以及它们在不同情况下的适用性。

超时处理和紧缩技术： 学到了如何使用超时处理，通过设置定时器（alarm函数）来实现对进程的超时控制。另外，了解了内存管理中的紧缩技术，如何优化内存的利用和释放。

解决实际问题的能力： 在实验中，我遇到了一些问题，例如信号处理函数的设置位置、子进程对信号的处理等，通过仔细分析和调整代码，成功解决了这些问题，提高了解决实际问题的能力。

## 附件

### 2.7.1 附件1 程序



### 2.7.2 附件2 ReadMe



# 文件系统

## 实验目的

通过一个简单文件系统的设计，理解文件系统的内部结构、基本功能及实现。

## 实验内容

（1）分析EXT2文件系统的结构；

（2）基于Ext2的思想和算法，设计一个类Ext2的多级文件系统，实现Ext2文件系统的一个功能子集；

（3）用现有操作系统上的文件来代替硬盘进行硬件模拟。

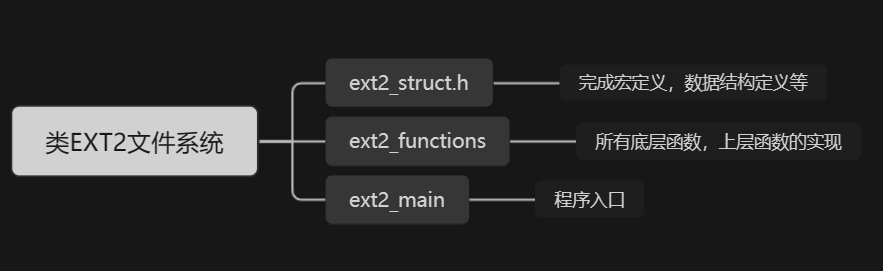
## 实验思想

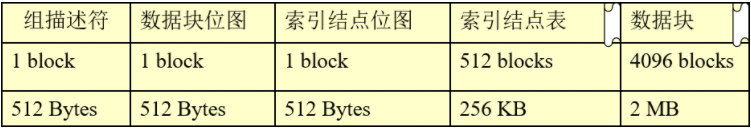
在分析 Linux 的文件系统的基础上，基于Ext2的思想和算法，设计一个类Ext2的虚拟多级文件系统，实现Ext2文件系统的一个功能子集。并且用现有操作系统上的文件来代替硬盘进行硬件模拟。设计文件系统应该考虑的几个层次：①介质的物理结构；②物理操作——设备驱动程序完成；③文件系统的组织结构（逻辑组织结构）；④对组织结构其上的操作；⑤为用户使用文件系统提供的接口。本实验只涉及后三个层次的内容。

## 文件系统架构

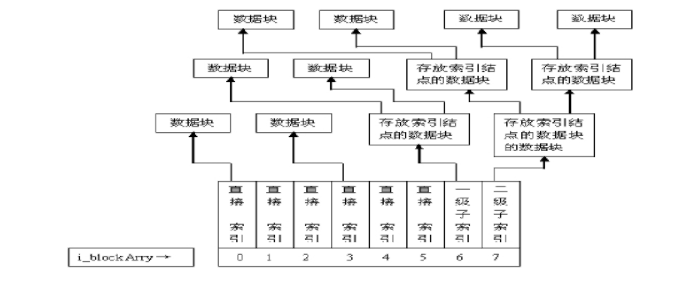
### 结构图

宏观结构图如图





其中，多级索引结构如图



### 主要数据结构的定义

#ifndef EXT2\_Header  
#define EXT2\_Header  
  
#include **<stdio.h>**  
#include **<string.h>**  
#include **<stdlib.h>**  
#include **<unistd.h>**  
#include**<time.h>**  
#include**<sys/ioctl.h>**  
#include**<termios.h>**  
*//块大小512字节，一组数据块或索引节点个数4096个*  
*//每组数据容量4096x512B=2MB*  
#define PATH "FS.txt"   
#define \_LEN 32  
  
**typedef** unsigned short u16;  
**typedef** short s16;  
**typedef** unsigned int u32;  
**typedef** unsigned long long u64;  
**typedef** unsigned char u8;  
**typedef** **struct** ext2\_group\_desc *//组描述符(占1个块，包括superblock的内容)*  
{  
 char bg\_volume\_name[16]; *//卷名*  
 u16 bg\_block\_bitmap; *//保存块位图的块号*  
 u16 bg\_inode\_bitmap; *//保存索引结点位图的块号*  
 u16 bg\_inode\_table; *//索引结点表的起始块号*  
 u16 bg\_free\_blocks\_count; *//本组空闲块的个数*  
 u16 bg\_free\_inodes\_count; *//本组空闲索引结点的个数*  
 u16 bg\_used\_dirs\_count; *//本组目录的个数*  
 char bg\_pad[4]; *//填充(0xff)*  
 char psw[20];  
}ext2\_group\_desc;  
*//不算密码合计32个字节，由于只有一个组，且占用一个块，故需要填充剩下的512-32=480字节*  
  
**typedef** **struct** ext2\_inode { *//索引节点*  
 u16 i\_mode; *//文件类型及访问权限*  
 u16 i\_blocks; *//文件的数据块个数*  
 u32 i\_size; *//大小(字节)*  
 u64 i\_atime; *//访问时间*  
 u64 i\_ctime; *//创建时间*  
 u64 i\_mtime; *//修改时间*  
 u64 i\_dtime; *//删除时间*  
 u16\* i\_block[8]; *//指向数据块的指针*  
 char i\_pad[8]; *//填充1(0xff)*  
}ext2\_inode;  
 *//每个索引结点的长度为64字节*  
  
**typedef** **struct** ext2\_dir\_entry {   
 u16 inode; *//索引节点号*  
 u16 rec\_len; *//目录项长度*  
 u8 name\_len; *//文件名长度*  
 u8 file\_type; *//文件类型(1:普通文件，2:目录…)*  
 char name[8\*\_LEN]; *//文件名*  
}ext2\_dir\_entry;  
  
  
ext2\_inode inode;  
ext2\_dir\_entry dir\_entry;  
FILE\* f; *//文件指针*  
  
*//常驻内存数据结构*  
unsigned short fopen\_table[16] ; */\*文件打开表，最多可以同时打开16个文件\*/*  
unsigned short last\_alloc\_inode; */\*上次分配的索引结点号\*/*  
unsigned short last\_alloc\_block; */\*上次分配的数据块号\*/*  
unsigned short current\_dir ; */\*当前目录(索引结点）\*/*  
char current\_path[256]; */\*当前路径(字符串) \*/*  
ext2\_group\_desc group\_desc; */\*组描述符\*/*   
#endif

### 3.4.3 主要函数及设计思路

所有底层函数和命令层函数

*//初始化文件系统*  
void initialize\_disk(); */\*建立文件系统\*/*  
void initialize\_memory(); */\*初始化文件系统的内存数据\*/*   
  
*//底层函数*  
void update\_group\_desc(**struct** ext2\_group\_desc \*group\_desc, FILE \*fs); */\*将内存中的组描述符更新到"硬盘"\*/*  
void reload\_group\_desc(**struct** ext2\_group\_desc \*group\_desc, FILE \*fs); */\*载入可能已更新的组描述符\*/*   
void load\_inode\_entry(unsigned short inode, **struct** ext2\_inode \*inode\_entry, FILE \*fs); */\*载入特定的索引结点\*/*  
void update\_inode\_entry(unsigned short inode, **struct** ext2\_inode \*inode\_entry, FILE \*fs); */\*更新特定的索引结点\*/*  
void load\_block\_entry(unsigned short block, void \*block\_entry, FILE \*fs); */\*载入特定的数据块\*/*  
void update\_block\_entry(unsigned short block, void \*block\_entry, FILE \*fs); */\*更新特定的数据块\*/*  
void update\_inode\_i\_block(**struct** ext2\_inode \*inode\_entry, unsigned short \*blocks, int num\_blocks, FILE \*fs); */\*根据多级索引机制更新索引结点的数据块信息域\*/*  
int ext2\_new\_inode(); */\*分配一个新的索引结点\*/*  
int ext2\_alloc\_block(); */\*分配一个新的数据块\*/*  
void ext2\_free\_inode(int inode); */\*释放特定的索引结点\*/*  
void ext2\_free\_block\_bitmap(int block); */\*释放特定块号的数据块位图\*/*  
void ext2\_free\_blocks(**struct** ext2\_inode \*inode); */\*释放特定文件的所有数据块\*/*  
int search\_filename(const char \*filename); */\*在当前目录中查找文件\*/*  
int test\_fd(int fd); */\*检测文件打开ID(fd)是否有效\*/*  
  
*//命令层*  
void dir(); */\*无参数\*/*   
void mkdir(); */\*filename\*/*   
void rmdir(); */\*filename\*/*   
void create(); */\*filename\*/*   
void delete\_(); */\*filename\*/*   
void cd(); */\*filename\*/*   
void attrib(); */\*filename，rw\*/*   
void open(); */\*filename\*/*   
void close(); */\*fd\*/*  
void read(); */\*fd\*/*  
void write(); */\*fd，source\*/*   
  
  
*//用户接口层*  
void shell(); */\*启动用户接口\*/*  
void format(); */\*重新建立文件系统,无参数\*/*  
void quit(); */\*退出shell(),无参数\*/*

**部分函数设计思路**

initialize\_disk()

该函数负责建立文件系统。通过创建一个名为"FS.txt"的文件，并填充相应大小的空白数据块，实现了文件系统的初始化。

initialize\_memory()

用于初始化文件系统的内存数据，包括文件打开表和其他一些跟踪文件系统状态的变量。未提供具体初始化细节。

update\_group\_desc(struct ext2\_group\_desc \*group\_desc, FILE \*fs)

将内存中的组描述符更新到"硬盘"。通过将组描述符写入文件系统的开头（假设为硬盘），实现了组描述符的更新。

reload\_group\_desc(struct ext2\_group\_desc \*group\_desc, FILE \*fs)

从"硬盘"读取可能已更新的组描述符到内存。通过从文件系统的开头读取组描述符，实现了组描述符的重新加载。

load\_inode\_entry(unsigned short inode, struct ext2\_inode \*inode\_entry, FILE \*fs)

从"硬盘"读取特定的索引结点到内存。根据给定的inode号，计算在文件系统中的偏移量，然后将相应的索引节点数据读入内存。

update\_inode\_entry(unsigned short inode, struct ext2\_inode \*inode\_entry, FILE \*fs)

将内存中的特定索引结点写入到"硬盘"。与load\_inode\_entry 相反，将给定的索引节点数据写回到文件系统。

load\_block\_entry(unsigned short block, void \*block\_entry, FILE \*fs)

从"硬盘"读取特定的数据块到内存。根据给定的块号，计算在文件系统中的偏移量，然后将相应的数据块数据读入内存。

update\_block\_entry(unsigned short block, void \*block\_entry, FILE \*fs)

将内存中的特定数据块写入到"硬盘"。与 load\_block\_entry 相反，将给定的数据块数据写回到文件系统。

update\_inode\_i\_block(struct ext2\_inode \*inode\_entry, unsigned short \*blocks, int num\_blocks, FILE \*fs)

根据多级索引机制更新索引结点的数据块信息域。根据给定的索引节点和数据块数组，更新索引节点的i\_block数组，实现了多级索引的更新。

ext2\_new\_inode()

分配一个新的索引结点。遍历索引节点位图，找到第一个未分配的索引节点，标记为已分配，并返回索引节点号。

ext2\_alloc\_block()

分配一个新的数据块。遍历数据块位图，找到第一个未分配的数据块，标记为已分配，并返回块号。

ext2\_free\_inode(int inode)

释放特定的索引结点。根据给定的inode号，更新索引节点位图，将相应的位标记为未分配。

ext2\_free\_block\_bitmap(int block)

释放特定块号的数据块位图。根据给定的块号，更新数据块位图，将相应的位标记为未分配。

ext2\_free\_blocks(struct ext2\_inode \*inode)

释放特定文件的所有数据块。遍历给定索引节点的数据块数组，释放对应的数据块。

search\_filename(const char \*filename)

在当前目录中查找文件。遍历当前目录，比较文件名，如果找到则返回文件的inode号。

test\_fd(int fd)

检测文件打开ID(fd)是否有效。根据文件描述符在文件打开表中的状态，返回相应的值：0表示无效，1表示有效，2表示文件描述符被占用。

dir()

列出当前目录的内容。遍历当前目录的数据块，输出文件和目录的名称。

mkdir(const char \*dirname)

创建一个新的目录。分配一个新的索引节点，设置相应的属性，并在当前目录添加一个新的目录项。

rmdir(const char \*dirname)

删除一个目录。释放目录对应的所有数据块和索引节点，然后在父目录中删除目录项。

create(const char \*filename)

创建一个新的文件。分配一个新的索引节点，设置相应的属性，并在当前目录添加一个新的文件项。

delete\_(const char \*filename)

删除一个文件。释放文件对应的所有数据块和索引节点，然后在父目录中删除文件项。

cd(const char \*dirname)

切换当前目录。根据目录名查找对应的目录项，如果找到，则将当前目录切换到该目录。

attrib(const char \*filename, const char \*rw)

更改文件属性。根据文件名和权限字符串（"r"或"w"）找到对应的文件，更新文件的权限属性。

open(const char \*filename)

打开一个文件。根据文件名查找对应的文件，找到一个空闲的文件描述符，将文件描述符与文件关联。

close(int fd)

关闭一个文件。根据文件描述符，将对应的文件描述符置为0，表示文件关闭。

read(int fd)

读取一个文件。根据文件描述符读取文件内容并输出。

write(int fd, const char \*source)

写入一个文件。根据文件描述符和源数据，将数据写入文件对应的数据块。

## 实验总结

通过对Ext2文件系统的模拟实验，我深入了解了文件系统的底层结构和基本操作。以下是我对实验的总结：

文件系统组成：

在Ext2文件系统中，组描述符、索引节点、数据块等是构成文件系统的关键组成部分。组描述符用于存储文件系统的整体信息，索引节点用于描述文件和目录的属性，数据块用于存储文件的实际数据。

底层函数的作用：

实验中的底层函数主要用于文件系统的基本操作，如更新组描述符、载入索引节点、载入数据块等。这些函数实现了文件系统数据的读写和更新。

索引节点和数据块管理：

通过ext2\_new\_inode和ext2\_alloc\_block函数，成功实现了新索引节点和新数据块的分配。释放索引节点和数据块的函数也确保了文件系统的空间高效利用。

文件系统操作命令层：

在命令层，实现了一系列文件系统的操作函数，如创建目录、删除文件、更改文件属性等。这些函数封装了底层的操作，提供了更高层次的文件系统操作接口。

用户接口层：

shell函数提供了用户友好的命令行接口，允许用户执行各种文件系统操作。这样的用户接口层使得文件系统的使用更加方便和直观。

### 3.5.1实验中的问题与解决过程

问题：文件系统初始化出现错误。

解决过程： 在初始化文件系统时，发现文件无法创建，导致后续的文件系统操作无法正常进行。通过检查代码，发现是对文件的写入操作出现问题。经过调试，发现是文件未成功打开导致的。解决方法是确保文件能够成功打开后再进行初始化。

问题：索引节点和数据块的分配和释放未能正确实现。

解决过程： 在实现索引节点和数据块的分配和释放时，出现了无法正确定位和标记空闲块的问题。通过添加调试输出，我发现是位图操作时的偏移量计算错误。调整计算逻辑后，成功解决了分配和释放问题。

问题：文件系统操作命令层实现不完善。

解决过程： 在实现文件系统操作命令时，部分功能的具体实现需要更多考虑。例如，对于目录的创建和删除，需要更新目录项，但代码中存在错误。通过仔细审查代码和参考文件系统设计规范，逐步修复了这些问题。

问题：文件读写操作存在潜在错误。

解决过程： 在实现文件读写操作时，发现部分操作可能导致内存越界或文件数据错误。通过对每个读写操作的参数进行严格的边界检查和错误处理，确保了文件读写的安全性。

问题：未完全实现的功能。

解决过程： 实验中由于时间有限，未能完全实现所有功能。例如，一些特殊文件属性的处理和异常情况的处理未进行详细考虑。解决方法是记录下这些未完成的部分，并在以后的学习和实践中逐步完善和改进。

### 3.5.2 实验收获

## 3.6 Q&A

*3.6.1.如何实现将文件作为硬盘来使用？*

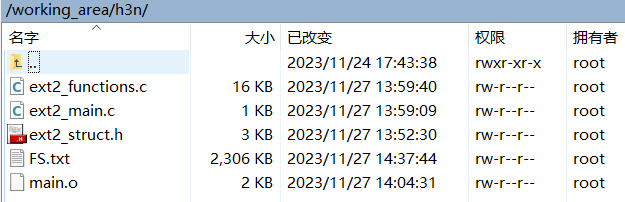
在openeuler的一个目录下创建一个固定大小(2.25MB)的FS.txt文件，用来模拟磁盘，命令如下

dd if=/dev/zero of=FS.txt bs=1 count=2360832

在这个命令中：

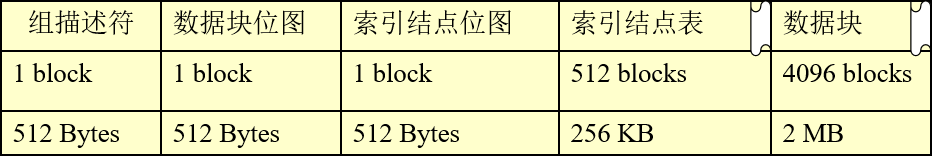
* if=/dev/zero表示输入文件是/dev/zero，这是一个特殊的文件，读取它会得到null字节（\0）。
* of=FS.txt表示输出文件是FS.txt。
* bs=1表示一次读写的字节数是1。
* count=2360832表示需要读写的次数是2360832。

这个命令会创建一个大小为2,360,832字节，内容全为null字节的文件。



也可以通过代码创建文件，直接写入2360832字节，并初始化所有块为0

*3.6.2.对于删除文件后的空闲区应该怎样处理？*



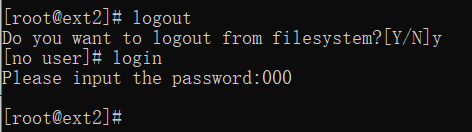
1. **更新位图**：你需要更新你的inode位图和块位图，将被删除文件所使用的inode和数据块标记为可用。
2. **更新索引节点**：你需要更新被删除文件的父目录的索引节点，移除指向被删除文件的目录项。
3. **释放数据块**：你需要释放被删除文件所使用的所有数据块。这通常意味着将这些数据块添加到你的文件系统的空闲块列表中。（多级索引依次删除）
4. **更新元数据**：你可能需要更新一些元数据，比如可用的inode数量和可用的数据块数量。

*3.6.3.多级索引如何实现？*

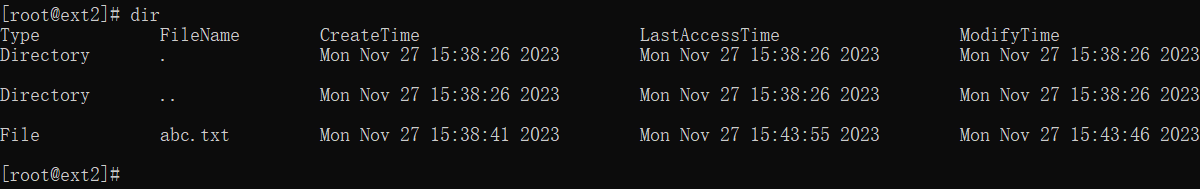
根据数据大小判定需要占用块数，根据大小确定需要采用0，1，2级索引。

## 3.7 运行结果

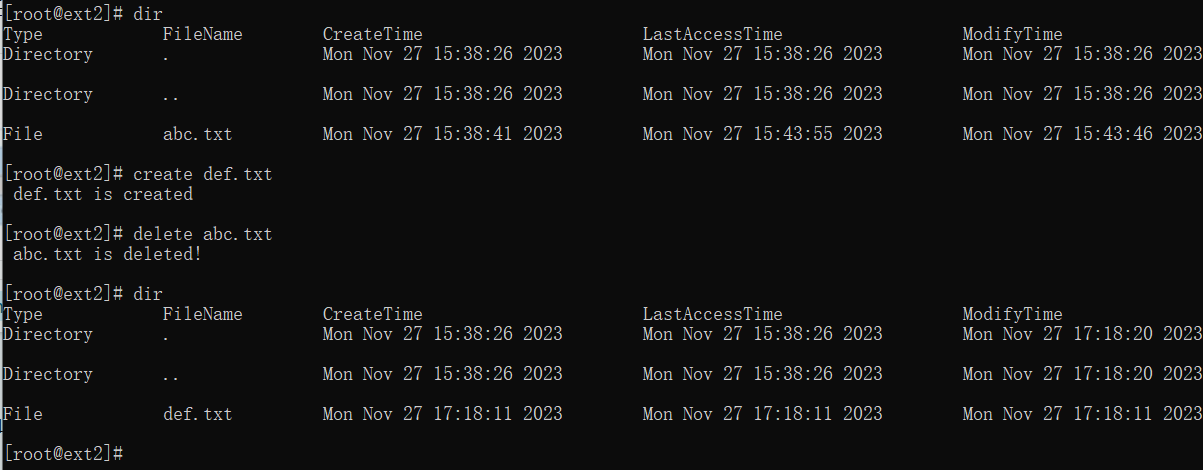
login/logout



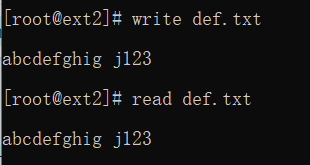
dir(ls)



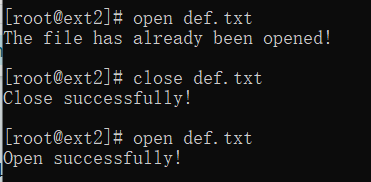
create/delete



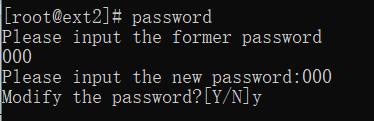
write/read



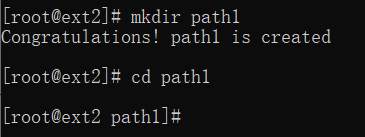
open/close



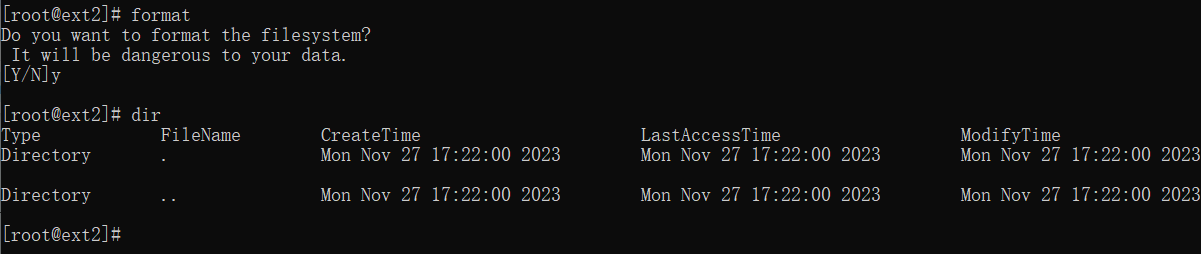
password



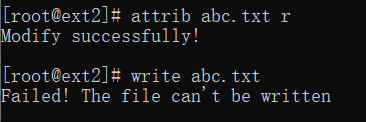
mkdir/cd



format

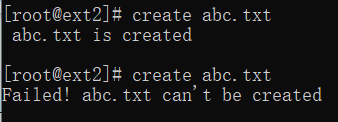


读写保护



5.2 可能出现的问题汇总

文件重名



多级索引

见源代码

目录项读取

通过读取目录文件的inode的数据块部分，按照流程图的方式读取目录项。（最大长度的方式）

数据块溢出问题

通过预先判断需要占用数据块个数和空闲块个数，避免分配过程中出现数据块溢出。

目录项变长的处理

用指针而不是直接存入内容。

## 3.8附件

### 3.8.1 附件1 程序



### 3.8.2 附件2 ReadMe

