**中国矿业大学计算机学院**

**2022级本科生课程报告**

课程名称 操作系统课程设计

报告时间 2024年5月4日

学生姓名 杨晓琦

学 号 08222213

专 业 计算机科学与技术

任课教师 王虎

# 一 系统调用实验

## 1.实验目的

深入了解Linux系统调用的执行过程，建立对系统调用的深入认识。

学会增加系统调用及添加内核函数的方法。

## 2.实验内容

在 Linux 0.11内核中添加新的系统调用,测试,并进行调试。

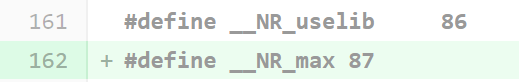
## 3.实验步骤

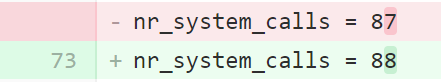
**3.1准备实验**

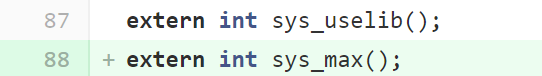
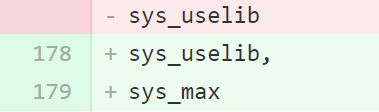
使用浏览器登录平台领取本次实验对应的任务：http://202.119.201.215/cumt-cs/2023/cUduruJS/mission1096.git从而在平台上创建个人项目（Linux 0.11 内核项目），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

**3.2在Linux 0.11内核中添加新的系统调用**

为Linux 0.11添加一个新的系统调用max函数，该函数实现比较两个参数的大小并将较大值返回的功能。

(1)为新系统调用分配一个唯一的系统调用号，以原来的最大系统调用号为基础加1作为新的系统调用号。在include/unistd.h文162行添加新的系统调用号\_\_NR\_max，如图4-1：

(2)添加新系统调用号的同时，也要使系统调用总数在原来的基础上增加1。修改kernel/system\_call.s文件第73行定义的系统调用总数，如下图：

(3)在include/linux/sys.h文件中的第88行使用C语言声明内核函数的原型，如图4-3所示。

(4)在kernel/sys.c文件的最后编写代码，实现新系统调用对应的内核函数，如图4-5：

3.2.5生成Linux 0.11内核，修改语法错误直到生成成功。

**3.3在Linux 0.11应用程序中测试新的系统调用**

(1)按F5启动调试。

(2)待Linux 0.11启动后，使用vi编辑器新建一个main.c文件。编辑main.c文件中的源代码(如下):

#define \_\_LIBRARY\_\_

#include<unistd.h>

#define \_\_NR\_max 87

\_syscall2( int, max, int, max1, int, max2 )

int main()

{

int i=max( 100, 200 );

printf( "the max of %d and %d is %d\n", 100, 200, i );

return 0;

}

(3)使用命令gcc main.c -o main生成可执行文件main。

(4)执行sync命令，将文件保存到磁盘。

(5)执行 chmod +x main 命令为main文件添加可执行权限。

(6)执行./main命令运行main。

(7)查看\_syscall2宏展开后得到的max函数。

依次执行下面的命令

gcc -E main.c -o main.i  
sync  
mcopymain.i b:main.i

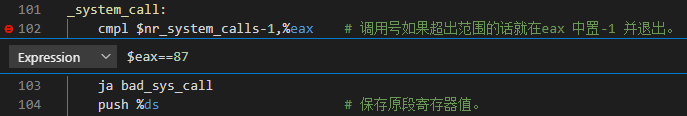
结束调试后，使用软盘编辑器工具打开floppyb.img文件。将其中的main.i文件复制到Windows的本地文件夹中。

将main.i文件拖动到VSCode中释放，在文件的最后部分就可以看到\_syscall2宏展开后得到的max函数。

**3.4调试系统调用执行的过程**

(1)结束之前的调试过程。

(2)在kernel/system\_call.s文件标号\_system\_call后的第一行汇编代码处（第102行）添加一个断点。

(3)在刚刚添加的断点上点击鼠标右键，在弹出的菜单中选择“Edit Breakpoint”，会在编辑器中显示出用于输入条件表达式的编辑框。在编辑框中设置断点条件 $eax==87

(4)按F5启动调试。

(5)待Linxu 0.11启动后输入命令./main运行应用程序main，会命中刚刚添加的条件断点。

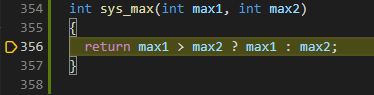
(6)选择“View”菜单中的“Run”，打开左侧的 “运行与调试”窗口。

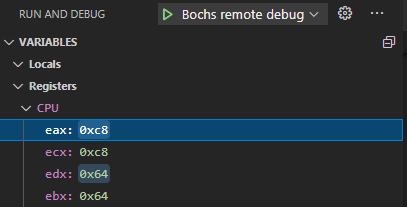
(7)在“运行与调试”窗口展开CPU寄存器，会发现EAX寄存器中的值为0x57（十进制为87），和max系统调用的调用号一致，说明应用程序正在调用max系统调用函数。查看EBX和ECX寄存器的值，存放的分别是max函数的参数0x64（十进制为100）和0xc8（十进制为200）。

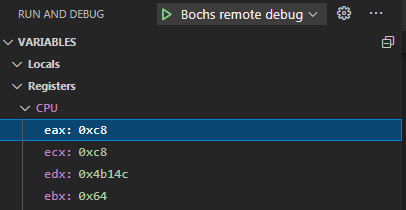
(8)按F10单步调试，直到黄色箭头指向第110行。其中第103行进行错误检查，第104-106行将各个段寄存器的值压入栈进行现场保护，第107-109行将保存在EBX、ECX和EDX寄存器中的参数压入栈.

(9)按F10继续单步调试，直到黄色箭头指向第119行。该行代码使用EAX寄存器存放的系统调用号作为系统调用函数指针表的下标，通过call指令调用对应的内核函数。

(10)按F11调试进入kernel/sys.c文件中的系统调用对应的内核函数，如下图：

****

(11)按F10单步调试，直到从内核函数返回到kernel/system\_call.s文件中的第120行。从内核函数返回后，返回值200会存放在EAX寄存器中，如下图：

(12)按F10单步调试，直到黄色箭头指向第168行。其中， 120-121行会将EAX存放的返回值（200）入栈，并把当前任务数据结构地址存入EAX寄存器，122-160行会进行进程调度以及信号的处理工作，161行-167行，恢复现场，恢复通用寄存器以及段寄存器，与保护现场时的顺序相反。此时查看监视窗口中的EAX，EBX和ECX寄存器的值，分别恢复为200（返回值），100（参数一），200（参数二），如下图：

(13)按F5继续运行，第168行的iret指令从0x80中断返回到main程序中。此时打开Bochs虚拟机的Display窗口，可以看到main程序已经执行完毕了。

(14)结束调试。

**3.5提交作业**

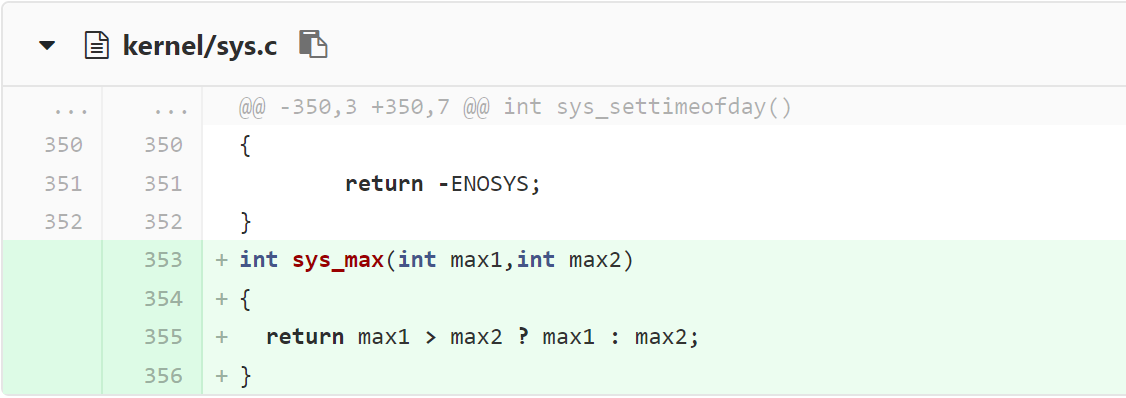
实验结束后先使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交到平台的个人项目中。

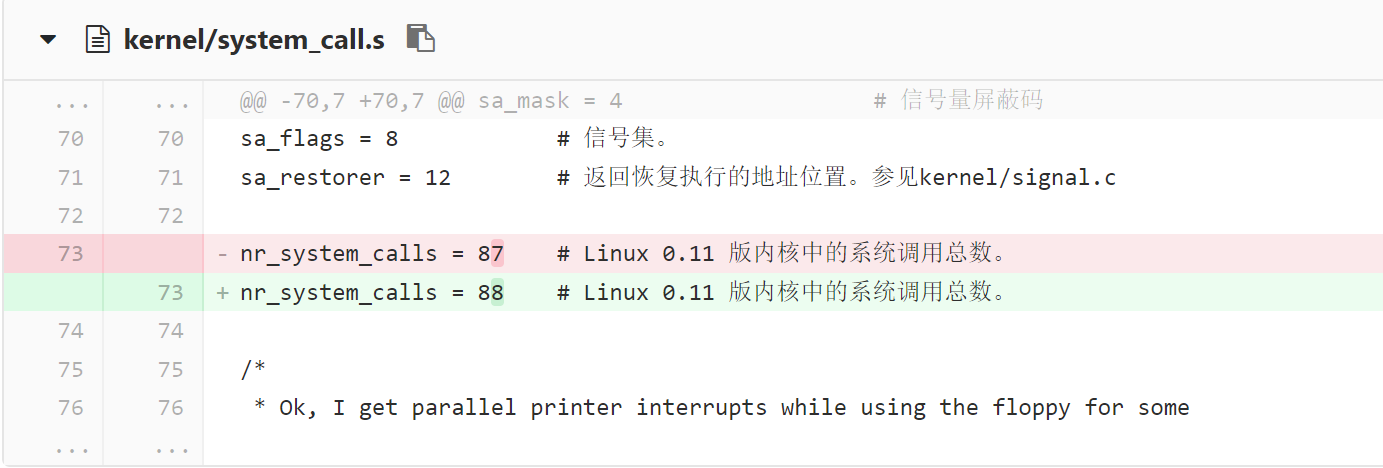
## 4.运行结果

 共修改4个文件包含9行增加和2行删除

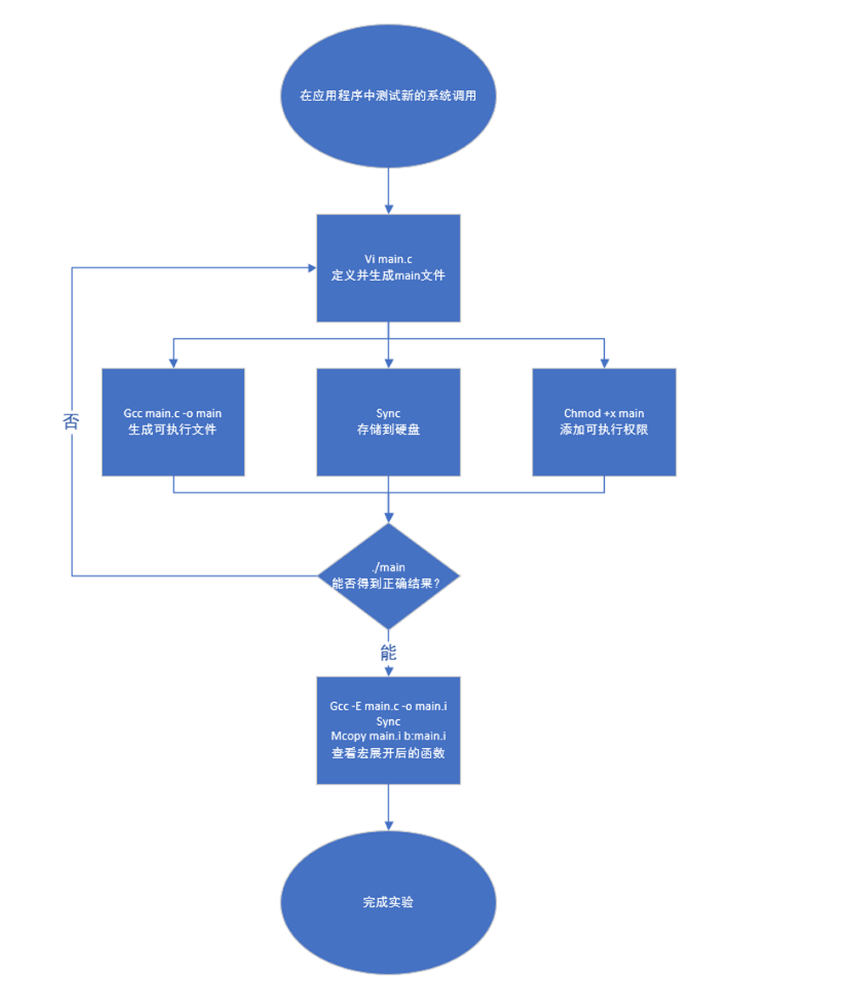


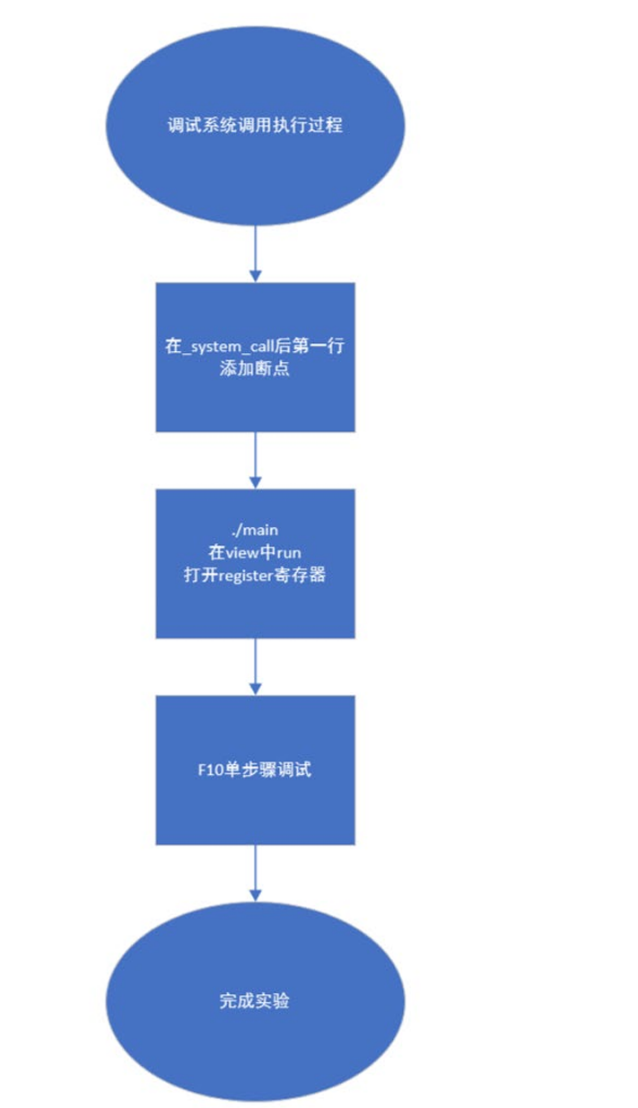






## 5.流程图





## 6.实验体会

Linux 系统语句中vi编辑器是最基本的文本编辑工具，在前面的实验中给了vi编 辑器的使用方法，但通过网上资料搜索才知道基本用法，比如按i进入插入模式，按 esc 退出命令模式，输入:进入编辑模式。在其中可以使用C语言来进行编辑（Linux由 C 语言组成）。系统调用是一个整体性的过程，\_syscall中的int 0x80指令是系统调用的 总入口，系统调用号\_\_NR\_max放在EAX寄存器中，通过此找到sys\_max()函数调用， 最后执行内核函数并通过EAX寄存器返回结果0xc8。

## 7.思考与练习

参考《Linux内核完全注释》第8.5.3.3节的内容，编写一个汇编程序，直接使用系统调用。

section .data

message db 'Hello, World!', 0xa ; 文本消息以null结尾

section .text

global \_start

\_start:

; 调用write系统调用

mov rax, 1 ; syscall号 1 代表write系统调用

mov rdi, 1 ; 文件描述符 1 代表标准输出

mov rsi, message ; 消息指针

mov rdx, 14 ; 消息长度

syscall ; 调用系统调用

; 调用exit系统调用

mov rax, 60 ; syscall号 60 代表exit系统调用

xor rdi, rdi ; 返回码为0

syscall ; 调用系统调用

在Linux 0.11内核中添加两个系统调用函数Iam和Whoami

函数原型如下：

int Iam(const char\* name)；将字符串name的内容保存到内核中，返回值是拷贝的字符数，如果name长度大于32，则返回-1，并置errno为EINVAL。

int Whoami(char\* name, int size)；将Iam保存到内核中的字符串拷贝到数据缓冲区name中，size为数据缓冲区name的长度，返回值是拷贝的字符数。如果size小于所需空间，则返回-1，并置errno为EINVAL。编写两个应用程序。其中，Iam应用程序调用Iam函数，并使用命令行中的第一个参数作为Iam函数的参数；Whoami应用程序调用Whoami函数，并打印输出获取的字符串。

Iam应用程序

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#define MAX\_NAME\_LENGTH 32

extern int Iam(const char\* name);

int main(int argc, char \*argv[]) {

if (argc != 2) {

fprintf(stderr, "Usage: %s <name>\n", argv[0]);

exit(EXIT\_FAILURE);

}

const char \*name = argv[1];

int ret = Iam(name);

if (ret == -1) {

if (errno == EINVAL) {

fprintf(stderr, "Name length exceeds maximum allowed length of %d\n", MAX\_NAME\_LENGTH);

} else {

perror("Iam");

}

exit(EXIT\_FAILURE);

}

printf("Name \"%s\" saved to kernel space\n", name);

printf("Characters copied: %d\n", ret);

return 0;

}

Whoami应用程序

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#define BUFFER\_SIZE 256

extern int Whoami(char\* name, int size);

int main() {

char name[BUFFER\_SIZE];

int ret = Whoami(name, BUFFER\_SIZE);

if (ret == -1) {

if (errno == EINVAL) {

fprintf(stderr, "Buffer size is too small to hold the name\n");

} else {

perror("Whoami");

}

exit(EXIT\_FAILURE);

}

printf("Name retrieved from kernel space: %s\n", name);

printf("Characters copied: %d\n", ret);

return 0;

}

# 二 进程创建实验一

## 1.实验目的

掌握创建子进程和加载执行新程序的方法，理解创建子进程和加载执行程序的不同。

调试跟踪fork和execve系统调用函数的执行过程。

## 2.实验内容

在Linux 0.11应用程序中调用fork函数创建子进程

## 3.实验步骤

**3.1准备实验**

使用浏览器登录平台领取本次实验对应的任务：http://202.119.201.215/cumt-cs/2023/cUduruJS/mission1097.git从而在平台上创建个人项目（Linux 0.11 内核项目），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

**3.2调用fork函数创建子进程**

(1)按F5启动调试。

(2)待Linux 0.11启动后，使用vi编辑器新建一个main.c文件，编写如下的代码。其中的getpid函数是一个系统调用函数，返回当前进程的进程号。

#define \_\_LIBRARY\_\_

#include <stdio.h>

#include <unistd.h>

int main(int argc, char \* argv[])

{

int pid;

printf("PID:%d parent process start.\n", getpid());

pid = fork();

if( pid != 0 )

{

 printf("PID:%d parent process continue.\n", getpid());

}

else

{

 printf("PID:%d child process start.\n", getpid());

 printf("PID:%d child process exit.\n", getpid());

 return 0;

}

 printf("PID:%d parent process exit.\n", getpid());

 return 0;

}

(3)使用命令gcc main.c -o app生成可执行文件app。

(4)执行 chmod +x app命令为app文件添加可执行权限。

(5)执行sync命令，将文件保存到硬盘。

(6)使用命令./app运行可执行文件app，分析运行结果。

(7) 使用vi编辑器修改main.c文件。

在printf(“PID:%d parent process continue\n”, getpid());语句前面添加一行语句：wait(NULL);重新编译、运行应用程序app，观察运行结果。

**3.3查看父进程与子进程的运行轨迹**

(1)为了方便观察父进程和子进程的运行轨迹，需要在进程结束的位置添加一个断点。在kernel/exit.c文件的第166行添加一个断点。

(2)按F5启动调试。在Linux操作系统启动完毕之前会多次命中刚刚添加的断点，每次命中断点后都按F5继续运行即可，直到Linux启动完毕。

(3)在Linux的终端输入命令./app后，父进程和子进程在结束时都会命中此断点，所以在第一次命中断点时，可以按F5继续运行，在第二次命中断点时，在VSCode的“View”菜单中选择“Command Palette...”，会在VSCode的顶部中间位置显示命令面板，输入“Lab: New Visualizer View”命令后，VSCode会在其右侧弹出一个窗口查看可视化视图。在右侧可视化视图顶部的编辑框中输入命令“#sched”后按回车，查看进程的运行轨迹。

**3.4调试跟踪fork函数的执行过程**

(1)在VSCode中删除所有断点，然后按F5启动调试。在Linux 0.11的终端输入下面的命令，查看可执行文件app的信息，将app文件的大小记录下来。

ls -l app

(2)结束调试，关闭Bochs虚拟机。

(3)使用VSCode打开kernel/system\_call.s文件，在第102行添加一个断点。

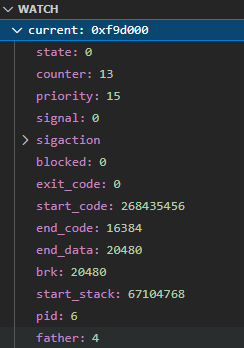
(4)在刚刚添加的断点上点击鼠标右键，在弹出的菜单中选择“Edit Breakpoint”，在编辑框中设置断点条件为下面的表达式后按回车确认：

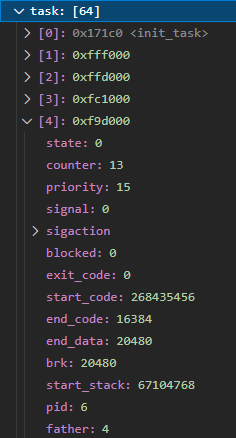
$eax==2 && current!=0 &&current->executable->i\_size==*文件大小*

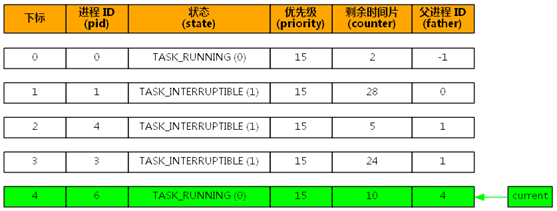
(5)按F5启动调试。

(6)在Linux 0.11的终端输入命令app，运行app应用程序，即可命中刚刚添加的条件断点。

(7) 在“WATCH”窗口添加last\_pid和current->pid，查看它们的值。

****(8)在“WATCH”窗口添加全局变量current并展开它的值，查看当前进程的信息。其中，“state=0”表示当前进程（即使用可执行文件app创建的进程）正处于运行态；“counter=13”表示其剩余时间片的大小；“priority=15”表示其优先级；“father=4”表示其父进程的进程号。如图5-1所示。

(9) 在“WATCH”窗口添加全局变量task并展开它的值，查看进程表中的所有进程的信息，如图5-2所示。

(10) 在VSCode的“View”菜单中选择“Command Palette...”，会在VSCode的顶部中间位置显示命令面板，输入“Lab: New Visualizer View”命令后，VSCode会在其右侧弹出一个窗口查看可视化视图。在右侧可视化视图顶部的编辑框中输入命令“#task”后按回车，就可以查看进程列表。如图5-3所示。

(11) 按F10单步调试至第119行，再按F11进入fork系统调用的内核函数。

(12)按F10单步调试至第272行。

(13)按F10单步调试至第279行，然后按F11进入copy\_process函数。

(14)按F10单步执行第98行的代码。

(15)按F10单步执行直到黄色箭头指向第103行。

(16)按F10单步执行第103行的代码，黄色箭头指向第104行。

(17)第104行设置子进程为“不可中断等待状态”；第105行设置子进程的进程号；第106行设置子进程的父进程号；第125行将子进程EAX寄存器的值设置为0。

(18)在第171行点击鼠标右键，在弹出的菜单中选择“Run to Cursor”。

(19)按F10单步调试，直到从copy\_process函数返回到kernel/system\_call.s文件中的第280行。copy\_process函数的返回值是子进程的进程号，会被放入EAX寄存器中，也就是父进程从fork函数返回时得到的返回值。

(20)按F10单步调试，直到从汇编函数返回到kernel/system\_call.s文件中的第120行。

(21)继续按F10单步调试，直到第133行。

(22)按F5继续调试，在Bochs的Display窗口中可以看到app可执行文件运行结束。

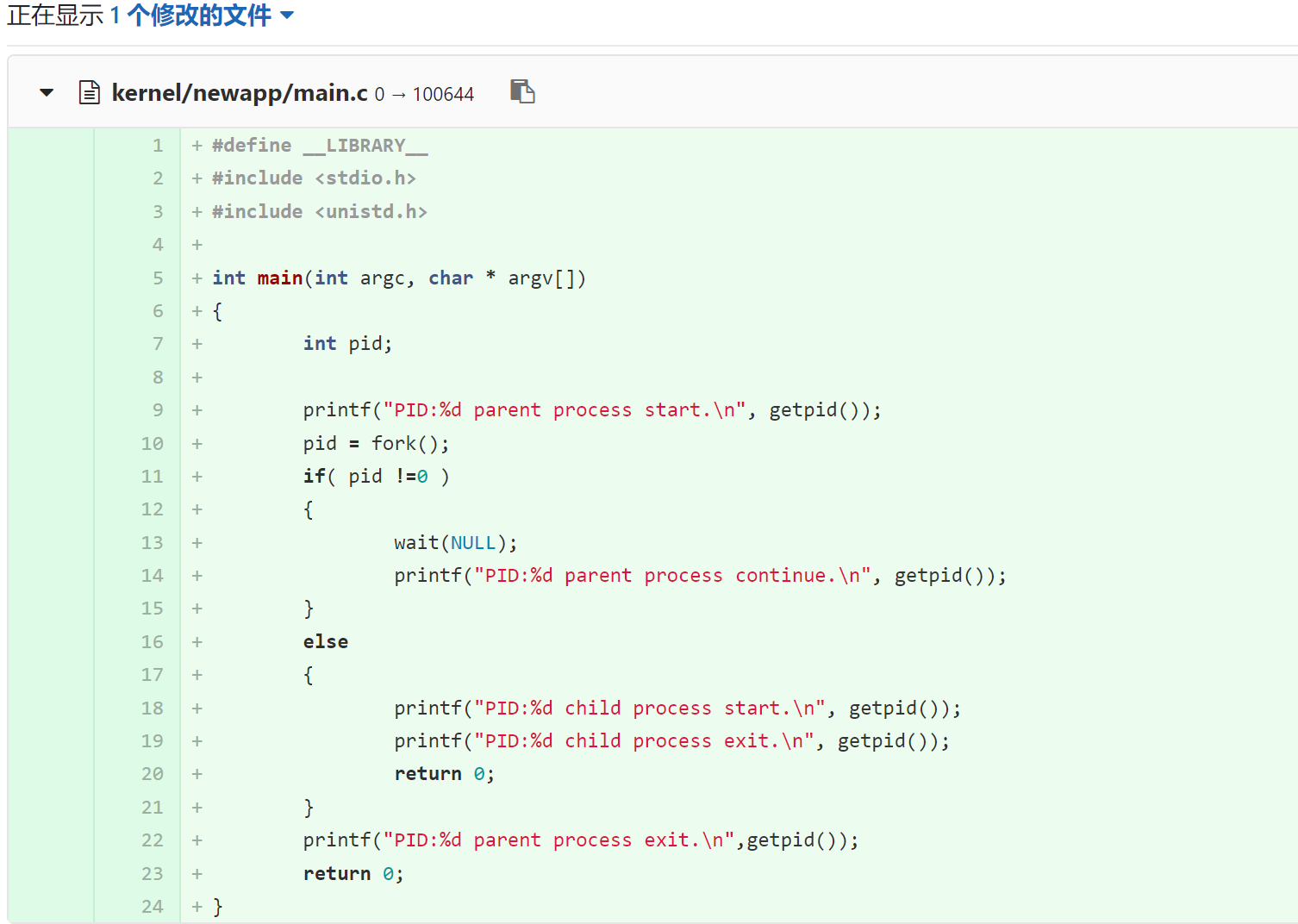
(23)结束调试，关闭Bochs虚拟机。

**3.5提交作业**

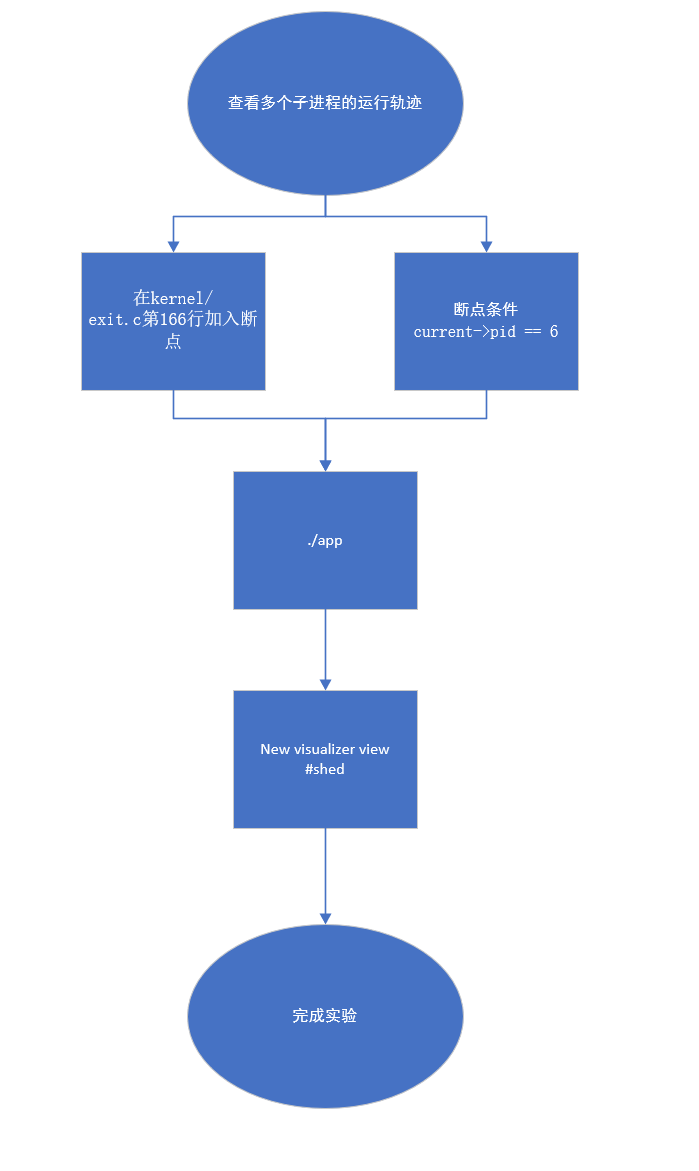
将Linux 0.11硬盘中的main.c文件通过软盘B复制到Linux 0.11内核项目的根目录中。然后使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交到平台的个人项目中。

## 4.运行结果

共修改1个文件

****

## 5.流程图



## 6.实验体会

在深入学习操作系统的进程管理时，理解进程的状态转换和调度机制至关重要。进程的状态，如就绪、运行、阻塞和终止，以及它们之间的转换，对进程的调度和执行产生直接影响。

通过实际调试进程在不同状态间的转变，我加深了对这些状态和转换机制的理解。在调试过程中，我熟练运用了调试工具，能够轻松查看进程的状态和堆栈信息，同时深入剖析了各种系统调用函数的执行流程。

此外，我还学会了利用跟踪进程运行轨迹和可视化窗口的方法，使进程的状态和调度过程变得更为直观和易于理解。这种形象化的学习方式不仅增强了我对进程管理和调度原理的掌握，也提升了我解决相关问题的能力。

## 7.思考与练习

无

# 三.进程创建实验二

## 1.实验目的

掌握创建子进程和加载执行新程序的方法，理解创建子进程和加载执行程序的不同。

调试跟踪fork和execve系统调用函数的执行过程。

## 2.实验内容

调用execve函数加载执行一个新程序

## 3.实验步骤

**3.1准备实验**

使用浏览器登录平台领取本次实验对应的任务：http://202.119.201.215/cumt-cs/2023/cUduruJS/mission1098.git从而在平台上创建个人项目（Linux 0.11 内核项目），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

**3.2调用execve函数加载执行一个新程序**

(1)按F5启动调试。

(2)使用vi编辑器新建一个new.c文件，编写如下的代码。

#define \_\_LIBRARY\_\_

#include <stdio.h>

#include <unistd.h>

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

{

printf("PID:%d newprocess.\n", getpid());

return 0;

}

(3)使用命令gcc new.c -o new生成可执行文件new。

(4)执行 chmod +x new命令为new文件添加可执行权限。

(5)执行sync命令，将文件保存到硬盘。

(6)使用命令./new运行可执行文件new，确保其可以正常运行。

(7)使用vi编辑器新建一个old.c文件，编写如下的代码。

#define \_\_LIBRARY\_\_

#include <stdio.h>

#include <unistd.h>

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

{

printf("PID:%d oldprocess start.\n", getpid());

execve("new", NULL, NULL );

printf("PID:%d oldprocess exit.\n", getpid());

return 0;

}

(8)使用命令gcc old.c -o old生成可执行文件old。

(9)执行 chmod +x old命令为old文件添加可执行权限。

(10)执行sync命令，将文件保存到硬盘。

(11)使用命令./old运行可执行文件old。

**3.3调试跟踪execve函数的执行过程**

(1)在Linux 0.11的终端输入命令ls -lold，查看可执行文件old的信息，将old文件的大小记录下。

(2)结束调试，关闭Bochs虚拟机。

(3)在kernel/system\_call.s文件第102行添加一个条件断点，条件为：  
$eax==11 &&current!=0 &&current->executable->i\_size==*文件大小*

(4)按F5启动调试

(5)在Linux 0.11的终端输入命令./old，运行old应用程序。

(6)按F10单步调试到第119行，按F11进入到execve系统调用对应的汇编函数sys\_execve，黄色箭头指向第260行。

(7)按F10单步调试到底262行，按F11进入到do\_execve函数中。

(8)在第314行点击鼠标右键，在弹出的菜单中选择“Run to Cursor”。

(9)在第472行点击鼠标右键，在弹出的菜单中选择“Run to Cursor”。

(10)在第494行点击鼠标右键，在弹出的菜单中选择“Run to Cursor”。

(11)按F10单步调试到第508行。

(12)在第515行点击鼠标右键，在弹出的菜单中选择“Run to Cursor”。

(13)按F10单步调试，do\_execve函数返回到sys\_execve函数。

(14)按F5继续调试，在Bochs的Display窗口中可以看到old可执行文件运行结束。

(15)结束调试，关闭Bochs虚拟机。

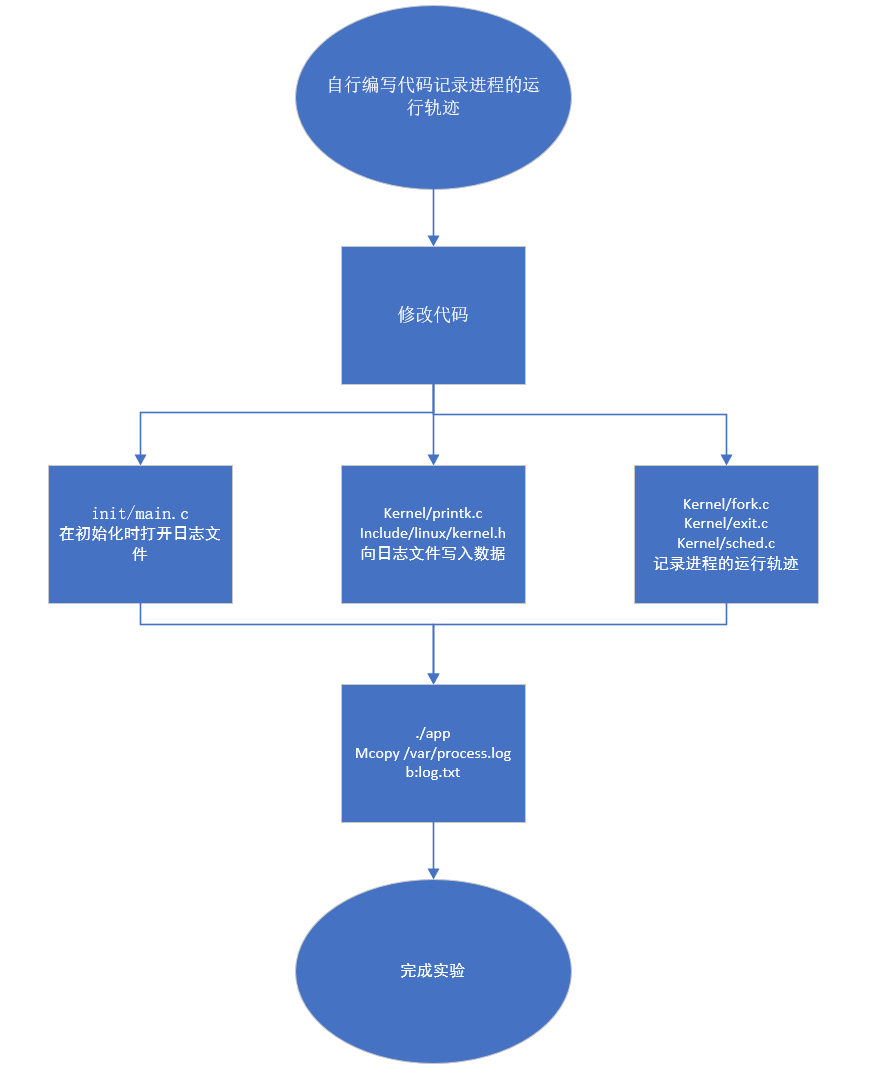
**3.4提交作业**

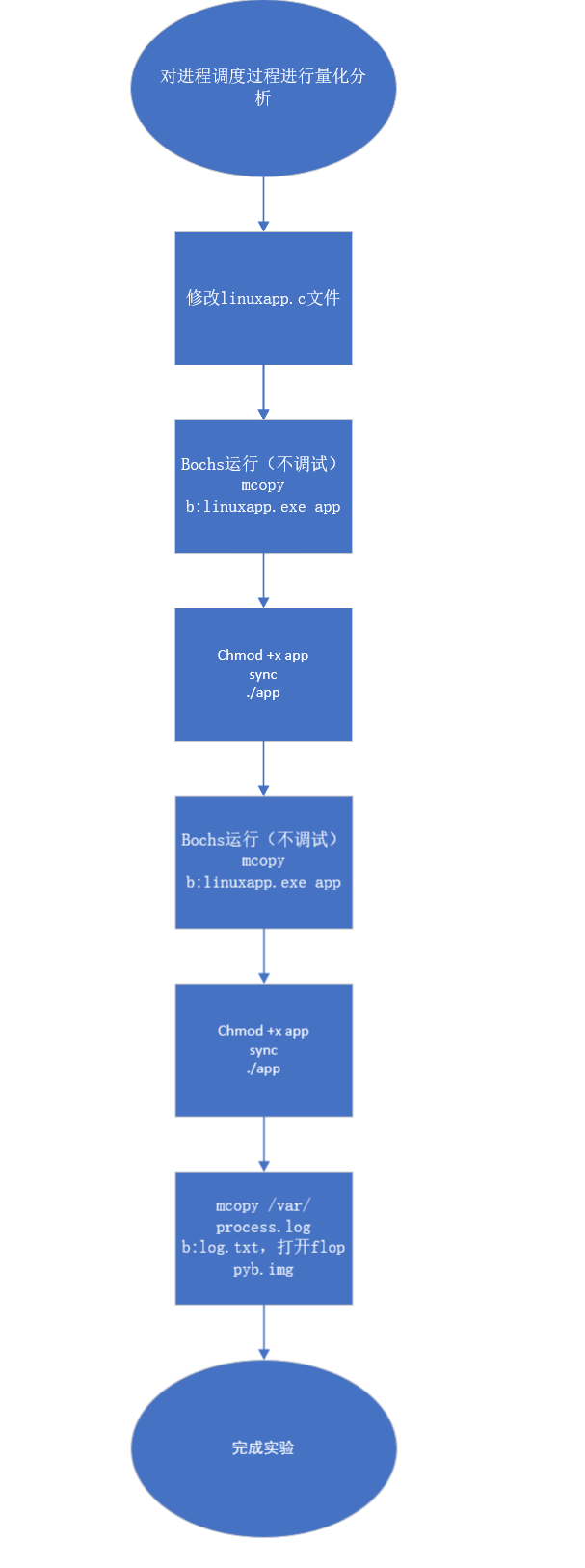
将Linux 0.11硬盘中的new.c文件和old.c文件通过软盘B复制到Linux 0.11内核项目的根目录中。然后使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交到平台的个人项目中。

## 4.运行结果

**共修改2个文件**

## 5.流程图





## 6.实验体会

在多进程编程的领域中，Linux系统提供了强大的系统调用和库函数集，使开发者能够轻松创建、管理和调度多个进程。然而，在学习和应用这些功能时，我们必须关注几个关键点以确保进程间的高效协作和系统的稳定性。

首先，进程间的通信（IPC）是多进程编程中的核心问题。Linux提供了多种IPC机制，如管道、消息队列、信号量、共享内存和套接字等。我们需要根据具体的应用场景选择合适的IPC方式，并确保数据在进程间的正确传递和同步。

其次，同步是另一个重要的考虑因素。在多进程环境中，多个进程可能同时访问共享资源，这可能导致数据竞争和不一致性问题。因此，我们需要使用适当的同步机制，如互斥锁、读写锁、条件变量等，来确保对共享资源的访问是安全的。

此外，进程的调度和优先级也是我们需要关注的问题。Linux内核负责进程的调度，它会根据进程的优先级、状态和其他因素来决定何时执行哪个进程。开发者可以通过设置进程的优先级来影响内核的调度决策，以确保重要进程获得足够的CPU资源。

在学习过程中，我们需要深入理解这些概念和机制，并通过实际编程实践来掌握它们。同时，我们还应该关注系统的性能和稳定性，避免过度使用进程导致系统资源耗尽或进程死锁等问题。

总之，多进程编程是Linux系统编程中的重要部分，我们需要掌握相关的系统调用和库函数，并关注进程间的通信、同步、调度和优先级等问题，以确保程序的正确性和高效性。

## 7.思考与练习

7.1模仿3.1中Linux 0.11应用程序的源代码，使用for语句编写一个循环，使父进程能够循环创建10个子进程，每个子进程在输出自己的pid后退出，父进程等待所有子进程结束后再退出。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#define NUM\_CHILDREN 10

int main() {

pid\_t pid;

for (int i = 0; i < NUM\_CHILDREN; ++i) {

pid = fork();

if (pid < 0) {

perror("fork");

exit(EXIT\_FAILURE);

} else if (pid == 0) { // 子进程

printf("Child %d PID: %d\n", i + 1, getpid());

exit(EXIT\_SUCCESS);

}

}

// 父进程等待所有子进程结束

for (int i = 0; i < NUM\_CHILDREN; ++i) {

int status;

wait(&status);

if (WIFEXITED(status)) {

printf("Child %d exited with status: %d\n", i + 1, WEXITSTATUS(status));

}

}

return 0;

}

7.2结合3.3中的内容编写一个Linux应用程序，在main函数中使用fork函数创建一个子进程，在子进程中使用execve函数加载执行另外一个程序的可执行文件，并且让父进程在子进程退出后再结束运行。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

perror("fork");

exit(EXIT\_FAILURE);

} else if (pid == 0) { // 子进程

char \*args[] = {"/path/to/your/program", NULL}; // 替换为要执行的程序的路径和参数

execve(args[0], args, NULL);

perror("execve"); // 如果execve失败，则输出错误信息

exit(EXIT\_FAILURE);

} else { // 父进程

int status;

waitpid(pid, &status, 0); // 等待子进程退出

if (WIFEXITED(status)) {

printf("Child process exited with status: %d\n", WEXITSTATUS(status));

} else {

printf("Child process exited abnormally\n");

}

}

# 四、五.进程的状态与进程调度实验

## 1.实验目的

调试进程在各种状态间的转换过程，熟悉进程的状态和转换。

通过对进程运行轨迹的跟踪来形象化进程的状态和调度。

掌握Linux下的多进程编程技术。

## 2.实验内容

编写程序，在Linux 0.11应用程序中调用fork函数创建多个子进程

在内核中编写程序，记录进程的运行轨迹

## 3.实验步骤

**3.1在Linux 0.11应用程序中调用fork函数创建多个子进程**

**(1) 编写一个可以创建多个子进程的Linux 0.11应用程序**

①使用VSCode打开之前克隆到本地的Linux 0.11应用程序项目。

②打开linuxapp.c文件main函数，让父进程新建三个子进程，并分别输出子进程id及其父进程id。代码如下：

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

{

if( 0 == fork() )

{

printf("child process pid=%d ppid=%d line=%d\n",

getpid(), getppid(), \_\_LINE\_\_);

}

else if( 0 == fork() )

{

printf("child process pid=%d ppid=%d line=%d\n",

getpid(), getppid(), \_\_LINE\_\_);

}

else if( 0 == fork() )

{

printf("child process pid=%d ppid=%d line=%d\n",

getpid(), getppid(), \_\_LINE\_\_);

}

else

{

wait( NULL );

printf("parent process pid=%d ppid=%d\n",getpid(), getppid());

}

return 0;

}

③将生成的可执行文件从软盘B拷贝到硬盘，命令为mcopy b:linuxapp.exe app

④为app文件添加可执行权限，命令为chmod +x app

⑤执行“sync”命令确保app文件写入硬盘。

⑥使用命令./app运行可执行文件app。

⑦结束调试，关闭Bochs虚拟机。

(2) 查看多个子进程的运行轨迹

①使用VSCode的“File”菜单中的“Open Folder”打开之前克隆到本地的Linux 0.11内核项目文件夹。

②在VSCode的“Terminal”菜单中选择“Run Build Task...”，会在VSCode的顶部中间位置弹出一个可以执行的Task列表，选择其中的“生成项目”。

③待Linux 0.11内核项目生成成功后，使用Windows资源管理器分别打开之前克隆到本地的Linux 0.11应用程序项目和Linux 0.11内核项目所在的文件夹。

④用Linux0.11应用程序项目文件夹中的硬盘镜像文件harddisk.img覆盖Linux0.11内核项目文件夹中的harddisk.img文件，这样就可以在Linux 0.11内核项目的硬盘中使用之前生成的app文件了。

⑤使用VSCode的“File”菜单中的“Open Folder”打开之前克隆到本地的Linux 0.11内核项目文件夹。

⑥为了方便观察app应用程序中的父进程和子进程的运行轨迹，需要在父进程结束的位置添加一个条件断点。请读者在kernel/exit.c文件的第166行（进程结束后触发进程调度的位置）添加一个条件断点，条件为“current->pid == 6”。

⑦按F5启动调试。输入命令./app后，会命中刚刚添加的条件断点。

⑧在VSCode的“View”菜单中选择“Command Palette...”，会在VSCode的顶部中间位置显示命令面板，输入“Lab: New Visualizer View”命令后，VSCode会在其右侧弹出一个窗口让读者查看可视化视图。在右侧可视化视图顶部的编辑框中输入命令“#sched”后按回车（需要等待较长时间完成刷新），就可以查看进程运行的轨迹了，如图6-2所示。

**3.2在Linux 0.11应用程序中调用fork函数创建多个子进程**

(1) **在系统初始化时打开日志文件process.log**

使用VSCode打开之前克隆到本地的Linux 0.11内核项目并定位到内核的入口函数，即init/main.c文件中的start函数。

(2) **编写fprintk函数用于向process.log文件写入数据**

将下面代码添加在kernel/printk.c文件的结尾处，并且在include/linux/kernel.h文件中添加该函数的声明。

#include<linux/sched.h>

#include<sys/stat.h>

static char logbuf[1024];

int fprintk( int fd, const char \* fmt, ... )

{

va\_listargs;

int i;

struct file \* file;

struct m\_inode \* inode;

va\_start (args, fmt);

i = vsprintf (logbuf, fmt, args);

va\_end (args);

if( fd<3 )

{

\_\_asm\_\_ ("push %%fs\n\t"

"push %%ds\n\t"

"pop %%fs\n\t"

"pushl %0\n\t"

"pushl $\_logbuf\n\t"

"pushl %1\n\t"

"call \_sys\_write\n\t"

"addl $8,%%esp\n\t"

"popl %0\n\t"

"pop %%fs"

::"r" (i),"r" (fd):"ax", "dx");

}

else

{

if(task[4]==0)

return 0;

if( !( file=task[1]->filp[fd] ) )

return 0;

inode=file->f\_inode;

\_\_asm\_\_ ("push %%fs\n\t"

"push %%ds\n\t"

"pop %%fs\n\t"

"pushl %0\n\t"

"pushl $\_logbuf\n\t"

"pushl %1\n\t"

"pushl %2\n\t"

"call \_file\_write\n\t"

"addl $12,%%esp\n\t"

"popl %0\n\t"

"pop %%fs"

::"r" (i), "r" (file), "r" (inode) );

}

return i;// 返回字符串长度

}

(3) **记录进程的运行轨迹**

①修改kernel/fork.c文件中的copy\_process函数，在第114行后面增加语句：p->start\_time = jiffies;  
fprintk( 3, "%ld\t%c\t%ld\n", p->pid, 'N', jiffies );  
记录进程在创建后进入了就绪状态。在第168行后面增加语句：  
fprintk( 3, "%ld\t%c\t%ld\n", p->pid, 'J', jiffies );

②修改kernel/exit.c文件中的do\_exit函数，在第159行将进程状态设置为TASK\_ZOMBIE的后面插入一条语句：  
current->state = TASK\_ZOMBIE;  
fprintk( 3, "%ld\t%c\t%ld\n", current->pid, 'E', jiffies );

③修改kernel/sched.c文件中的schedule函数，在第213行进程的后面添加一条语句：  
fprintk( 3, "%ld\t%c\t%ld\n", (\*p)->pid, 'J', jiffies );

④在第251行if判断语句中增加下面的语句：

fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'J', jiffies);

⑤在第262行后面增加下面的语句：

fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'J', jiffies);

⑥在第268行后面增加下面的语句：

fprintk(3, "%ld\t%c\t%ld\n", task[next]->pid, 'R', jiffies);

⑦在sys\_pause函数的第288行的后面添加的跟踪语句应该在if条件语句中：  
if(current->pid != 0)  
fprintk( 3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies );

⑧使用Windows资源管理器打开3.1中克隆到本地的Linux 0.11应用程序项目的文件夹，复制其中的harddisk.img硬盘镜像文件。

⑨使用Windows资源管理器打开3.2中克隆到本地的Linux 0.11内核项目的文件夹，将上一步复制的harddisk.img文件粘贴覆盖同名的文件。这样就将之前构建的应用程序app的可执行文件通过硬盘复制到Linux 0.11内核项目中了。

⑩使用VSCode打开Linux 0.11内核项目，按F5启动调试。

⑪在终端输入命令./app执行app可执行文件，然后在纸上记录下打印输出的父进程和子进程的id。

⑫使用mcopy /var/process.log b:log.txt命令将日志文件复制到软盘B。

结束此次调试，关闭Bochs虚拟机。

⑬使用软盘编辑器工具打开floppyb.img文件，将软盘B中的log.txt文件复制到Windows本地目录中。

⑭打开log.txt文件。

**(4)对进程调度过程进行量化分析**

①使用VSCode打开在3.1中克隆到本地的Linux 0.11应用程序。首先在linuxapp.c文件中的main函数的前面添加一个新函数cpuio\_bound代码为:

#include<sys/wait.h>

#include<linux/sched.h>

#include<time.h>

void cpuio\_bound( int last, int cpu\_time, int io\_time )

{

struct tmsstart\_time, current\_time;

clock\_tutime, stime;

int sleep\_time;

while( last>0 )

{

times( &start\_time );

do

{

times( &current\_time );

utime=current\_time.tms\_utime-start\_time.tms\_utime;

stime=current\_time.tms\_stime-start\_time.tms\_stime;

}while( ( ( utime+stime )/HZ )<cpu\_time );

last-=cpu\_time;

if( last<=0 )

break;

sleep\_time=0;

while( sleep\_time<io\_time )

{

sleep( 1 );

sleep\_time++;

}

last-=sleep\_time;

}

}

将main函数修改为如下的代码：

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

{

pid\_t p1, p2, p3, p4;

if( ( p1=fork() )==0 )

{ printf( "in child1\n" ); cpuio\_bound( 5, 2, 2 );}

else if( ( p2=fork() )==0 )

{ printf( "in child2\n" ); cpuio\_bound( 5, 4, 0 );}

else if( ( p3=fork() )==0 )

{ printf( "in child3\n" ); cpuio\_bound( 5, 0, 4 );}

else if( ( p4=fork() )==0 )

{ printf( "in child4\n" ); cpuio\_bound( 4, 2, 2 );}

else

{

printf( "========This is parent process=======\n" );

printf( "pid=%d\n", getpid() );

printf( "pid1=%d\n", p1 );

printf( "pid2=%d\n", p2 );

printf( "pid3=%d\n", p3 );

printf( "pid4=%d\n", p4 );

}

wait( NULL );

return 0;

}

②生成Linux0.11应用程序项目后，使用Task中的“Bochs 运行（不调试）”启动Bochs虚拟机。

③待Linux 0.11启动完成后，在终端输入mcopy b:linuxapp.exe app

命令将可执行文件从软盘B拷贝到硬盘

④为app文件添加可执行权限，命令为：chmod +x app

⑤执行“sync”命令将文件保存到硬盘。

⑥使用命令./app运行app应用程序，确保应用程序可以正常运行。

⑦关闭Bochs虚拟机。

⑧使用Linux0.11应用程序项目文件夹中的硬盘镜像文件harddisk.img覆盖Linux0.11内核项目文件夹中的harddisk.img文件。

⑨使用VSCode打开Linux 0.11内核项目，按F5启动调试。

⑩待Linux 0.11启动完成后，使用命令./app运行app应用程序，然后在纸上记录下打印输出的父进程和子进程的id，以便下面分析数据时使用。

⑪将process.log文件复制到软盘B，命令为：mcopy /var/process.log b:log.txt

⑫结束调试，关闭Bochs虚拟机。

⑬使用软盘编辑器工具打开floppyb.img文件，将软盘B中的log.txt文件复制到Windows本地目录中。

⑭打开log.txt文件。

**3.3提交作业**

使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交到平台的个人项目中。

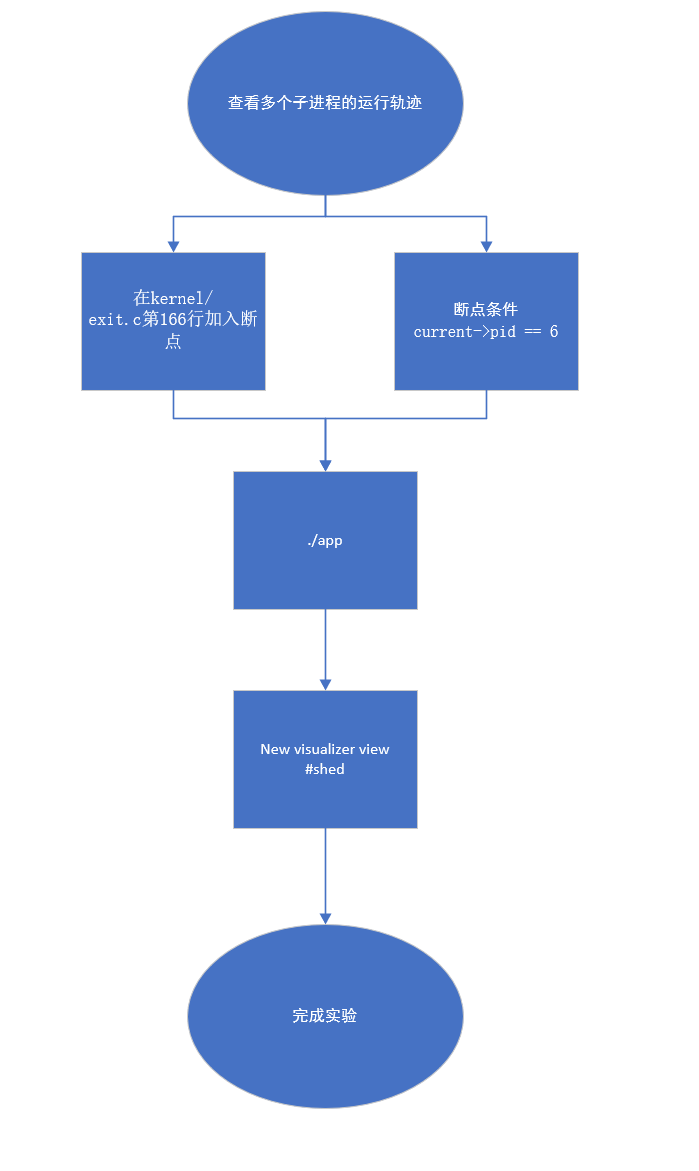
## 4.运行结果

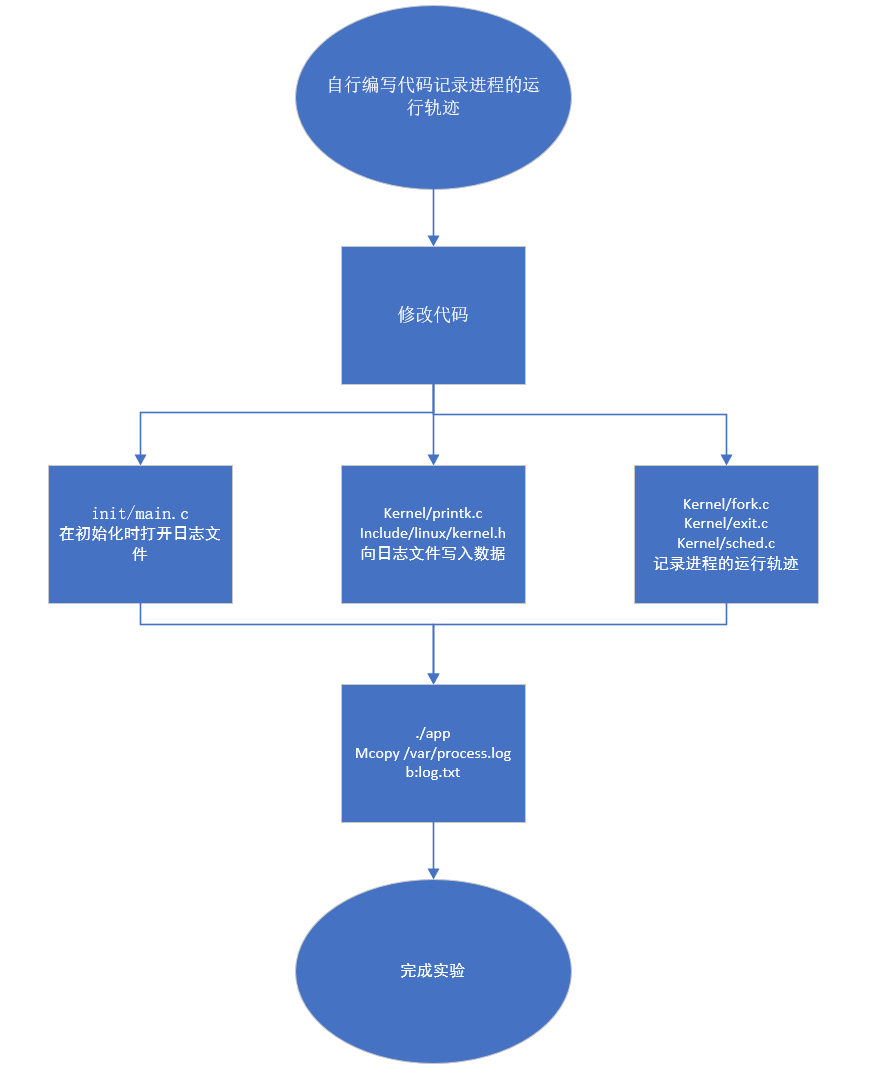
共修改1个文件

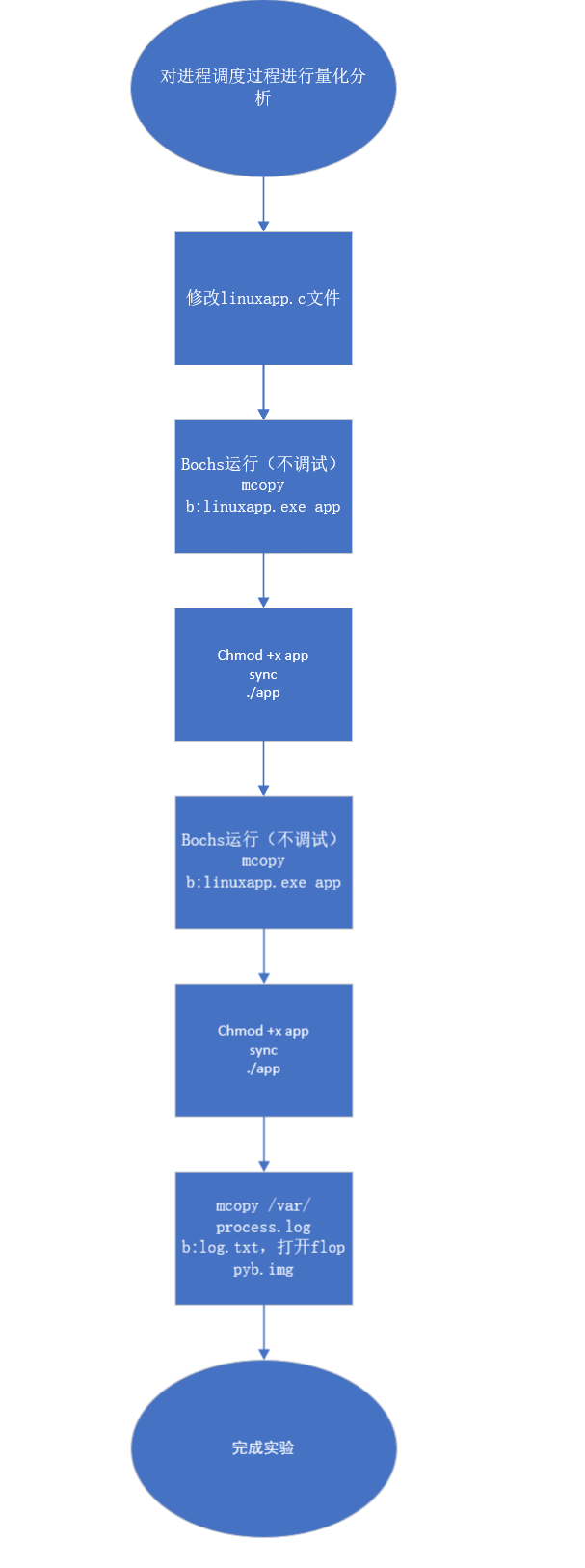




## 5.流程图







## 6.实验体会

在深入学习操作系统的进程管理时，对进程状态及其调度的理解是至关重要的。进程的状态主要包括就绪、运行、阻塞和终止等几种，它们之间的转换机制直接影响着系统的性能和效率。为了更深入地掌握这些概念，我进行了调试练习，通过调试工具细致地观察了进程在不同状态间的转换过程。

在调试过程中，我熟练掌握了使用调试工具来查看进程状态、堆栈信息以及系统调用函数的执行流程。这些工具不仅提供了详尽的数据，还让我能够直观地理解进程的运行轨迹。通过可视化窗口，我将复杂的进程状态和调度过程以图形化的方式展现出来，这使得我对进程管理和调度的原理有了更为直观和深入的理解。

在探索多进程编程时，我发现Linux系统提供了丰富的系统调用函数和库函数，它们为创建、管理和调度多个进程提供了极大的便利。然而，这并不意味着多进程编程就是一件简单的事情。在实际应用中，我们还需要注意进程间的通信和同步问题，以确保数据的一致性和系统的稳定性。此外，进程的调度策略和优先级设置也是影响系统性能的重要因素，需要我们根据具体的应用场景进行合理的配置和调整。

通过这一系列的学习和实践，我对操作系统的进程管理有了更为全面和深入的认识，也为我未来的学习和工作奠定了坚实的基础。

## 7.思考与练习

无

# 六.进程同步与信号量的实现实验

## 1.实验目的

加深对进程同步与互斥概念的理解。

掌握信号量的使用方法，并解决生产者—消费者问题。

掌握信号量的实现原理。

## 2.实验内容

在Linux内核中，编写实现信号量的代码，并编写Linux应用程序进行测试。

## 3.实验步骤

3.1**准备实验**

使用浏览器登录平台领取本次实验对应的任务，从而在平台上创建个人项目（Linux 0.11 内核项目），然后使用VSCode将个人项目克隆到本地磁盘中。

3.2**在内核中实现信号量的系统调用**

(1)在文件include/unistd.h中的第161行之后，定义四个新的系统调用号，如下：

#define \_\_NR\_sem\_open 87

#define \_\_NR\_sem\_wait 88

#define \_\_NR\_sem\_post 89

#define \_\_NR\_sem\_unlink 90

(2)在文件kernel/system\_call.s中的第73行，修改系统调用的总数：

nr\_system\_calls = 91

(3)在文件include/linux/sys.h中的第87行之后，添加系统调用内核函数的声明：

extern int sys\_sem\_open();

extern int sys\_sem\_wait();

extern int sys\_sem\_post();

extern int sys\_sem\_unlink();

在此文件的最后，向系统调用函数指针表sys\_call\_table[]中添加新系统调用函数的指针

fn\_ptrsys\_call\_table[] = {

……

sys\_sem\_open, //87

sys\_sem\_wait, //88

sys\_sem\_post, //89

sys\_sem\_unlink //90

};

(4)打开“学生包”文件夹，在本实验对应的文件夹中找到“sem.c”文件。将此文件拖动到VSCode中释放，即可打开此文件。其中实现了信号量的四个系统调用函数。

(5)在VSCode的“文件资源管理器”窗口中，右键点击“kernel”文件夹节点，选择菜单“New File”新建一个名为“semaphore.c”的文件，并将步骤4中找到的“sem.c”中的代码复制到刚刚创建的kernel/semaphore.c文件中。

(6)生成项目，确保没有语法错误。

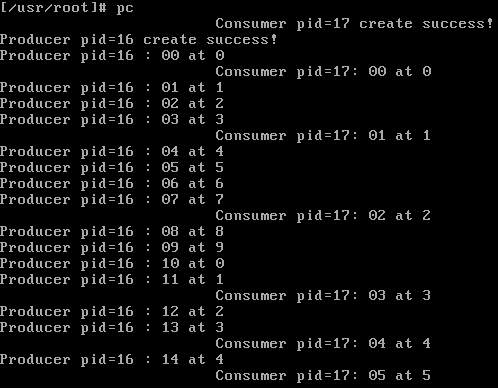
**3.3在Linux应用程序中使用信号量解决生产者-消费者问题**

(1)在VSCode的“Terminal”菜单中选择“Run Build Task...”，会在VSCode的顶部中间位置弹出一个可以执行的Task列表，选择其中的“打开 floppyb.img”后会使用Floppy Editor工具打开该项目中的floppyb.img文件，用于查看软盘镜像中的文件。将pc.c文件拖动到此工具窗口中释放，点击工具栏上的保存按钮后关闭此工具，这样就将pc.c文件复制到了软盘B中。

(2)按F5键启动调试，待Linux 0.11启动后，将软盘B中的pc.c文件复制到硬盘的当前目录，命令为：mcopy b:pc.c pc.c

(3)使用gcc编译pc.c文件，命令为：gcc pc.c -o pc

(4)执行sync命令将对硬盘的更改保存下来。

(5)执行pc命令，查看生产者—消费者同步执行的过程，如图7-1所示。

(6)使用命令pc > pc.txt将输出保存到文件pc.txt中，再使用命令vi pc.txt启动vi编辑器查看文本文件pc.txt中的内容。

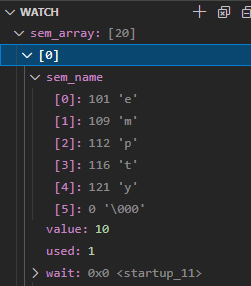
3.4**调试信号量的工作过程**

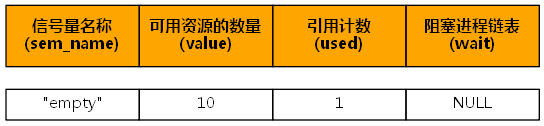
(1)创建信号量

①在文件kernel/semaphore.c的sys\_sem\_open函数中调用cli函数处（第90行）添加一个断点。

②按F5启动调试，在Linux的终端执行pc命令，会命中刚刚添加的断点。按F10执行cli函数后，再按F10执行调用了get\_name函数的代码行

③再按F10单步调试一行代码，会执行调用了find\_sem函数的代码行。

④继续按F10单步调试，直到从第112行返回(停止在第117行)选中第105行中信号量数组“sem\_array”，点击右键后选择“Addto Watch”，这样在左侧的监视窗口“WATCH”中可以查看在信号量数组中下标为0的信号量“empty”已经创建，如图7-2所示。

⑤在VSCode的“View”菜单中选择“Command Palette...”，会在VSCode的顶部中间位置显示命令面板，输入“Lab: New Visualizer View”命令后，VSCode会在其右侧弹出一个窗口让读者查看可视化视图。在右侧可视化视图顶部的编辑框中输入命令“#sem”后按回车，可以看到已经创建完毕的“empty”信号量，如图7-3所示。

⑥按F5继续运行，仍然会在之前添加的断点处中断，可以按照之前的步骤继续调试“full”和“mutex”信号量的创建过程。注意观察“WATCH”窗口中信号量数组下标为1的“full”信号量和下标为2的“mutex”信号量的变化。也在右侧可视化视图顶部的编辑框中输入命令“#sem”后按回车查看信号量的创建结果。

(2) **等待信号量和释放信号量**

**3.5实现一个生产者进程与多个消费者进程同步工作**

①使用gcc编译pc.c文件：gcc pc.c -o pc

②执行sync命令将对硬盘的更改保存下来。

③使用命令pc >> pc.txt将输出保存到文件pc.txt中（此过程可能时间较长，请读者耐心等待），再使用命令vi pc.txt启动vi编辑器查看文本文件pc.txt中的内容。

④在Linux 0.11的终端使用下面的命令将刚刚编写的文件pc.c复制到软盘B中。

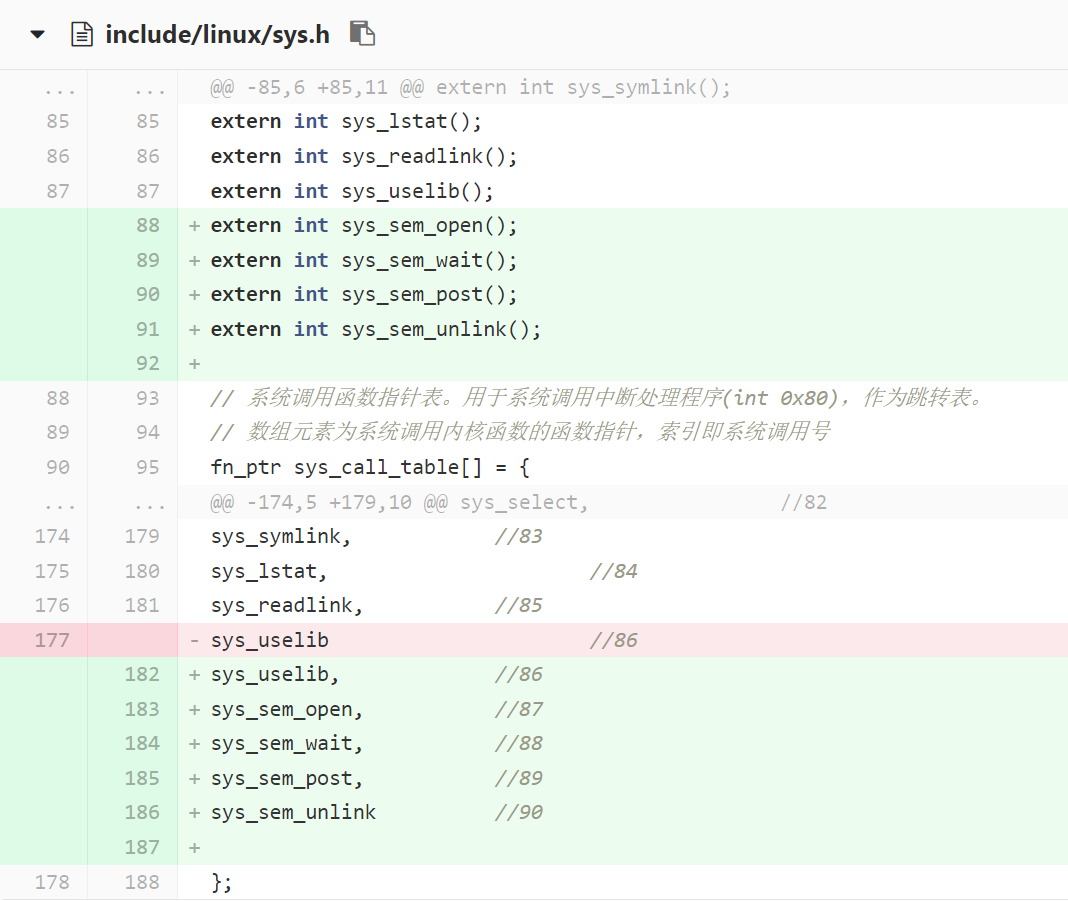
mcopypc.c b:pc.c

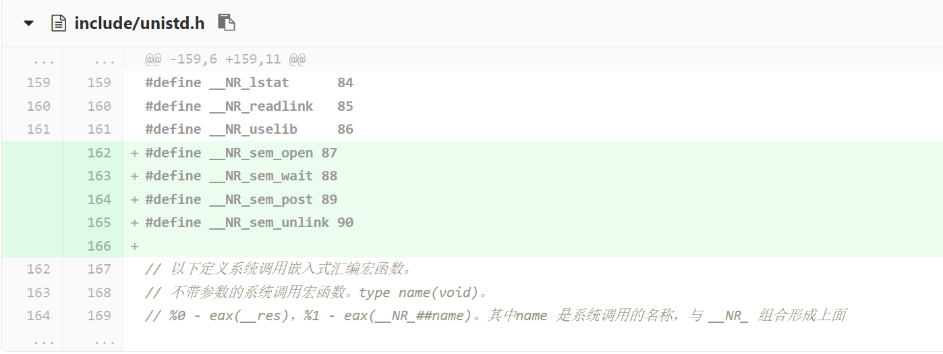
⑤关闭Bochs虚拟机。

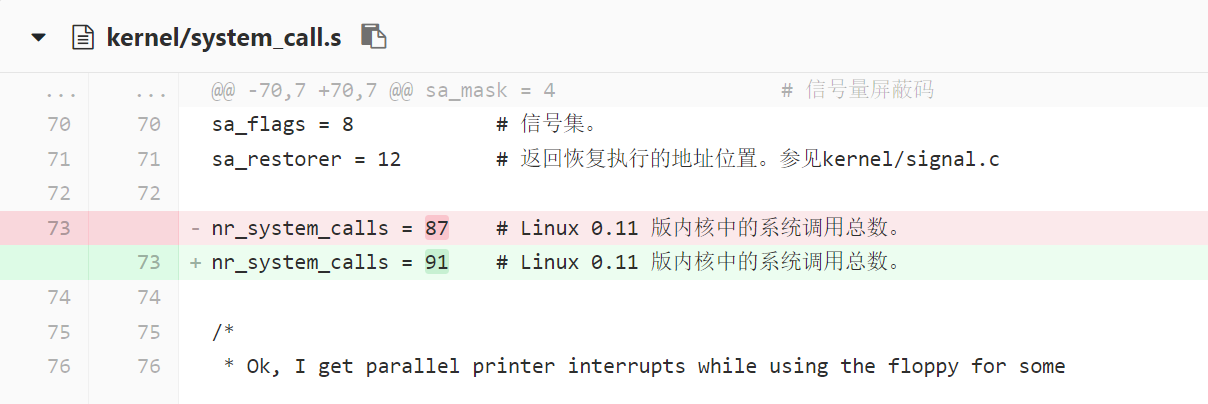
⑥在VSCode左侧的“文件资源管理器”窗口顶部点击“New Folder”按钮，新建一个名称为newapp的文件夹。在“文件资源管理器”窗口中的newapp文件夹节点上点击鼠标右键，选择菜单中的“Reveal in File Explorer”，可以使用Windows的资源管理器打开此文件夹所在的位置，双击打开此文件夹。

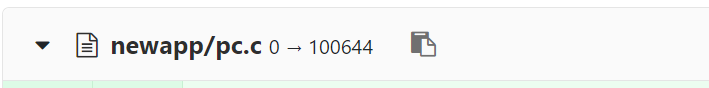
⑦在VSCode的“Terminal”菜单中选择“Run Build Task...”，会在VSCode的顶部中间位置弹出一个可以执行的Task列表，选择其中的“打开 floppyb.img”后会使用Floppy Editor工具打开该项目中的floppyb.img文件，查看软盘镜像中的文件列表，确保刚刚编写的文件已经成功复制到软盘镜像文件中。在文件列表中选中pc.c文件，并点击工具栏上的“复制”按钮，然后粘贴到Windows的资源管理器打开的newapp文件夹中。

## 4.运行结果

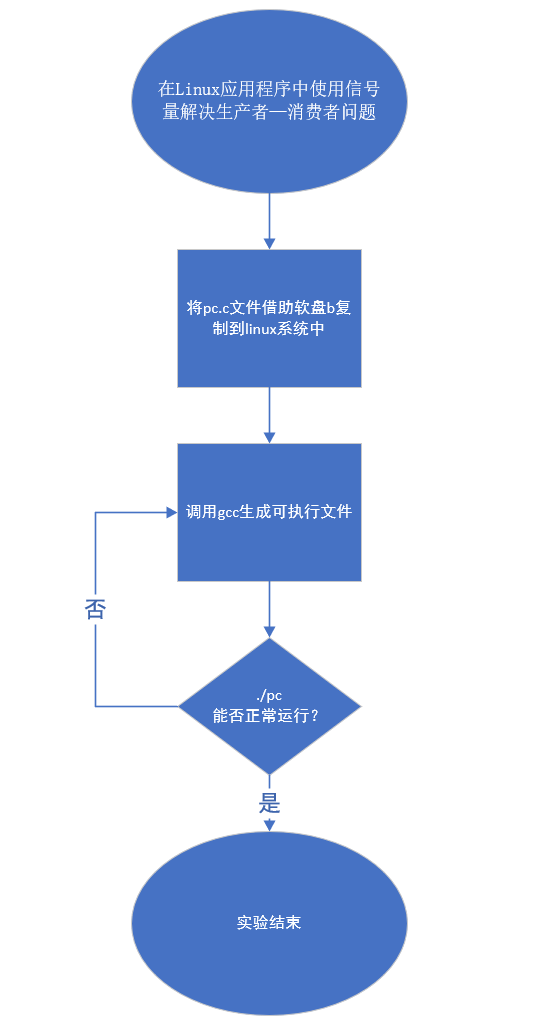
共修改七个文件

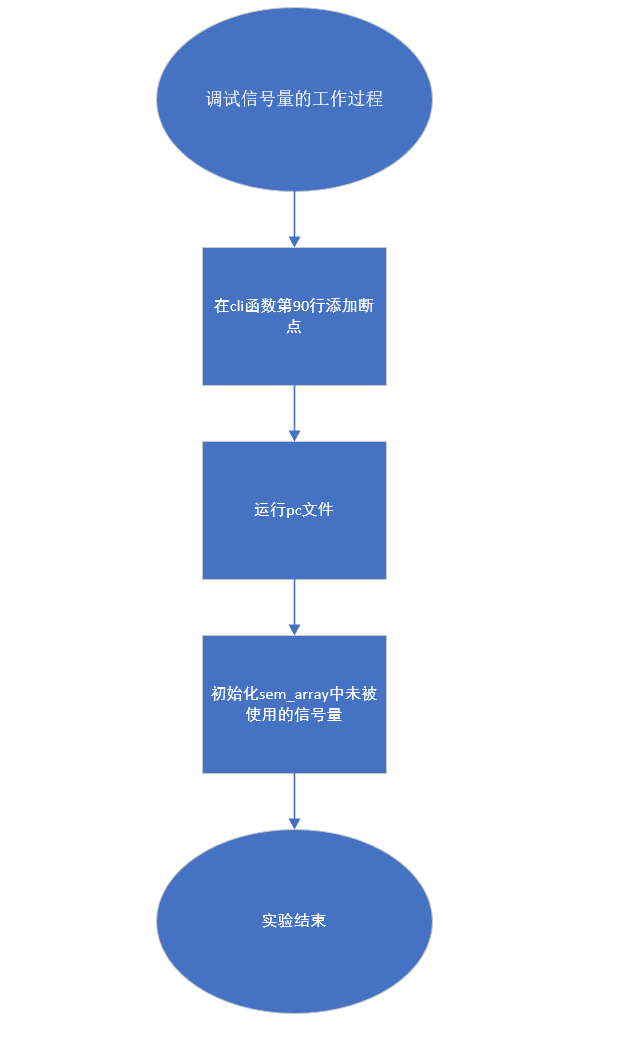


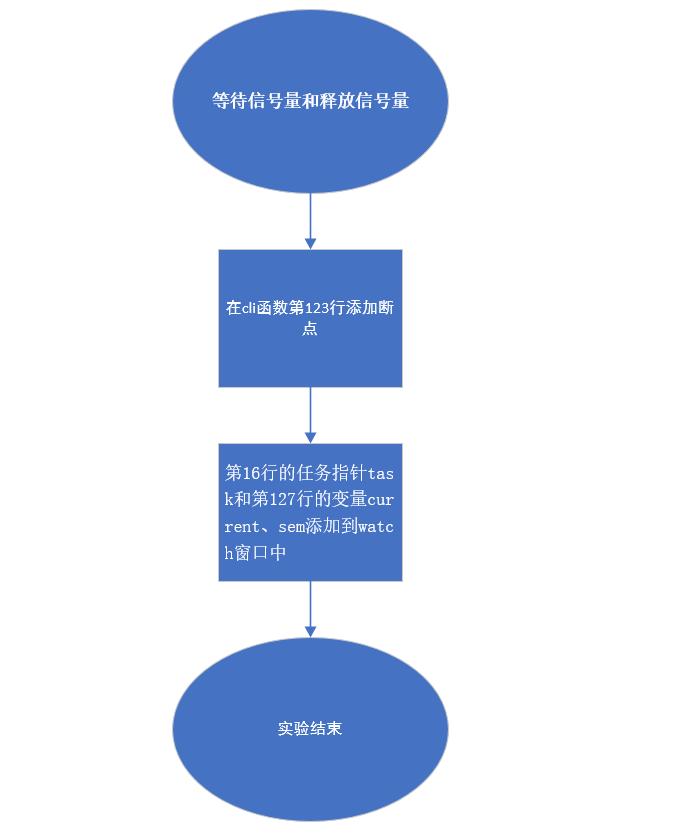




## 5.流程图







## 6.实验体会

在进程同步与信号量的实践探索中，我深刻领会了进程同步和互斥的基本原理，并成功应用信号量解决了经典的生产者-消费者问题。

实验过程中，我认识到进程同步确保了多个进程间有序的执行流程，而互斥则保证了在任意时刻只有一个进程能够访问共享资源。这两个概念在操作系统中扮演着至关重要的角色。为了实现这一机制，我利用了信号量这一工具。信号量，本质上是一个计数器，用于精确地控制多个进程对共享资源的访问。

在解决生产者-消费者问题时，我巧妙地运用了两个信号量：一个用来监控缓冲区是否为空，另一个则监控缓冲区是否已满。通过这两个信号量的精确控制，我成功地构建了一个高效且稳定的生产者-消费者模型。

在掌握信号量使用方法的同时，我也对信号量的实现原理产生了浓厚的兴趣。信号量的实现依赖于原子操作，即在多进程环境中，对信号量值的修改必须是不可分割的。当一个进程想要访问共享资源时，它会尝试减少信号量的值；如果信号量的值已经为0，则进程会被阻塞，直到其他进程增加了信号量的值。同样，当一个进程完成对共享资源的访问后，它会增加信号量的值，从而允许其他进程访问。

通过这次实验，我不仅对进程同步和互斥有了更深刻的理解，还掌握了信号量的实际应用和实现原理，为未来的学习和工作打下了坚实的基础。

## 7.思考与练习

1. 本实验的设计者在第一次编写生产者--消费者程序的时候，是这么做的：

|  |  |
| --- | --- |
| Producer()  {  P(Mutex); //互斥信号量  生产一个产品item;  P(Empty); //空闲缓存资源  将item放到空闲缓存中;  V(Full); //产品资源  V(Mutex);  } | Consumer()  {  P(Mutex);  P(Full);  从缓存区取出一个赋值给item;  V(Empty);  消费产品item;  V(Mutex);  } |

这样可行吗？如果可行，那么它和标准解法在执行效果上会有什么不同？如果不可行，那么它有什么问题使它不可行？

不可行。

缺少对互斥信号量的定义：在代码中，**Mutex**被用作互斥信号量，但没有给出其具体实现或定义。通常情况下，互斥信号量需要初始化为1，并在临界区代码前后使用P和V操作来实现临界区的互斥访问。

缺少对空闲缓存资源和产品资源的定义：**Empty**和**Full**被用来表示空闲缓存资源和产品资源的信号量，但同样没有给出其具体实现或定义。这些信号量应该在开始时初始化为缓冲区的大小，并在生产者和消费者之间同步缓冲区的状态。

生产者和消费者的顺序：在这个实现中，生产者和消费者的顺序是交替执行的。这意味着当生产者生产一个产品后，消费者可能会立即消费它，而不管是否有其他产品在等待被消费。这可能会导致资源浪费或者不必要的等待。

2.本实验设计的生产者--消费者问题是在同一个应用程序（同一个main函数）中实现的，请读者试着将生产者和消费者在两个不同的应用程序中实现。

分别在两个Linux应用程序项目中分别实现生产者程序和消费者程序，生成各自的项目从而得到应用程序的可执行文件，此时，应用程序的可执行文件已经自动写入各自项目文件夹中的软盘镜像文件floppyb.img中了。将消费者项目文件夹下的floppyb.img拷贝覆盖已经实现了信号量功能的Linux Kernel项目文件夹下的floppyb.img文件，在内核项目中按F5启动调试后，按顺序执行命令：

mcopyb:linuxapp.exe consumer

chmod +x consumer

sync

从而将消费者程序复制到硬盘，然后结束调试。

将生产者项目文件夹下的floppyb.img拷贝覆盖本已经实现了信号量功能的Linux Kernel项目文件夹下的floppyb.img文件，在内核项目中按F5启动调试后，按顺序执行命令：

mcopyb:linuxapp.exe producer

chmod +x producer

sync

从而将消费者程序复制到硬盘。

这样在内核项目的硬盘上就同时存在了生产者和消费者应用程序的可执行文件。先执行命令：

consumer &

此命令会让一个消费者进程在后台开始执行。此时读者可以立即执行下面的命令开始执行生产者进程。如果读者已经完成了3.5的内容，使多个消费者可以共享文件中的游标的话，可以再执行3次之前的命令，然后再执行下面的命令：

producer

这样就可以查看1个生产者和4个消费者同步运行的结果了。可以将输出结果保存到文件中查看，具体操作步骤可参考本实验3.3的步骤6，但此处输出到文件需要使用“>>”符号在文件的尾部追加写入，如果使用’>’符号，每次都会从文件开始处写入，则无法看到正确的执行结果。另外，“>>”和文件名需要写到 “&”的前面，这样才能先重定向到文件再在后台运行。