

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

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

目 录

第一部分 进程管理（任务四—九）	1
任务四 系统调用	1
1. 实验目的	1
2. 实验内容	1
3. 实验步骤	1
4. 运行结果	4
5. 流程图	6
6. 实验体会	8
7. 思考与练习	8
任务五 在 Linux 0.11 应用程序中调用 fork 函数创建子进程	11
1. 实验目的	11
2. 实验内容	11
3. 实验步骤	11
4. 运行结果	15
5. 流程图	16
6. 实验体会	18
7. 思考与练习	18
任务六 调用 execve 函数加载执行一个新程序	18
1. 实验目的	18
2. 实验内容	18
3. 实验步骤	18
4. 运行结果	20
5. 流程图	21
6. 实验体会	23
7. 思考与练习	23
任务七、八 进程的状态与进程调度	25
1. 实验目的	25
2. 实验内容	25
3. 实验步骤	25
4. 运行结果	31
5. 流程图	33
6. 实验体会	37
7. 思考与练习	37
任务九 进程同步与信号量的实现	37
1. 实验目的	37
2. 实验内容	37
3. 实验步骤	38
4. 运行结果	41
5. 流程图	43
6. 实验体会	46
7. 思考与练习	47
第二部分 内存管理（任务十一—十六）	49

任务十 分配和释放物理页.....	49
1. 实验目的.....	49
2. 实验内容.....	49
3. 实验步骤.....	49
4. 运行结果.....	52
5. 流程图.....	53
6. 实验体会.....	54
7. 思考与练习.....	54
任务十一 跟踪 Linux 应用程序中的逻辑地址、线性地址、物理地址的映射过程 ...	54
1. 实验目的.....	54
2. 实验内容.....	54
3. 实验步骤.....	54
4. 运行结果.....	57
5. 流程图.....	59
6. 实验体会.....	60
7. 思考与练习.....	60
任务十二 输出应用程序进程的页目录和页表.....	60
1. 实验目的.....	60
2. 实验内容.....	60
3. 实验步骤.....	60
4. 运行结果.....	62
5. 流程图.....	63
6. 实验体会.....	64
7. 思考与练习.....	64
任务十三 用共享内存做缓冲区解决生产者—消费者问题.....	64
1. 实验目的.....	64
2. 实验内容.....	64
3. 实验步骤.....	64
4. 运行结果.....	65
5. 流程图.....	67
6. 实验体会.....	67
7. 思考与练习.....	68
任务十四 页面置换算法.....	70
1. 实验目的.....	70
2. 实验内容.....	70
3. 实验步骤.....	70
4. 运行结果.....	73
5. 流程图.....	74
6. 实验体会.....	74
7. 思考与练习.....	74
任务十五 动态内存分配 - 边界标识法.....	75
1. 实验目的.....	75
2. 实验内容.....	75
3. 实验步骤.....	75

4. 运行结果.....	77
5. 流程图.....	78
6. 实验体会.....	79
7. 思考与练习.....	79
任务十六 动态内存分配 -伙伴系统.....	79
1. 实验目的.....	79
2. 实验内容.....	79
3. 实验步骤.....	79
4. 运行结果.....	82
5. 流程图.....	82
6. 实验体会.....	82
7. 思考与练习.....	83
第三部分 设备管理（任务十七—十八）.....	86
任务十七 字符显示的控制.....	86
1. 实验目的.....	86
2. 实验内容.....	86
3. 实验步骤.....	86
4. 运行结果.....	87
5. 流程图.....	88
6. 实验体会.....	91
7. 思考与练习.....	91
任务十八 实现贪吃蛇游戏.....	91
1. 实验目的.....	91
2. 实验内容.....	91
3. 实验步骤.....	91
4. 运行结果.....	94
5. 流程图.....	95
6. 实验体会.....	96
7. 思考与练习.....	96
第四部分 文件管理（任务十九—二十）.....	97
任务十九 proc 文件系统的实现.....	97
1. 实验目的.....	97
2. 实验内容.....	97
3. 实验步骤.....	97
4. 运行结果.....	99
5. 流程图.....	101
6. 实验体会.....	103
7. 思考与练习.....	103
任务二十 MINIX 1.0 文件系统的实现.....	105
1. 实验目的.....	105
2. 实验内容.....	105
3. 实验步骤.....	105
4. 运行结果.....	107
5. 流程图.....	107

6. 实验体会.....	107
7. 思考与练习.....	108

第一部分 进程管理（任务四一九）

任务四 系统调用

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

161	#define __NR_uselib	86
162	+ #define __NR_max	87

图 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：

	- nr_system_calls = 87
73	+ nr_system_calls = 88

图 2

(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。

	- sys_uselib
178	+ sys_uselib,
179	+ sys_max

图 3

(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。

3.2.5 生成 Linux 0.11 内核，修改语法错误直到生成成功。

87	extern int sys_uselib();
88	+ extern int sys_max();

图 4

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_syscall12 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

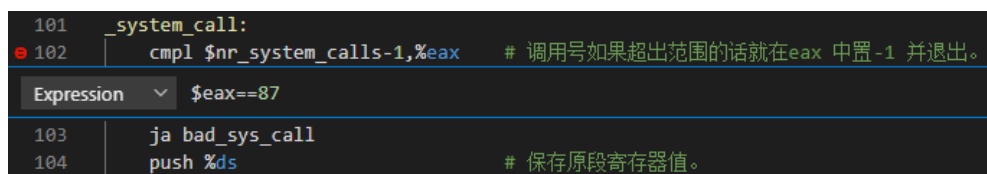


图 5

(4) 按 F5 启动调试。

(5) 待 Linux 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：

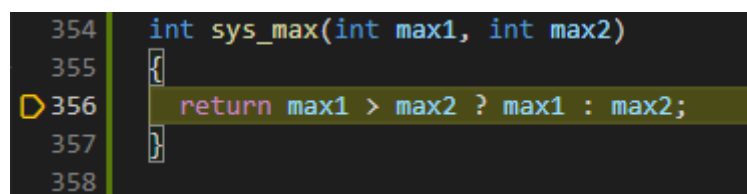


图 6

(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：

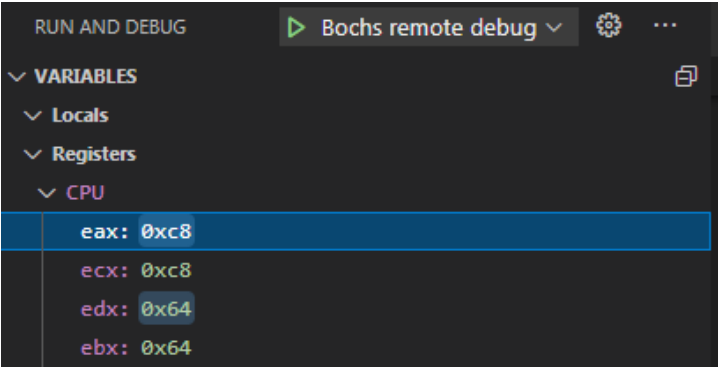


图 7

(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：

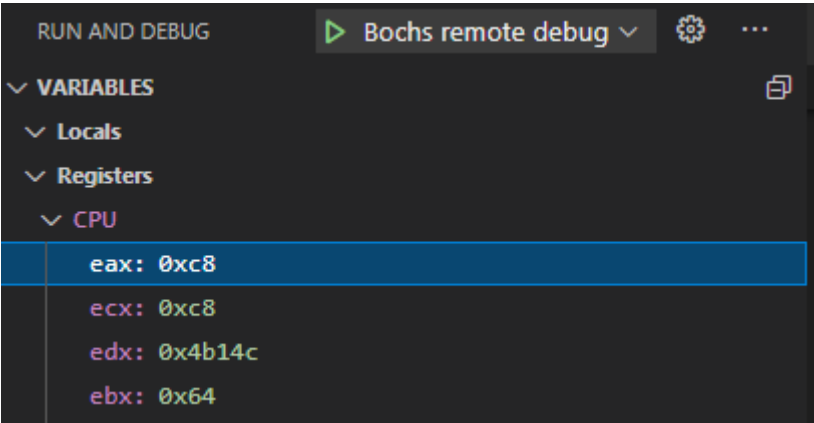


图 8

(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。

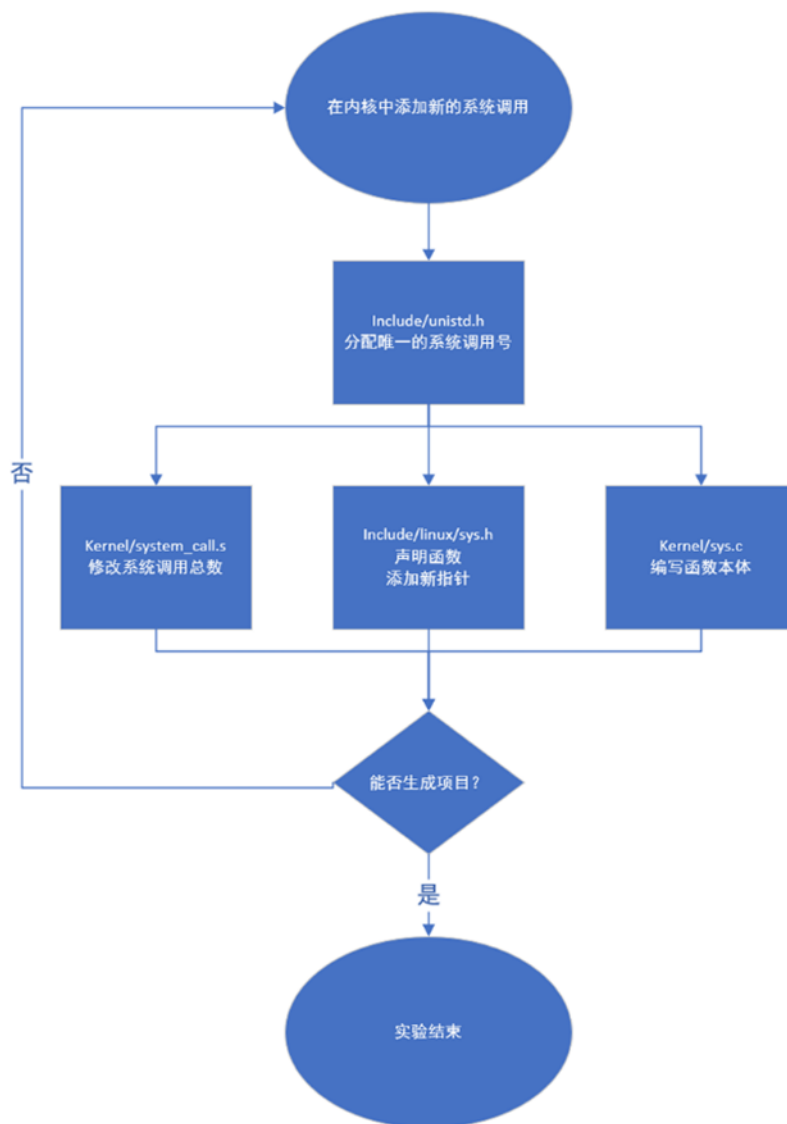
4. 运行结果

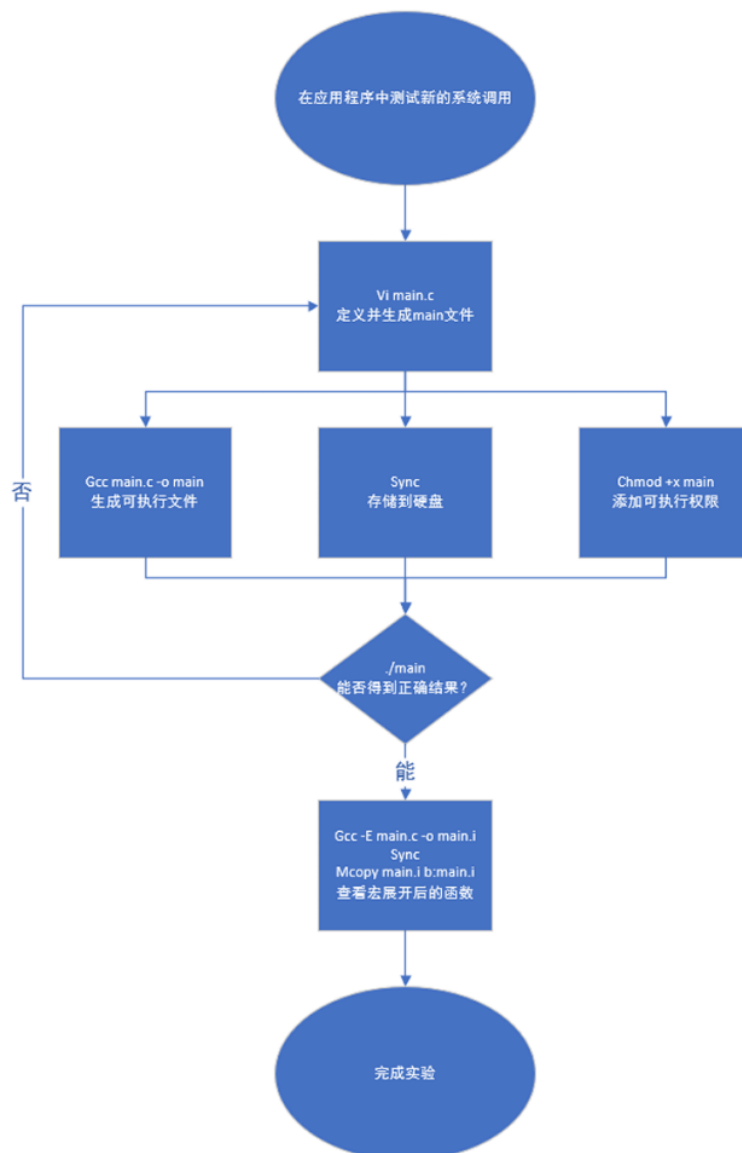
共修改 4 个文件包含 9 行增加和 2 行删除

▼ include/linux/sys.h		
...	...	@@ -85,6 +85,7 @@ extern int sys_symlink();
85	85	extern int sys_lstat();
86	86	extern int sys_readlink();
87	87	extern int sys_uselib();
	88	+ extern int sys_max();
88	89	// 系统调用函数指针表。用于系统调用中断处理程序(int 0x80)，作为跳转表。
89	90	// 数组元素为系统调用内核函数的函数指针，索引即系统调用号
90	91	fn_ptr sys_call_table[] = {
...	...	@@ -174,5 +175,6 @@ sys_select, //82
174	175	sys_symlink, //83
175	176	sys_lstat, //84
176	177	sys_readlink, //85
177		- sys_uselib //86
	178	+ sys_uselib, //86
	179	+ sys_max
178	180	};
▼ include/unistd.h		
...	...	@@ -159,6 +159,7 @@
159	159	#define __NR_lstat 84
160	160	#define __NR_readlink 85
161	161	#define __NR_uselib 86
	162	+ #define __NR_max 87
162	163	// 以下定义系统调用嵌入式汇编宏函数。
163	164	// 不带参数的系统调用宏函数。type name(void)。
164	165	// %0 - eax(__res), %1 - eax(__NR_##name)。其中name 是系统调用的名称，与 __NR_ 组合形成上面
...	...	
▼ kernel/sys.c		
...	...	@@ -350,3 +350,7 @@ int sys_settimeofday()
350	350	{
351	351	return -ENOSYS;
352	352	}
	353	+ int sys_max(int max1,int max2)
	354	+ {
	355	+ return max1 > max2 ? max1 : max2;
	356	+ }
▼ kernel/system_call.s		
...	...	@@ -70,7 +70,7 @@ sa_mask = 4 # 信号量屏蔽码
70	70	sa_flags = 8 # 信号集。
71	71	sa_restorer = 12 # 返回恢复执行的地址位置。参见kernel/signal.c
72	72	
73		- nr_system_calls = 87 # Linux 0.11 版内核中的系统调用总数。
	73	+ nr_system_calls = 88 # Linux 0.11 版内核中的系统调用总数。
74	74	
75	75	/*
76	76	* Ok, I get parallel printer interrupts while using the floppy for some
...	...	

图 9

5. 流程图





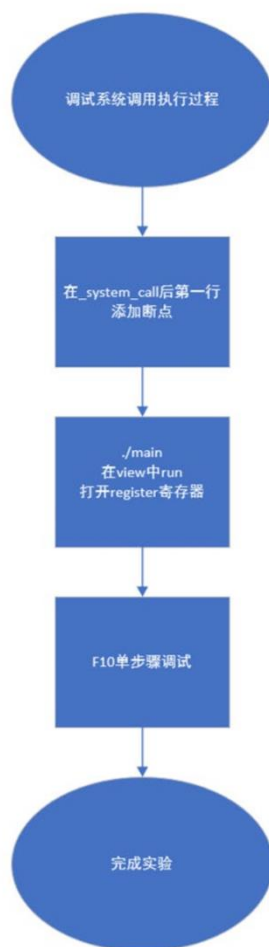


图 10

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;
}

```

任务五 在 Linux 0.11 应用程序中调用 fork 函数创建子进程

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。

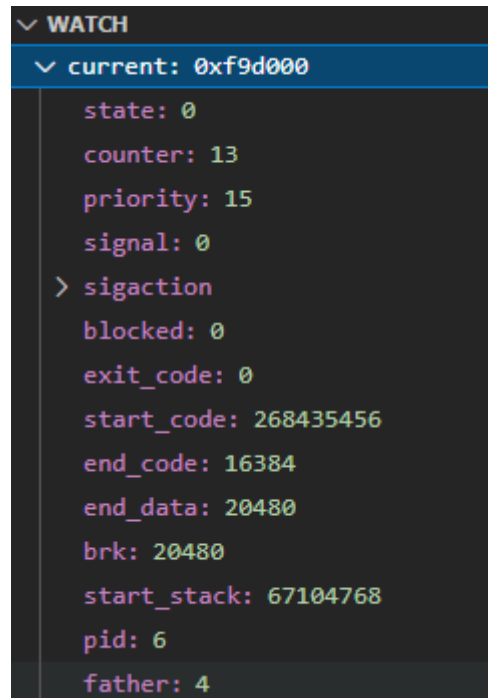


图 11

(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。

```
▼ task: [64]
  > [0]: 0x171c0 <init_task>
  > [1]: 0xffff000
  > [2]: 0xffd000
  > [3]: 0xfc1000
  ▼ [4]: 0xf9d000
    state: 0
    counter: 13
    priority: 15
    signal: 0
  > sigaction
    blocked: 0
    exit_code: 0
    start_code: 268435456
    end_code: 16384
    end_data: 20480
    brk: 20480
    start_stack: 67104768
    pid: 6
    father: 4
```

图 12

(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。

下标	进程 ID (pid)	状态 (state)	优先级 (priority)	剩余时间片 (counter)	父进程 ID (father)
0	0	TASK_RUNNING (0)	15	2	-1
1	1	TASK_INTERRUPTIBLE (1)	15	28	0
2	4	TASK_INTERRUPTIBLE (1)	15	5	1
3	3	TASK_INTERRUPTIBLE (1)	15	24	1
4	6	TASK_RUNNING (0)	15	10	4

current ←

图 13

- (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。

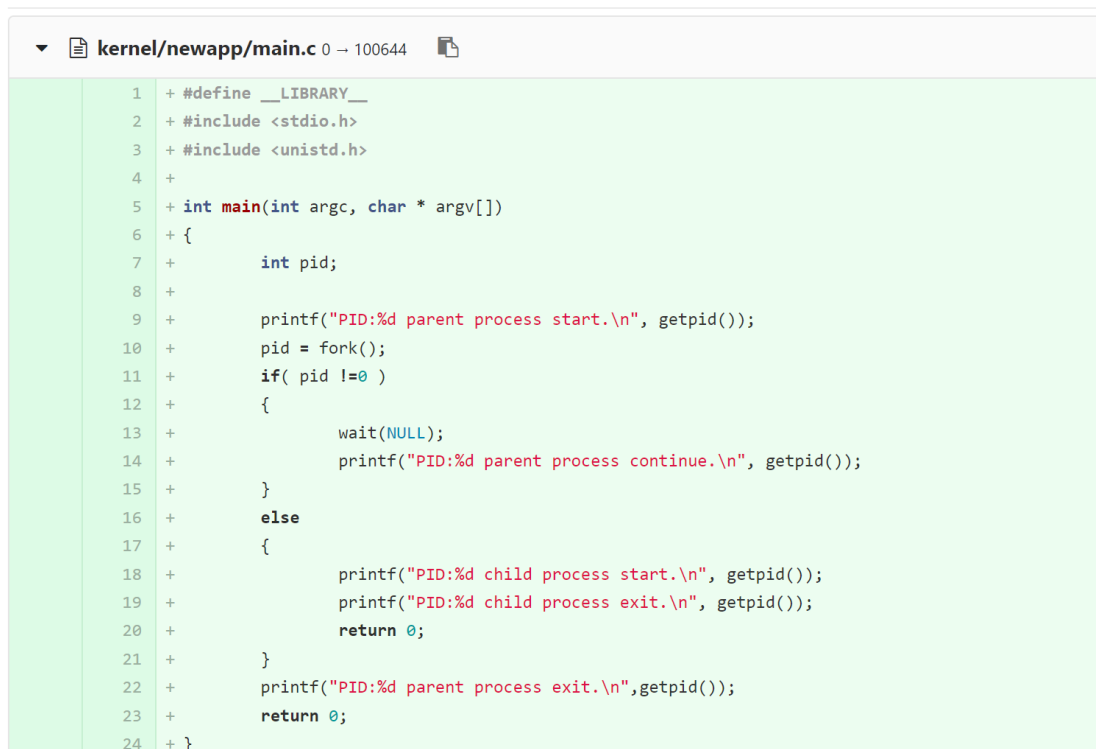
3.5 提交作业

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

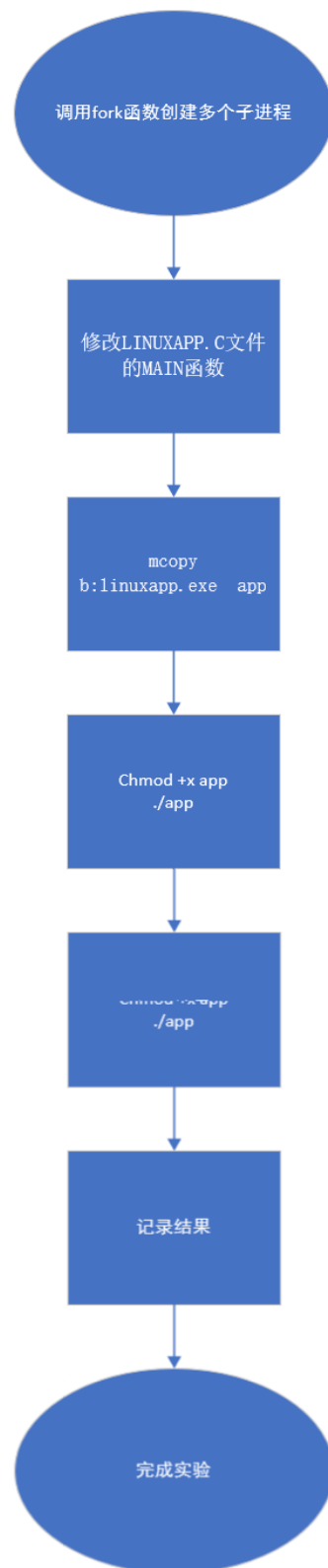
正在显示 1 个修改的文件 ▾



```
kernel/newapp/main.c 0 → 100644
1 + #define __LIBRARY__
2 + #include <stdio.h>
3 + #include <unistd.h>
4 +
5 + int main(int argc, char * argv[])
6 + {
7 +     int pid;
8 +
9 +     printf("PID:%d parent process start.\n", getpid());
10 +     pid = fork();
11 +     if( pid !=0 )
12 +     {
13 +         wait(NULL);
14 +         printf("PID:%d parent process continue.\n", getpid());
15 +     }
16 +     else
17 +     {
18 +         printf("PID:%d child process start.\n", getpid());
19 +         printf("PID:%d child process exit.\n", getpid());
20 +         return 0;
21 +     }
22 +     printf("PID:%d parent process exit.\n",getpid());
23 +     return 0;
24 + }
```

图 14

5. 流程图



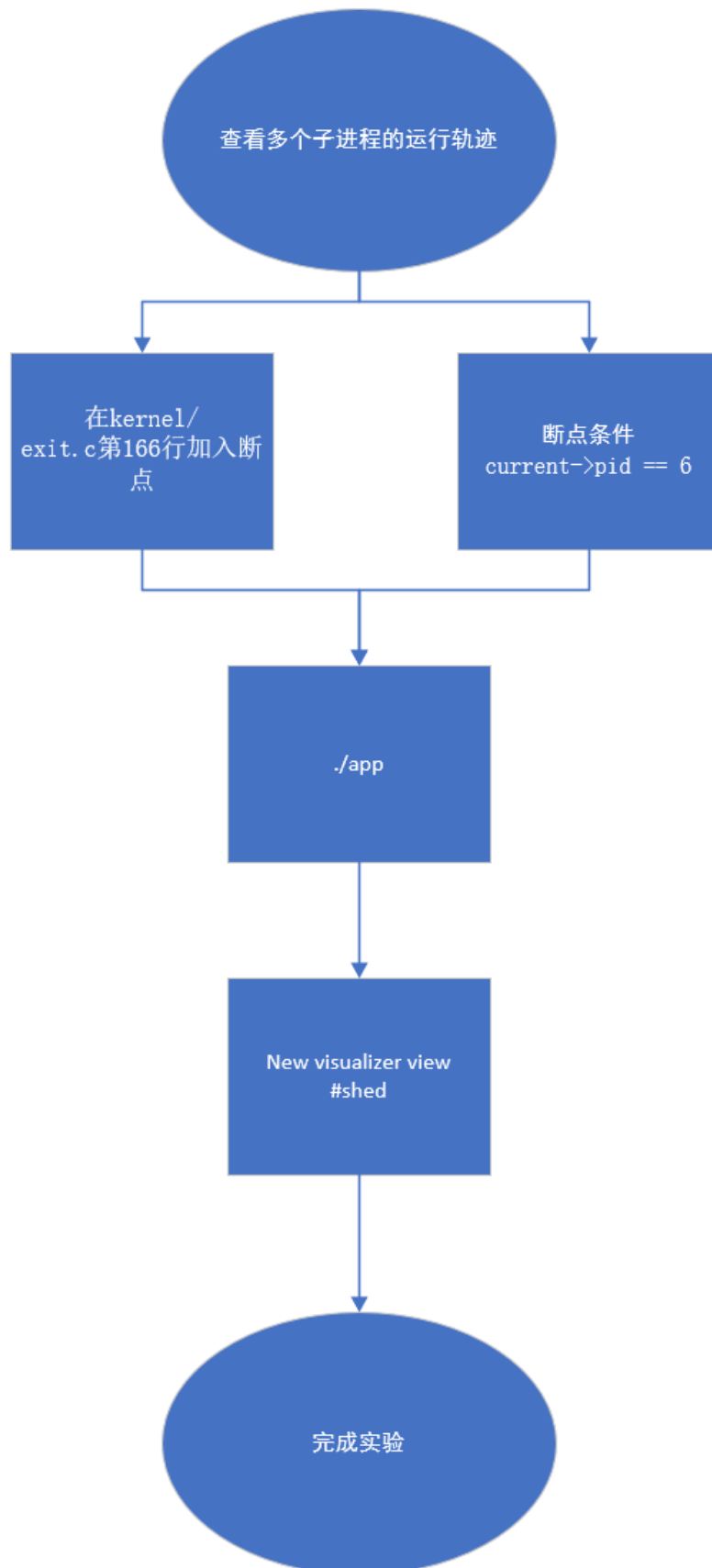


图 15

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

任务六 调用 `execve` 函数加载执行一个新程序

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 &¤t!=0 &¤t->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。

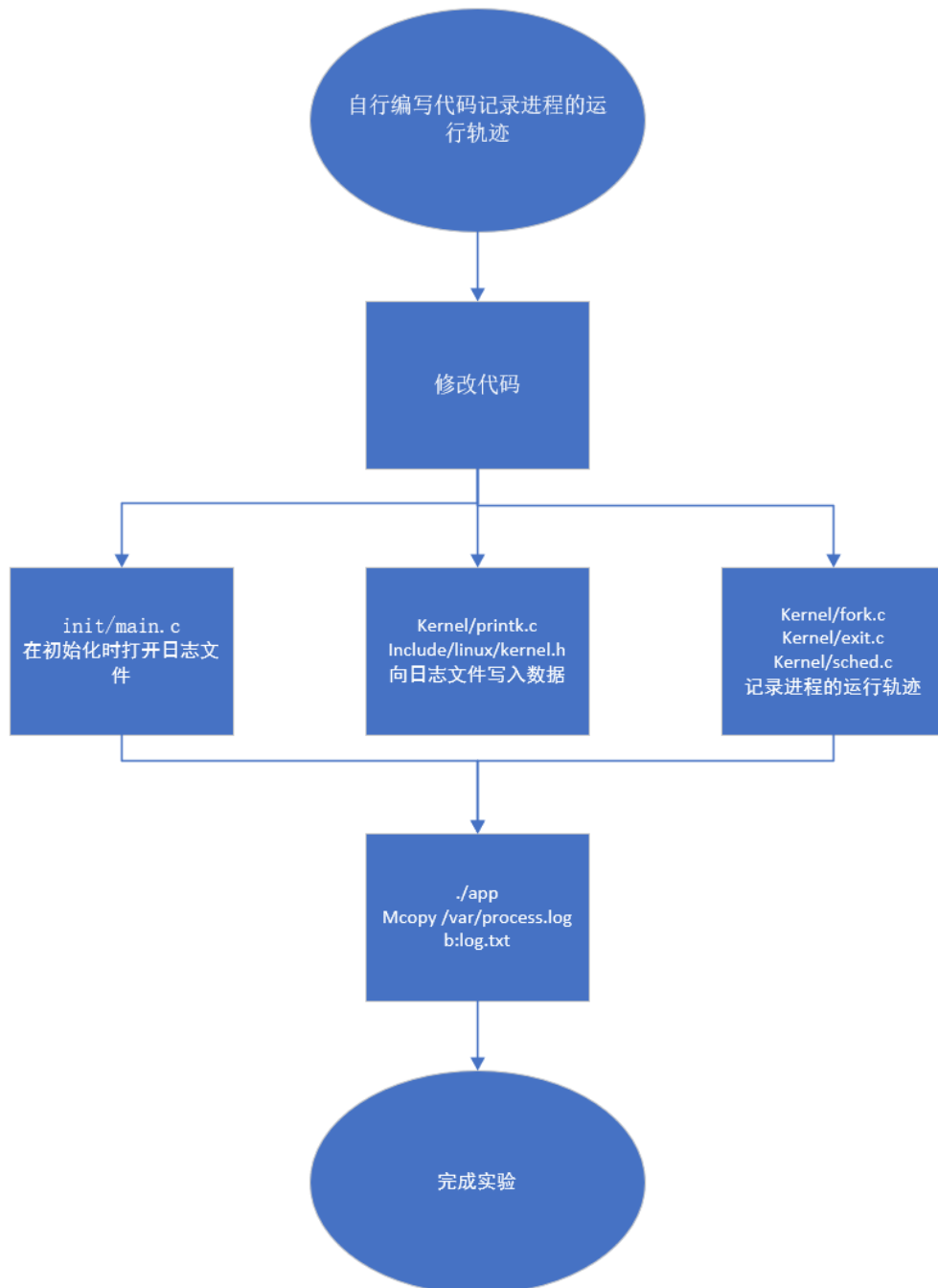
4. 运行结果

共修改 2 个文件



图 16

5. 流程图



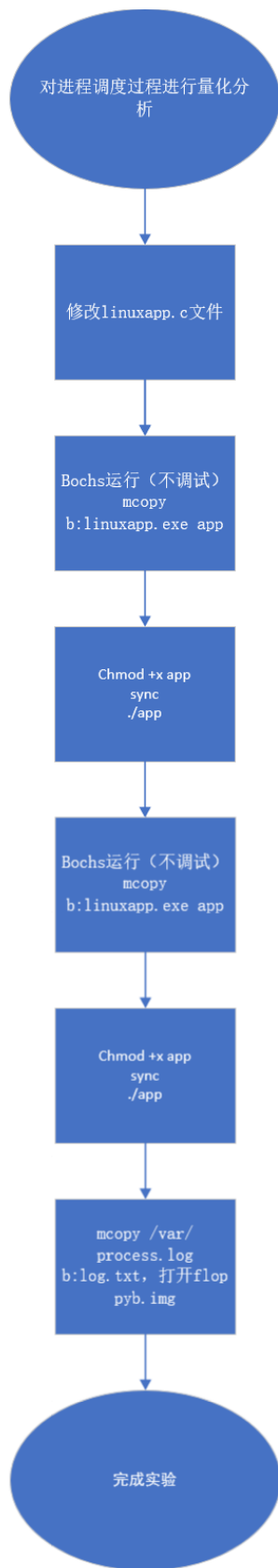


图 17

6. 实验体会

在多线程编程的领域中，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。在多线程环境中，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，多线程编程是 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");
        }
    }
}

```

```
    return 0;
}
```

任务七、八 进程的状态与进程调度

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。

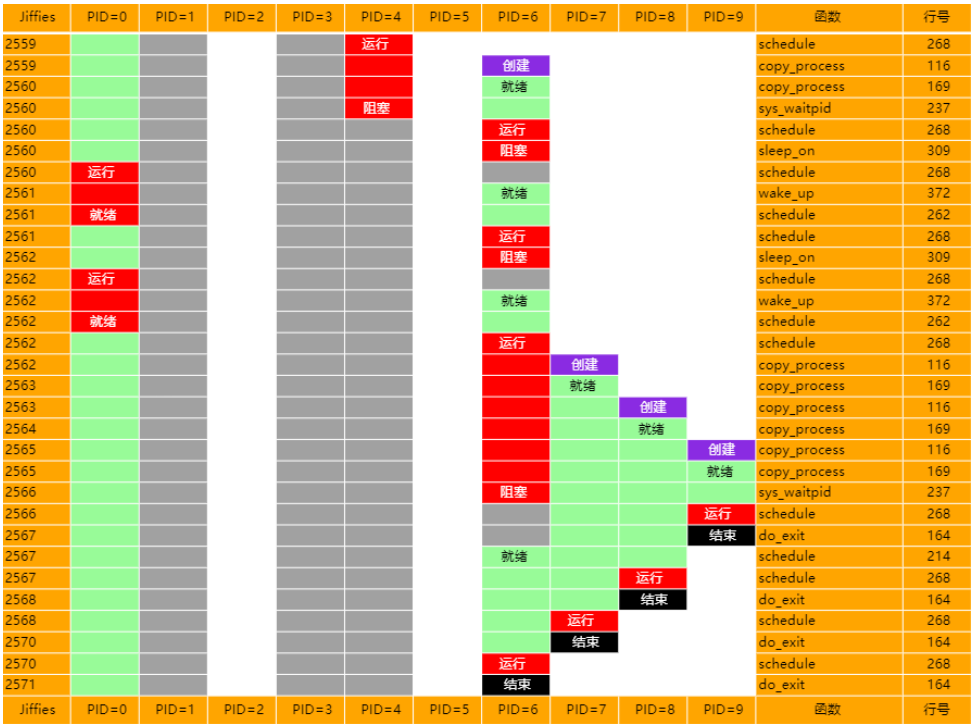


图 18

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 fprintf 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 fprintf( int fd, const char * fmt, ... )
{
    va_list args;
    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

```
▼ linuxapp.c
1 1 #define __LIBRARY__
2 2 #include "linuxapp.h"
3 + #include<sys/wait.h>
4 + #include<linux/sched.h>
5 + #include<time.h>
6 + void cpuio_bound( int last, int cpu_time, int io_time )
7 + {
8 +     struct tms start_time, current_time;
9 +     clock_t utime, stime;
10 +     int sleep_time;
11 +     while( last>0 )
12 +     {
13 +         times( &start_time );
14 +         do
15 +         {
16 +             times( &current_time );
17 +             utime=current_time.tms_utime-start_time.tms_utime;
18 +             stime=current_time.tms_stime-start_time.tms_stime;
19 +         }while( ( ( utime+stime )/HZ )<cpu_time );
20 +         last-=cpu_time;
21 +         if( last<=0 )
22 +             break;
23 +         sleep_time=0;
24 +         while( sleep_time<io_time )
25 +         {
26 +             sleep( 1 );
27 +             sleep_time++;
28 +         }
29 +         last-=sleep_time;
30 +     }
31 + }
```

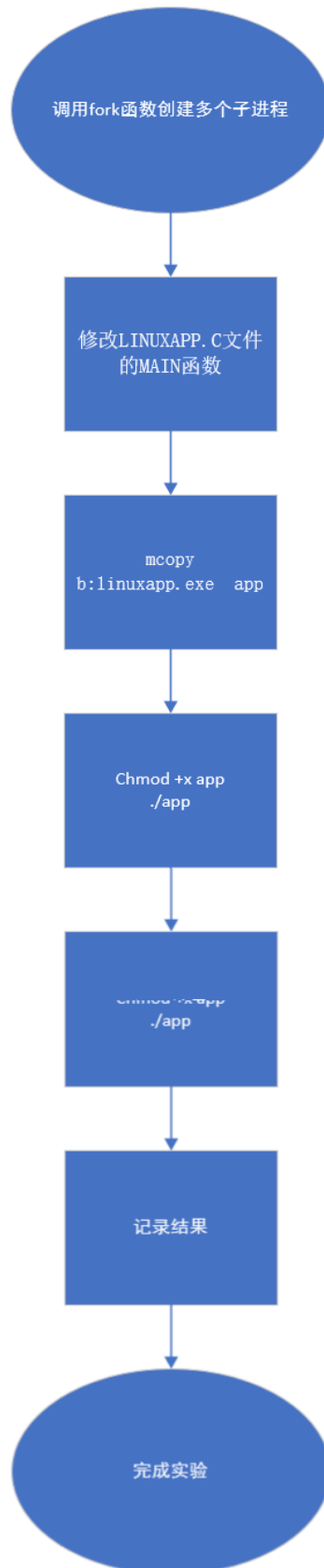
```

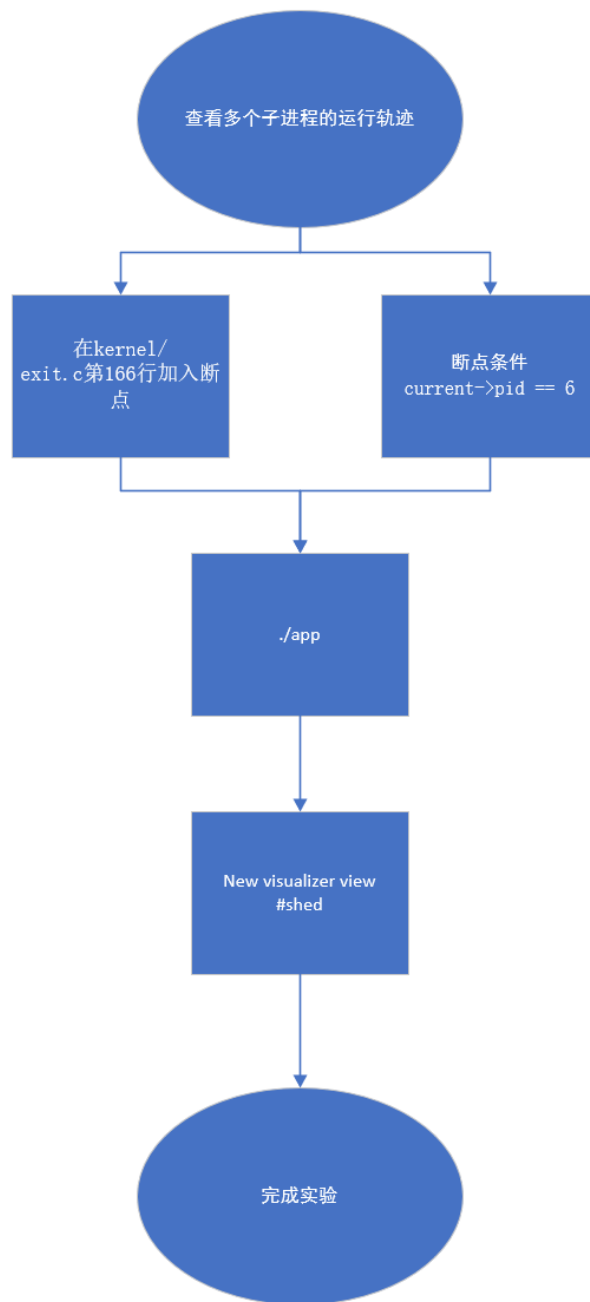
33  int main(int argc, char* argv[])
34  {
35      /* 注意: 在应用程序中不能使用断点等调试功能 */
36
37      /* TODO: 在此处添加自己的代码 */
-      printf("Hello world!\n");
-
38  +      pid_t p1, p2, p3, p4;
39  +      if( ( p1=fork() )==0 )
40  +      {          printf( "in child1\n" ); cpuio_bound( 5, 2, 2 );}
41  +      else if( ( p2=fork() )==0 )
42  +      {          printf( "in child2\n" ); cpuio_bound( 5, 4, 0 );}
43  +      else if( ( p3=fork() )==0 )
44  +      {          printf( "in child3\n" ); cpuio_bound( 5, 0, 4 );}
45  +      else if( ( p4=fork() )==0 )
46  +      {          printf( "in child4\n" ); cpuio_bound( 4, 2, 2 );}
47  +      else
48  +      {
49  +          printf( "====This is parent process====\n" );
50  +          printf( "pid=%d\n", getpid() );
51  +          printf( "pid1=%d\n", p1 );
52  +          printf( "pid2=%d\n", p2 );
53  +          printf( "pid3=%d\n", p3 );
54  +          printf( "pid4=%d\n", p4 );
55  +      }
56  +      wait( NULL );
57      return 0;
58  }

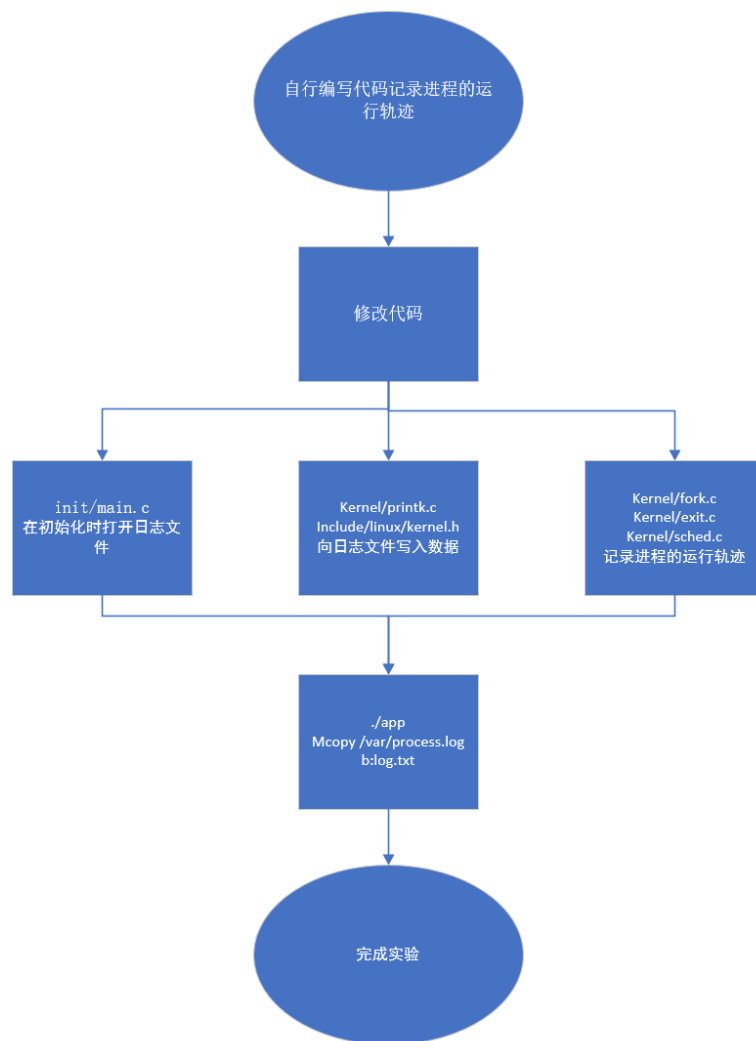
```

图 19

5. 流程图







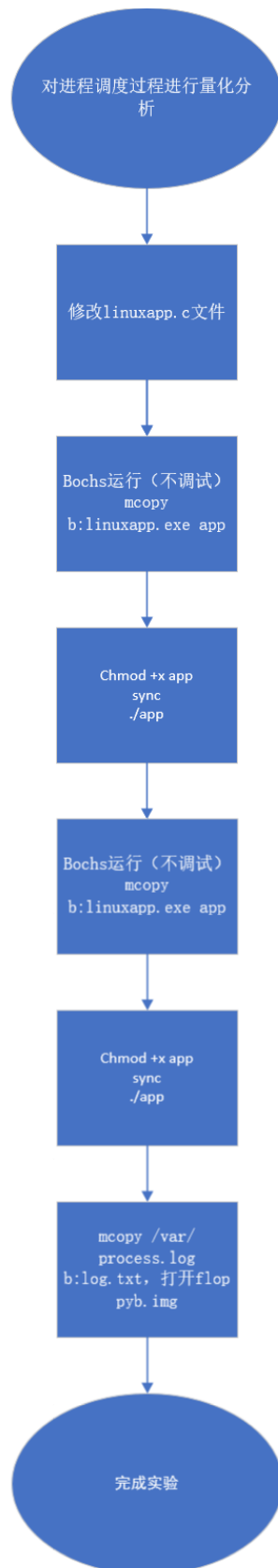


图 20

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 we 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，也为我的未来的学习和工作奠定了坚实的基础。

7. 思考与练习

无

任务九 进程同步与信号量的实现

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。

```
l/usr/root1# pc
Consumer pid=17 create success!
Producer pid=16 create success!
Producer pid=16 : 00 at 0
Consumer pid=17: 00 at 0
Producer pid=16 : 01 at 1
Producer pid=16 : 02 at 2
Producer pid=16 : 03 at 3
Consumer pid=17: 01 at 1
Producer pid=16 : 04 at 4
Producer pid=16 : 05 at 5
Producer pid=16 : 06 at 6
Producer pid=16 : 07 at 7
Consumer pid=17: 02 at 2
Producer pid=16 : 08 at 8
Producer pid=16 : 09 at 9
Producer pid=16 : 10 at 0
Producer pid=16 : 11 at 1
Consumer pid=17: 03 at 3
Producer pid=16 : 12 at 2
Producer pid=16 : 13 at 3
Consumer pid=17: 04 at 4
Producer pid=16 : 14 at 4
Consumer pid=17: 05 at 5
```

图 21

- (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。

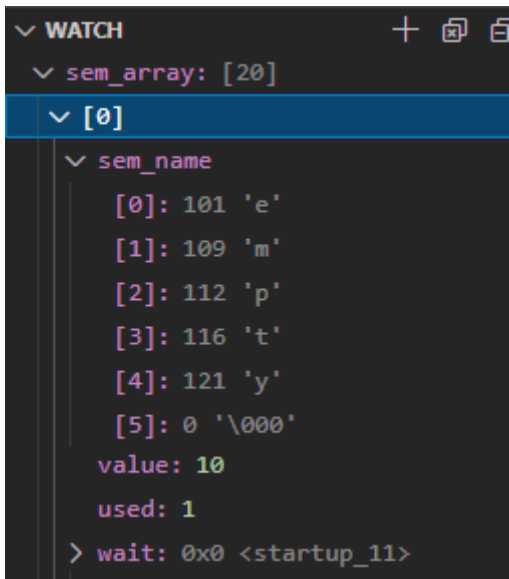


图 22

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。

信号量名称 (sem_name)	可用资源的数量 (value)	引用计数 (used)	阻塞进程链表 (wait)
"empty"	10	1	NULL

图 23

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

```

▼ include/linux/sys.h
...    ...    @@ -85,6 +85,11 @@ extern int sys_symlink();
85    85    extern int sys_lstat();
86    86    extern int sys_readlink();
87    87    extern int sys_uselib();
      88    + extern int sys_sem_open();
      89    + extern int sys_sem_wait();
      90    + extern int sys_sem_post();
      91    + extern int sys_sem_unlink();
      92    +
88    93    // 系统调用函数指针表。用于系统调用中断处理程序(int 0x80)，作为跳转表。
89    94    // 数组元素为系统调用内核函数的函数指针，索引即系统调用号
90    95    fn_ptr sys_call_table[] = {
...    ...    @@ -174,5 +179,10 @@ sys_select,           //82
174    179    sys_symlink,           //83
175    180    sys_lstat,           //84
176    181    sys_readlink,       //85
177    - sys_uselib           //86
      182    + sys_uselib,       //86
      183    + sys_sem_open,     //87
      184    + sys_sem_wait,     //88
      185    + sys_sem_post,     //89
      186    + sys_sem_unlink   //90
      187    +
178    188    };

```

```

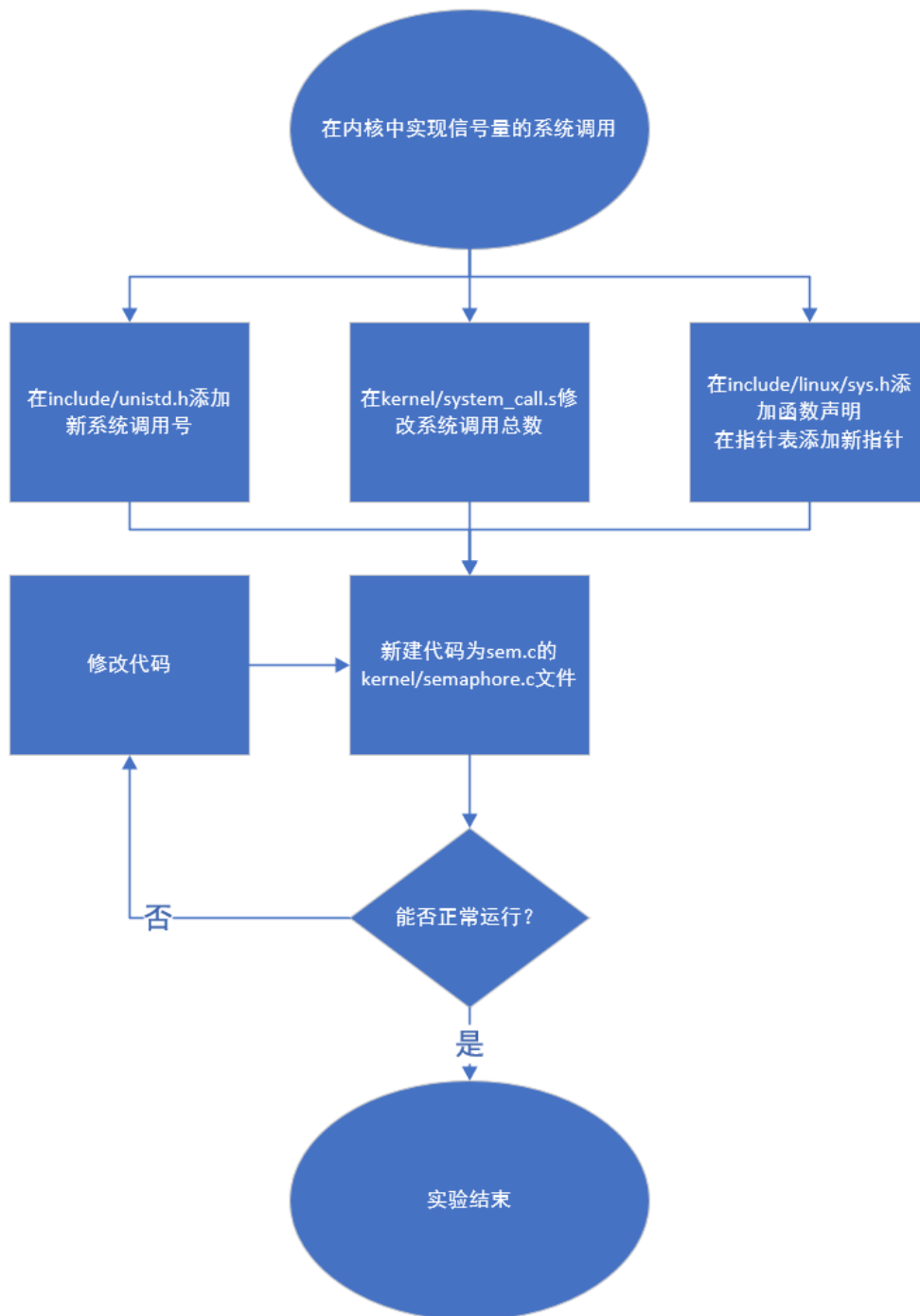
▼ include/unistd.h
...    ...    @@ -159,6 +159,11 @@
159    159    #define __NR_lstat      84
160    160    #define __NR_readlink  85
161    161    #define __NR_uselib    86
      162    + #define __NR_sem_open 87
      163    + #define __NR_sem_wait 88
      164    + #define __NR_sem_post 89
      165    + #define __NR_sem_unlink 90
      166    +
162    167    // 以下定义系统调用嵌入式汇编宏函数。
163    168    // 不带参数的系统调用宏函数。type name(void)。
164    169    // %0 - eax(__res), %1 - eax(__NR_##name)。其中name 是系统调用的名称，与 __NR_ 组合形成上面
...    ...

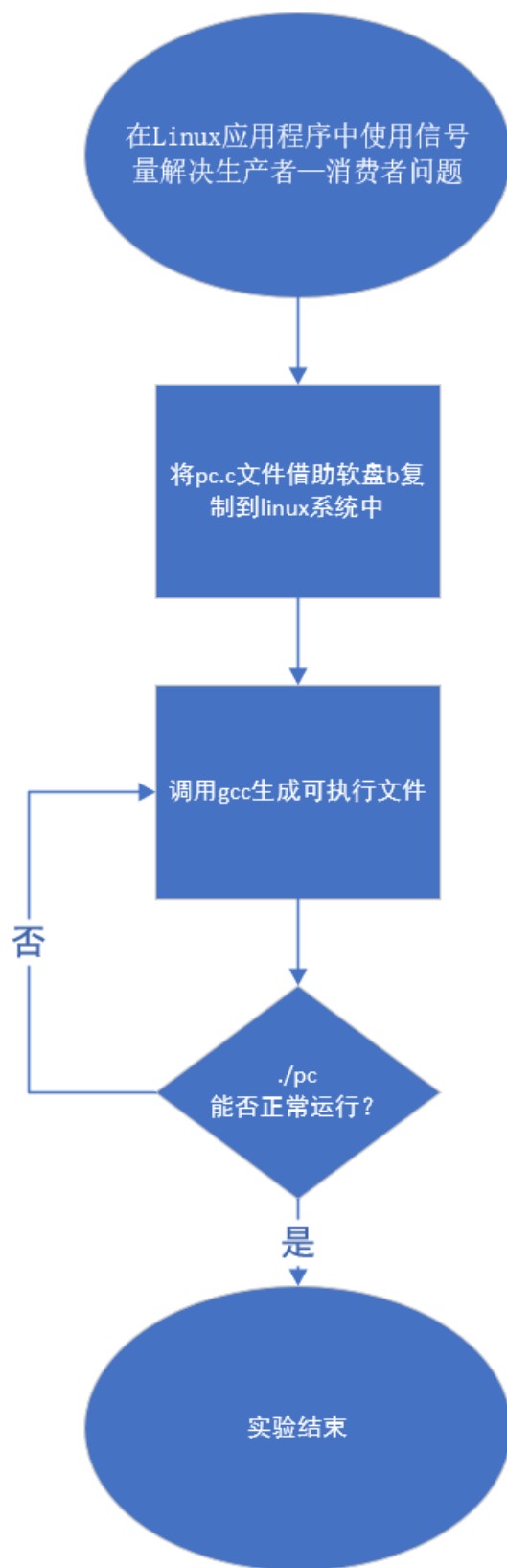
```

▼ kernel/system_call.s		
...	...	@@ -70,7 +70,7 @@ sa_mask = 4 # 信号量屏蔽码
70	70	sa_flags = 8 # 信号集。
71	71	sa_restorer = 12 # 返回恢复执行的地址位置。参见kernel/signal.c
72	72	
73		- nr_system_calls = 87 # Linux 0.11 版内核中的系统调用总数。
	73	+ nr_system_calls = 91 # Linux 0.11 版内核中的系统调用总数。
74	74	
75	75	/*
76	76	* Ok, I get parallel printer interrupts while using the floppy for some
▼ newapp/pc.c 0 → 100644		

图 24

5. 流程图





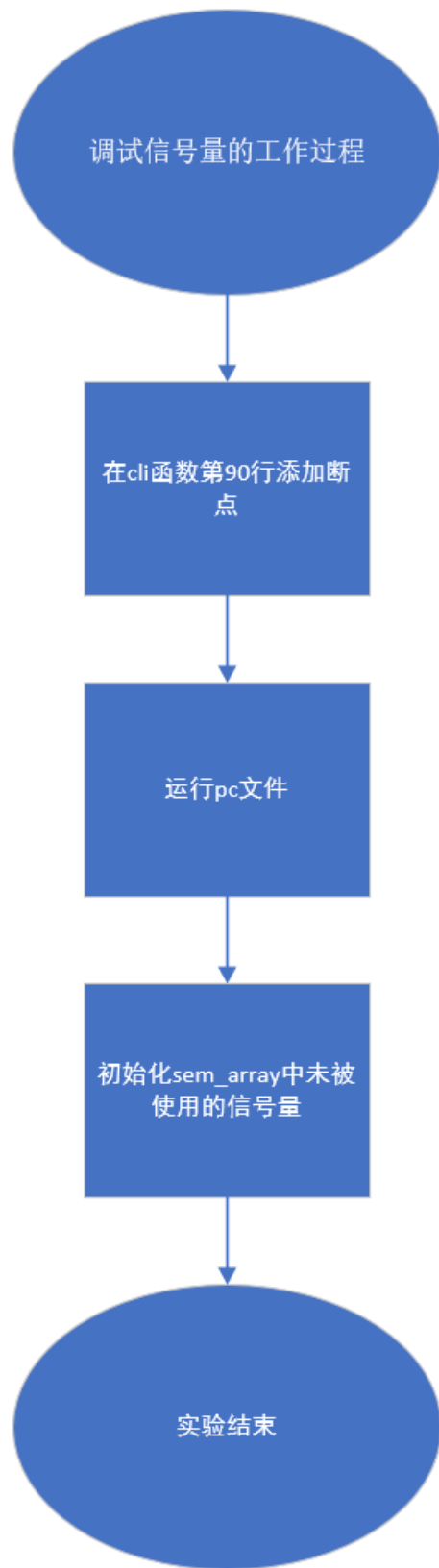




图 25

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：

<pre>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); }</pre>	<pre>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); }</pre>
--	---

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：

```
mcopbyb: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：

```
mcopbyb:linuxapp.exe producer
chmod +x producer
sync
```

从而将生产者程序复制到硬盘。

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。

第二部分 内存管理（任务十一至十六）

任务十 分配和释放物理页

1. 实验目的

- (1) 深入理解物理内存的分页管理方式。
- (2) 深入理解操作系统的段、页式内存管理。包括理解段表、二级页表，以及逻辑地址、线性地址、物理地址的映射过程。
- (3) 编程实现段、页式内存管理上的内存共享，从而深入理解操作系统的内存管理。

2. 实验内容

分配和释放物理页

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并打开。

3.2 物理内存的管理

- ①在 `init/main.c` 文件中的第 170 行添加一个断点，在 `kernel/fork.c` 文件中的第 99 行添加一个断点，按“F5”启动调试。
- ②首先会命中刚刚添加的第一个断点。
- ③首先，在 VSCode 的“View”菜单中选择“Debug Console”，会在 VSCode 的底部显示出“DEBUG CONSOLE”窗口，在该窗口底部的调试命令编辑器中输入调试命令“`-exec call pm()`”后按回车。

④然后，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“#pm”后按回车，可以看到物理内存存在刚刚完成初始化后的情况。

物理页框号	分页信息
0x0	未参与分页
0x1	未参与分页
.....
0xfe	未参与分页
0xff	未参与分页
0x100	mem_map[0]=100
0x101	mem_map[1]=100
.....
0x3fe	mem_map[766]=100
0x3ff	mem_map[767]=100
0x400	空闲页
0x401	空闲页
.....
0xffe	空闲页
0xfff	空闲页

图 26

⑤此时，物理内存刚刚完成初始化，其中 1MB 以下的物理内存未参与分页，1M 以上的物理内存存在完成分页后包含了已使用的物理页和空闲页。此时标记为已使用的物理页是为 Linux 内核空间保留的物理页，并且已经将这些物理页映射到了逻辑地址空间，确保逻辑地址与物理地址相同，这样 Linux 内核就可以直接使用物理地址访问这些物理页了。

⑥将全局变量“mem_map”添加到“WATCH”窗口查看其内容。

⑦按“F5”继续运行，会在刚刚添加的第二个断点处中断，此时通过执行断点之前的那一行代码，调用 get_free_page 函数第一次申请到了一个空闲的物理页。

⑧首先删除所有断点，然后在 kernel/exirt.c 文件中的第 30 行添加一个断点，按“F5”继续调试，会命中刚刚添加的断点，此时将鼠标移动到此行代码中传递给 free_page 的参数 p 上，记录下参数 p 的值。

⑨结束此次调试。

3.3 通过编程的方式练习分配物理页和释放物理页

①打开“学生包”，在本实验对应文件夹下找到 mem.c 文件，拖动到 VSCode 中释放，即可打开此文件。将其中的函数 physical_mem 复制到 Linux0.11 内核项目下的 mm/memory.c 文件的末尾处，并且需在 include/linux/kernel.h 中添加该函数的声明。

②添加一个系统调用号为 87 的系统调用。

③生成项目，确保没有语法错误和警告。

④按 F5 启动调试，待 Linux 0.11 完全启动后，使用 vi 编辑器新建一个 main.c 文件，其源代码如下所示：

```
#define __LIBRARY__
#include <unistd.h>
#define __NR_dump_physical_mem 87
_syscall0(int,dump_physical_mem)
int main()
{
    dump_physical_mem();
    return 0;
}
```

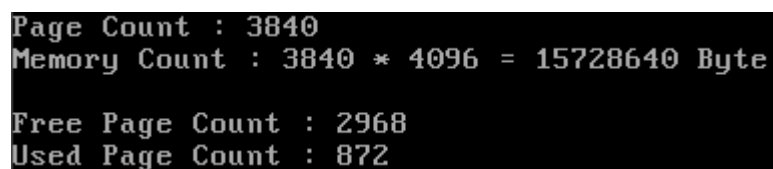
⑤保存 main.c 文件并退出 vi 编辑器后，依次执行如下命令：

```
gccmain.c -o mem
```

```
sync
```

```
mem
```

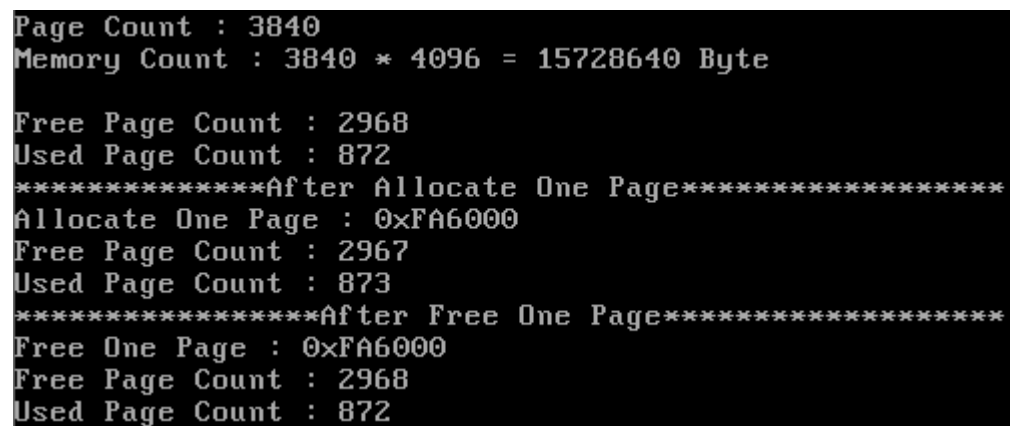
应用程序执行后打印输出的物理存储器的信息。



```
Page Count : 3840
Memory Count : 3840 * 4096 = 15728640 Byte

Free Page Count : 2968
Used Page Count : 872
```

图 27



```
Page Count : 3840
Memory Count : 3840 * 4096 = 15728640 Byte

Free Page Count : 2968
Used Page Count : 872
*****After Allocate One Page*****
Allocate One Page : 0xFA6000
Free Page Count : 2967
Used Page Count : 873
*****After Free One Page*****
Free One Page : 0xFA6000
Free Page Count : 2968
Used Page Count : 872
```

图 28

⑥对 physical_mem 函数中的源代码进行修改，运行结果如下：

⑦在 Linux 0.11 的终端使用下面的命令将刚刚编写的文件 main.c 复制到软盘 B 中。

```
mcopymain.c b:main.c
```

⑧关闭 Bochs 虚拟机。

⑨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。

⑩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 main.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。

3.4 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 8 个文件

▼ include/linux/kernel.h

...	...	@@ -18,6 +18,7 @@ int tty_write (unsigned ch, char *buf, int count);
18	18	void *malloc (unsigned int size);
19	19	// 释放指定对象占用的内存。(Lib/malloc.c)。
20	20	void free_s (void *obj, int size);
	21	+ int physical_mem();
21	22	
22	23	#define free(x) free_s((x), 0)
23	24	
...	...	

▼ include/linux/sys.h

...	...	@@ -85,6 +85,7 @@ extern int sys_symlink();
85	85	extern int sys_lstat();
86	86	extern int sys_readlink();
87	87	extern int sys_uselib();
	88	+ extern int dump_physical_mem();
88	89	// 系统调用函数指针表。用于系统调用中断处理程序(int 0x80)，作为跳转表。
89	90	// 数组元素为系统调用内核函数的函数指针，索引即系统调用号
90	91	fn_ptr sys_call_table[] = {
...	...	@@ -174,5 +175,6 @@ sys_select, //82
174	175	sys_symlink, //83
175	176	sys_lstat, //84
176	177	sys_readlink, //85
177		- sys_uselib //86
	178	+ sys_uselib, //86
	179	+ dump_physical_mem
178	180	};

图 29

5. 流程图

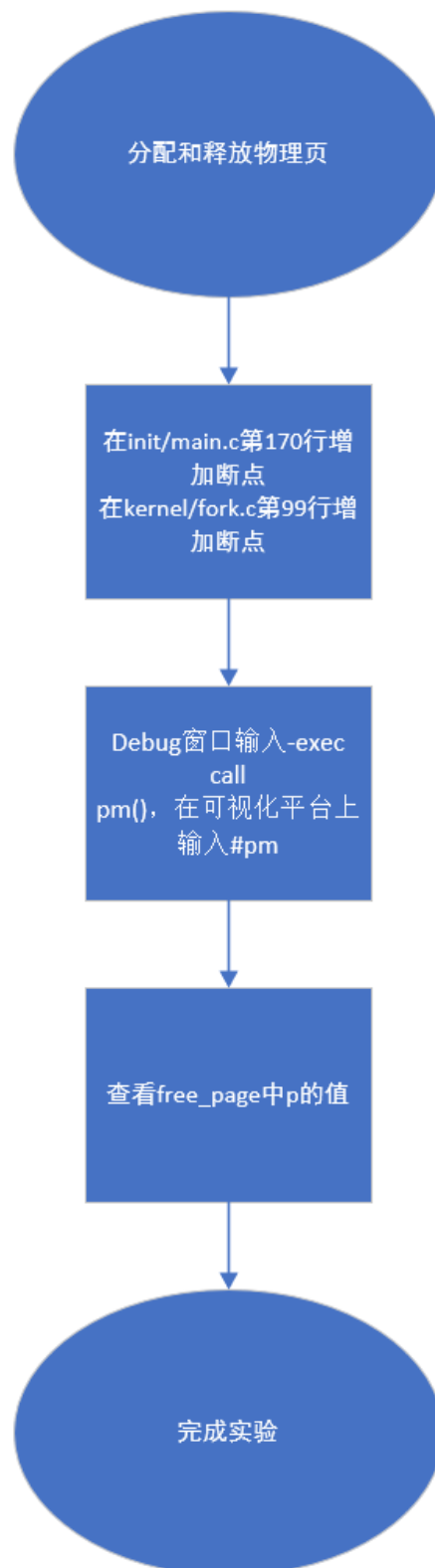


图 30

6. 实验体会

理解物理内存的分页管理方式和操作系统的段、页式内存管理非常重要。物理内存的分页管理方式是将物理内存划分为若干个大小相等的页框。虚拟内存和物理内存之间的映射关系，是通过页表来完成的。在实现页式内存管理时，需要使用段表和二级页表来完成逻辑地址、线性地址和物理地址之间的映射关系。

7. 思考与练习

无

任务十一 跟踪 Linux 应用程序中的逻辑地址、线性地址、物理地址的映射过程

1. 实验目的

- (1) 深入理解物理内存的分页管理方式。
- (2) 深入理解操作系统的段、页式内存管理。包括理解段表、二级页表，以及逻辑地址、线性地址、物理地址的映射过程。
- (3) 编程实现段、页式内存管理上的内存共享，从而深入理解操作系统的内存管理。

2. 实验内容

跟踪 Linux 应用程序中的逻辑地址、线性地址、物理地址的映射过程

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，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 跟踪 Linux 应用程序中的逻辑地址、线性地址、物理地址的映射过程

- ① 输入命令 “c” 按回车，Bochs 虚拟机会继续运行 Linux 0.11 操作系统，直到其启动完毕等待用户输入命令。
- ② 在 Linux 0.11 中使用 vi 编辑器新建 loop.c 文件，并输入之前给出的那个包含死循环的应用程序源代码。
- ③ 使用 GCC 将 loop.c 编译为 loop 可执行文件并使用 sync 保存到硬盘后，运行 loop，会打

印输出如下信息：

④读者在这里需要先结束此次调试，并关闭 Bochs 虚拟机。然后重新使用 Task 列表中的“Bochs 命令调试”启动 Bochs 虚拟机，待启动 Linux 0.11 后再运行一次 loop 应用程序，这样可以保证 loop 应用程序创建的进程 PID 为 6，并且产生的数据与下面描述的一致。

⑤点击 Bochs 的命令窗口(标题为“Bochs for Windows - Console”)的头部名称位置激活窗口，按“Ctrl+c”，Bochs 会暂停运行，等待用户输入调试命令。

⑥使用命令“u /8”，显示从当前位置开始的 8 条指令的反汇编代码

在读者自行将逻辑地址转换为线性地址之前，需要先掌握段选择符和段描述符的格式。

段选择符

⑦在 Bochs 的命令窗口中输入命令“sreg”查看 DS 寄存器的值

⑧由于 LDT 表的基址也是由一个段描述符来描述的，而且这个段描述符存储在 GDT 表中，其索引由 ldt 寄存器确定。所以，从上图中也可以得到 ldt 的值为 0x0068，换算为 16 位的二进制为 0000000001101000，使用粗体表示的就是索引值，二进制为 1101，换算为十进制为 13，表示 LDT 表的描述符在 GDT 的索引为 13（第 14 个描述符）。

⑨GDT 的起始物理地址存储在寄存器 gdtr 中，在上图中显示寄存器 gdtr 的值为 0x00005cb8。所以在 Bochs 的命令窗口中输入命令“xp /2w 0x00005cb8+13*8”，就可以查看 GDT 表中索引值为 13 的段描述符。

3.3 页目录和页表

①首先需要算出线性地址中的页目录号、页表号和页内偏移，它们分别对应了 32 位线性地址的高 10 位+中间 10 位+低 12 位，所以 0x10003004 的页目录号为 64，页表号为 3，页内偏移为 4。

②页目录表的起始物理地址由控制寄存器 CR3 指定。在 Bochs 的命令窗口中输入命令“creg”可以查看 CR3 寄存器的值，如下：

③在 Bochs 的命令窗口中输入命令“xp /w 0+64*4”，在页目录中查看页目录号为 64 的页表项，如下图：

④在 Bochs 的命令窗口中输入命令“xp /w 0x00faa000+3*4”，在页表中查看页表号为 3 的页表项，如下图：

⑤在 Bochs 的命令窗口中输入命令“xp /w 0x00fa7004”，查看从该物理地址开始的 4 个字节的值，也就是变量 i 的值，如下图：

⑥在 Bochs 的命令窗口中输入命令“setpmem 0x00fa7004 4 0”，将从物理地址 0x00fa7004 开始的 4 个字节的值都设为 0。然后再使用命令“c”让 Bochs 继续运行，可以看到应用程序退出了，说明变量 i 的值在被修改为 0 后结束了死循环。

3.4 深入研究 Linux 0.11 应用程序进程内存的管理方式

①按 F5 启动调试后，使用 vi loop.c 命令开始编辑 loop.c 文件，使用下面的源代码修改 loop 应用程序，并编译生成可执行文件，使之在开始运行的时候就调用 fork 函数创建一个子进程，这样父进程和子进程就会从同样的位置继续向后并发的运行。

```
#include <stdio.h>
int i = 0x12345678;
int main(void)
```

```

{
fork();
printf("The logical/virtual address of i is 0x%08x\n", &i);
fflush(stdout);
while(i)
    ;
return 0;
}

```

②使用 `loop` 命令运行可执行文件，观察打印输出的信息可以发现父进程和子进程中全局变量 `i` 的逻辑地址是相同的，说明父进程和子进程使用了完全相同的逻辑地址空间。

③结束调试。在 `kernel/fork.c` 文件的第 171 行添加一个条件断点，条件设置为 “`p->pid == 7`”，这样当 `loop` 程序创建子进程完毕后就会命中此断点。

④按 F5 启动调试，在终端运行 `loop` 应用程序后，会命中刚刚添加的条件断点。

⑤首先，在 VSCode 的 “View” 菜单中选择 “Debug Console”，会在 VSCode 的底部显示出 “DEBUG CONSOLE” 窗口，在该窗口底部的调试命令编辑器中输入调试命令 “`-exec call showgdt()`” 后按回车。

⑥然后，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 “`#gdt`” 后按回车，可以看到全局描述符表的可视化数据。

⑦在全局描述符表的可视化数据中可以看到（注意，左侧是全局描述符表 GDT，但是为了方便读者查看，在右侧同时显示了全局描述符表项映射的任务状态段 TSS 和局部描述符表 LDT），进程 6（父进程）的局部描述符 LDT 表中代码段描述符和数据段描述符所使用的段基址均为 `0x10000000`，而进程 7（子进程）的局部描述符 LDT 表中代码段描述符和数据段描述符所使用的段基址均为 `0x14000000`，也就是说，虽然父进程和子进程使用了相同的逻辑地址（通过打印输出全局变量 `i` 的逻辑地址确定的），但是当两个进程中的逻辑地址转换为线性地址时，还需要再加上各自的段基址，这样父进程和子进程中的相同的逻辑地址就会映射到不同的线性地址空间，从而完成了分段隔离。

3.5 共享物理页

①仍然使用上面练习中的 `loop` 程序和条件断点。

②按 F5 启动调试，在终端运行 `loop` 应用程序后，会命中条件断点。接下来读者需要按照下面的步骤查看二级页表的可视化数据。

③首先，在 VSCode 的 “View” 菜单中选择 “Debug Console”，会在 VSCode 的底部显示出 “DEBUG CONSOLE” 窗口，在该窗口底部的调试命令编辑器中输入调试命令 “`-exec call vm2(0x40)`” 后按回车。

④然后，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。

⑤在 “DEBUG CONSOLE” 窗口底部的调试命令编辑器中输入调试命令 “`-exec call vm2(0x50)`” 后按回车，然后在右侧可视化视图顶部的编辑框中输入命令 “`#vm2`” 后按回车，可以看到进程 7 的完整的二级页表映射关系。

3.6 写时复制(Copy on write)

- ①在 kernel/sched.c 文件的第 271 行添加一个条件断点，条件设置为 “task[next]->pid == 7”。
- ②按 F5 继续调试，会命中刚刚添加的条件断点
- ③此时，子进程一定是已经运行过一段时间后才命中的断点。

3.7 按需加载(Load on demand)

- ①结束调试，关闭虚拟机。
- ②保留在 kernel/sched.c 文件的第 271 行添加的条件断点，删除其它所有断点。
- ③按 F5 启动调试。在终端运行 loop 应用程序后会命中条件断点。
- ④在“DEBUG CONSOLE”窗口底部的调试命令编辑器中输入调试命令“-exec call vm2(0x50)”后按回车，然后在右侧可视化视图顶部的编辑框中输入命令“#vm2”后按回车。
- ⑤按 F5 继续运行，会再次命中条件断点。由于此时进程 7 已经运行过一段时间了，在“DEBUG CONSOLE”窗口底部的调试命令编辑器中输入调试命令“-exec call vm2(0x50)”后按回车。

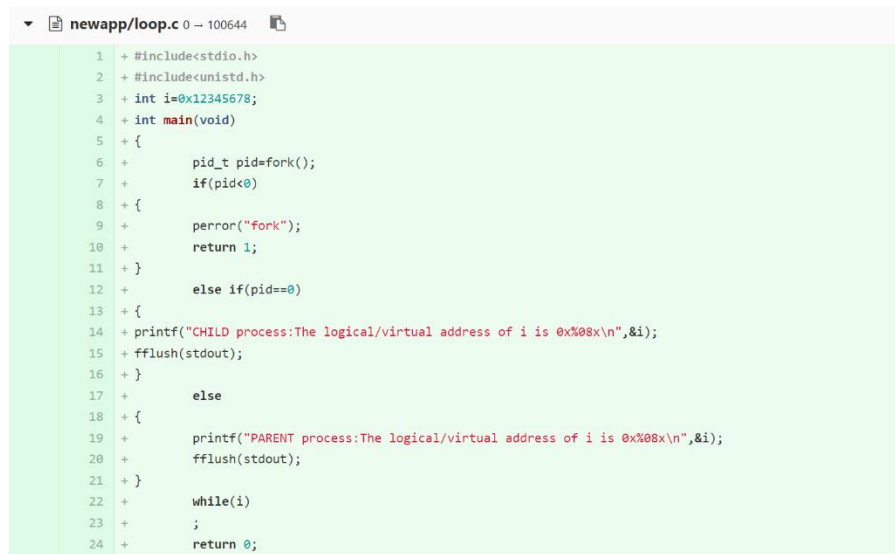
3.8 提交作业

当代码编写完成并运行成功后，为了将刚刚编写的文件提交到平台，需要按照下面的步骤将这些文件从 Linux 0.11 操作系统中复制到软盘 B，然后再复制到本地。

- ①在 Linux 0.11 的终端使用下面的命令将刚刚编写的文件 loop.c 复制到软盘 B 中。
mcopyloop.c b:loop.c
- ②关闭 Bochs 虚拟机。
- ③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。
- ④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 loop.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。
- ⑤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

共修改一个文件



```
1 + #include<stdio.h>
2 + #include<unistd.h>
3 + int i=0x12345678;
4 + int main(void)
5 + {
6 +     pid_t pid=fork();
7 +     if(pid<0)
8 +     {
9 +         perror("fork");
10 +         return 1;
11 +     }
12 +     else if(pid==0)
13 +     {
14 +         printf("CHILD process:The logical/virtual address of i is 0x%08x\n",&i);
15 +         fflush(stdout);
16 +     }
17 +     else
18 +     {
19 +         printf("PARENT process:The logical/virtual address of i is 0x%08x\n",&i);
20 +         fflush(stdout);
21 +     }
22 +     while(i)
23 +     ;
24 +     return 0;
```

图 31

5. 流程图

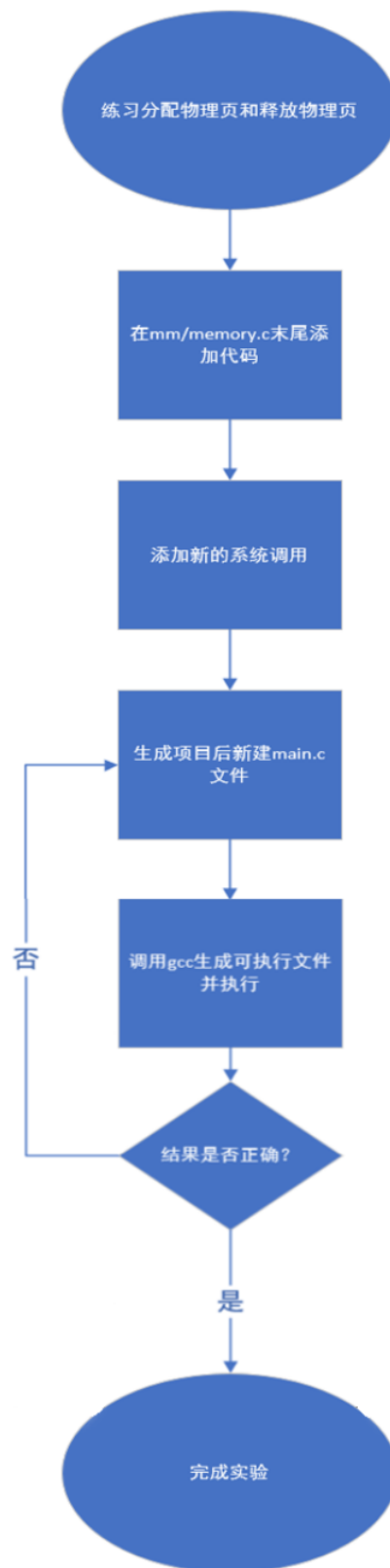


图 32

6. 实验体会

在操作系统中，每个进程都有自己的段表和二级页表。逻辑地址、线性地址和物理地址的映射过程，是通过段表和二级页表的查找过程来完成的。内存共享允许多个进程共享同一块物理内存区域。在实现内存共享时，需要特殊处理页表项，以确保多个进程共享同一块物理内存区域时，它们都能够正确地读写该内存区域。

7. 思考与练习

无

任务十二 输出应用程序进程的页目录和页表

1. 实验目的

- (1) 深入理解物理内存的分页管理方式。
- (2) 深入理解操作系统的段、页式内存管理。包括理解段表、二级页表，以及逻辑地址、线性地址、物理地址的映射过程。
- (3) 编程实现段、页式内存管理上的内存共享，从而深入理解操作系统的内存管理。

2. 实验内容

输出应用程序进程的页目录和页表

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并打开。

3.2 编写代码输出应用程序进程的页目录和页表

- ① 添加一个系统调用号为 87 的系统调用（添加系统调用的方法请参考实验四），该系统调用的内核函数 `sys_table_mapping` 可以写在 `kernel/sys.c` 文件的末尾。其源代码可以参见“学生包”中本实验对应文件夹下的 `table.c` 文件。
- ② 在 `sys_table_mapping` 函数中调用了一个名称为 `fprintk` 的函数用于向标准输出打印信息，
- ③ 在 `sys_table_mapping` 函数的开始位置还调用了 `calc_mem` 函数，此函数会计算内存中空闲页面的数量以及各个页表中映射的物理页的数量并显示。

④源代码修改完毕后生成项目，确保没有语法错误和警告。

⑤按 F5 启动调试，待 Linux011 完全启动后，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_table_mapping 87
_syscall0(int, table_mapping)
int main()
{
    table_mapping();
    return 0;
}
```

⑥保存 main.c 文件后退出 vi 编辑器，依次执行如下命令：

```
gccmain.c -o table
sync
table > a.txt
```

⑦通过命令 vi a.txt 打开 vi 编辑器查看文本内容，分析输出的结果。

3.3 提交作业

①在 Linux 0.11 的终端使用下面的命令将刚刚编写的文件 main.c 复制到软盘 B 中。

```
mcopymain.c b:main.c
```

②关闭 Bochs 虚拟机。

③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。

④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 main.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。

⑤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平台的个人项目中。

4. 运行结果

共修改 10 个文件

▼ include/linux/sys.h		
...	...	@@ -85,6 +85,7 @@ extern int sys_symlink();
85	85	extern int sys_lstat();
86	86	extern int sys_readlink();
87	87	extern int sys_uselib();
	88	+ extern int sys_table_mapping();
88	89	// 系统调用函数指针表, 用于系统调用中断处理程序(int 0x80), 作为跳转表。
89	90	// 数组元素为系统调用内核函数的函数指针, 索引即系统调用号
90	91	fn_ptr sys_call_table[] = {
...	...	@@ -174,5 +175,6 @@ sys_select, //82
174	175	sys_symlink, //83
175	176	sys_lstat, //84
176	177	sys_readlink, //85
177		- sys_uselib //86
	178	+ sys_uselib, //86
	179	+ sys_table_mapping
178	180	};
▼ include/unistd.h		
...	...	@@ -159,6 +159,7 @@
159	159	#define __NR_lstat 84
160	160	#define __NR_readlink 85
161	161	#define __NR_uselib 86
	162	+ #define __NR_max 87
162	163	// 以下定义系统调用嵌入式汇编宏函数。
163	164	// 不带参数的系统调用宏函数。type name(void)。
164	165	// %0 - eax(__res), %1 - eax(__NR_#name)。其中name 是系统调用的名称, 与 __NR_ 组合形成上面
...	...	
▼ kernel/system_call.s		
...	...	@@ -70,7 +70,7 @@ sa_mask = 4 # 信号量屏蔽码
70	70	sa_flags = 8 # 信号集。
71	71	sa_restorer = 12 # 返回恢复执行的地址位置, 参见kernel/signal.c
72	72	
73		- nr_system_calls = 87 # Linux 0.11 版内核中的系统调用总数。
	73	+ nr_system_calls = 88 # Linux 0.11 版内核中的系统调用总数。
74	74	
75	75	/*
76	76	* Ok, I get parallel printer interrupts while using the floppy for some
...	...	
▼ mm/memory.c		
...	...	@@ -802,7 +802,7 @@ void calc_mem(void)
802	802	// 扫描内存页面映射数组mem_map[], 获取空闲页面数并显示。
803	803	for(i=0 ; i<PAGING_PAGES ; i++)
804	804	if (!mem_map[i]) free++;
805		- printk("%d pages free (of %d)\n\r",free,PAGING_PAGES);
	805	+ fprintfk("%d pages free (of %d)\n\r",free,PAGING_PAGES);
806	806	// 扫描所有页目录项 (除0, 1 项), 如果页目录项有效, 则统计对应页表中有效页面数, 并显示。
807	807	for(i=2 ; i<1024 ; i++) {
808	808	if (1&pg_dir[i]) {
...	...	@@ -810,7 +810,7 @@ void calc_mem(void)
810	810	for(j=k=0 ; j<1024 ; j++)
811	811	if (pg_tbl[j]&1)
812	812	k++;
813		- printk("Pg-dir[%d] uses %d pages\n",i,k);
	813	+ fprintfk("Pg-dir[%d] uses %d pages\n",i,k);
814	814	}
815	815	}
816	816	}

图 33

5. 流程图

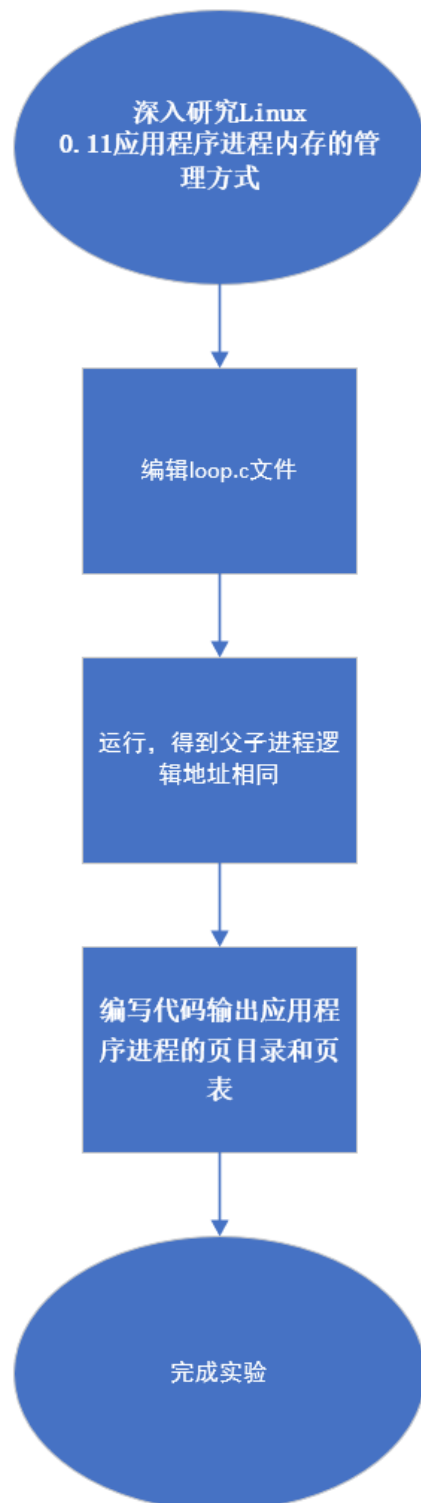


图 34

6. 实验体会

通过该实验的学习，编程实现段、页式内存管理上的内存共享，我深入理解了操作系统的内存管理机制。在实现内存共享时，我需要特殊处理页表项，确保多个进程共享同一块物理内存区域时，它们都能够正确地读写该内存区域。这个过程让我更加深入地理解了操作系统的内存管理机制，同时也更加熟悉了操作系统的开发流程。

7. 思考与练习

无

任务十三 用共享内存做缓冲区解决生产者—消费者问题

1. 实验目的

- (1) 深入理解物理内存的分页管理方式。
- (2) 深入理解操作系统的段、页式内存管理。包括理解段表、二级页表，以及逻辑地址、线性地址、物理地址的映射过程。
- (3) 编程实现段、页式内存管理上的内存共享，从而深入理解操作系统的内存管理。

2. 实验内容

用共享内存做缓冲区解决生产者—消费者问题

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并打开。

3.2 用共享内存做缓冲区解决生产者—消费者问题

- ①将 `sem.c` 文件中的四个信号量的系统调用和两个共享内存的系统调用添加到内核中。添加系统调用的方法可以参考实验四和实验七，具体步骤这里不再详细说明。
- ②生成项目，确保没有语法错误和警告。
- ③将“学生包”本实验文件夹下的 `pc.c` 文件放入软盘 B 中。
- ④按 F5 启动调试。使用 `mcop` 工具将软盘 B 中的 `pc.c` 文件复制到硬盘的当前目录。
- ⑤使用命令 `vi pc.c` 打开源代码文件，这些源代码仍然是使用文件作为生产者和消费者之

间的共享缓冲区，请读者在此基础上将其修改为使用共享内存作为缓冲区。

提示：

⑥代码修改完毕后，退出 vi 编辑器。

⑦在 Linux 中依次执行下面的命令，运行 app：

```
gcc pc.c -o app
sync
app
```

3.3 提交作业

①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
```

②关闭 Bochs 虚拟机。

③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。

④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。

⑤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，方便教师通过平台查看读者提交的作业。

4. 运行结果

共修改 6 个文件

▼ include/linux/sys.h		
...	...	@@ -85,6 +85,12 @@ extern int sys_symlink();
85	85	extern int sys_lstat();
86	86	extern int sys_readlink();
87	87	extern int sys_uselib();
	88	+ extern int sys_sem_open();
	89	+ extern int sys_sem_wait();
	90	+ extern int sys_sem_post();
	91	+ extern int sys_sem_unlink();
	92	+ extern int sys_shmget();
	93	+ extern int sys_shmat();
88	94	// 系统调用函数指针表。用于系统调用中断处理程序(int 0x80)，作为跳转表。
89	95	// 数组元素为系统调用内核函数的函数指针，索引即系统调用号
90	96	fn_ptr sys_call_table[] = {
...	...	@@ -174,5 +180,11 @@ sys_select, //82
174	180	sys_symlink, //83
175	181	sys_lstat, //84
176	182	sys_readlink, //85
177		- sys_uselib //86
	183	+ sys_uselib, //86
	184	+ sys_sem_open, //87
	185	+ sys_sem_wait, //88
	186	+ sys_sem_post, //89
	187	+ sys_sem_unlink, //90
	188	+ sys_shmget, //91
	189	+ sys_shmat //92
178	190	};

图 35

5. 流程图

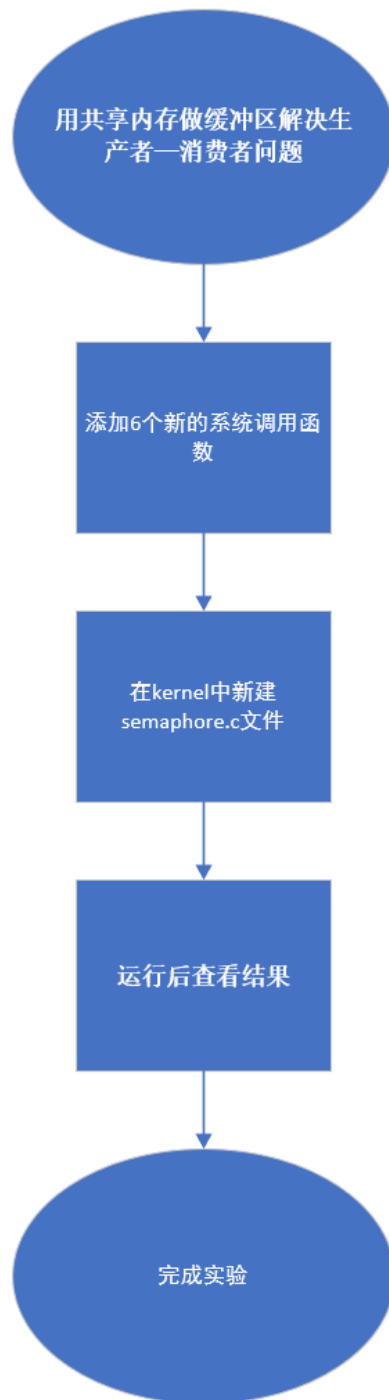


图 36

6. 实验体会

我了解了内存页式段式的格式，通过大量的练习熟悉了分配内存的基本步骤，深刻体会到了内存管理在 Linux 中的重要作用，同时熟悉了在本地编写分配内存文件后通过

软盘 b 拷贝到虚拟机中的相关操作。

7. 思考与练习

1. 请读者认真体会本实验中关于将逻辑地址映射为线性地址，线性地址映射为物理地址的过程，尝试列出映射过程中最为重要的几步。

分段和分页是将逻辑地址映射为线性地址的关键步骤。在分段过程中，通过将逻辑地址划分为段号和段内偏移量两部分，可以通过查找段表来获取该段的基地址。在分页过程中，将段内偏移量划分为页号和页内偏移量两部分，可以通过查找页表来获取该页的物理地址。最后，将页表中获取到的物理地址与页内偏移量相加，得到最终的物理地址。

2. 在本实验第 3.2 节的 loop 程序退出后，如果读者接着再运行一次，并再次进行地址跟踪，会发现有哪些异同？尝试说明原因？

在第一次运行 loop 程序时，操作系统会将程序加载到内存中并创建一个进程来执行该程序。在程序执行期间，操作系统会为该进程分配一段连续的内存空间。当程序执行完毕并退出时，操作系统会回收该进程所占用的内存空间，并将该进程从系统中删除。

如果接着再次运行 loop 程序，并进行地址跟踪，会发现程序的执行过程与第一次运行时的执行过程基本相同。因为在第二次运行 loop 程序时，操作系统会重新将程序加载到内存中，并创建一个新的进程来执行该程序。由于程序代码、数据等信息的地址在内存中是随机分配的，因此在第二次运行时，这些地址可能会与第一次运行时的地址不同，但程序的执行过程和结果应该是一致的。

3. 使用下面的代码在 Linux 0.11 内核中写一个系统调用函数，并在 Linux 0.11 的应用程序中调用此函数，然后仿照本实验第 3.2 节的内容跟踪变量 i 的逻辑地址、线性地址、物理地址的映射过程，最后通过将物理内存中变量 i 的值修改为 0 的方式使应用程序退出。体会 Linux 内核地址映射过程与 3.2 节中 Linux 应用程序地址映射过程的异同。

```
volatile int i = 0x12345678;
void sys_testg()
{
    printk("The logical/virtual address of i is 0x%08x\n", &i);
    while(i)
        ;
}
```

注意，在定义变量 i 时使用“volatile”关键字目的是防止编译器对其进行优化，否则 i 的值会被拷贝到寄存器中，而通过 Bochs 调试命令修改的是内存中 i 的值，不是寄存器中的值，就会导致程序无法退出死循环。

内核空间 and 用户空间的划分不同。在 Linux 内核中，内核空间和用户空间是分开的，内核空间的地址范围是 0xC0000000 到 0xFFFFFFFF，用户空间的地址范围是 0x00000000 到 0xBFFFFFFF。在 3.2 节中，内核空间和用户空间是共享的，它们的地址范围都是 0x00000000 到 0xFFFFFFFF。

内核代码和数据的映射方式不同。在 Linux 内核中，内核代码和数据是直接映射到物理

地址空间中的，在 3.2 节中，内核代码和数据是通过段机制映射到逻辑地址空间中的。

地址空间的划分方式不同。在 Linux 内核中，地址空间是通过页表机制划分的，每个进程都有自己的页表，用于将逻辑地址映射到物理地址。在 3.2 节中，地址空间是通过段机制划分的，每个进程都有自己的段描述符表，用于将逻辑地址映射到物理地址。

4. 将上一个练习中的变量 `i` 修改为 `sys_testg` 函数内的局部变量，仍然通过将物理内存中变量 `i` 的值修改为 0 的方式使应用程序退出。请读者结合 Linux 0.11 内核的逻辑地址空间的内存布局，体会内核中的全局变量与局部变量在逻辑地址空间中的位置有什么不同。

在 Linux 0.11 内核中，全局变量和局部变量在逻辑地址空间中的位置是不同的。全局变量存储在内核数据段中，而局部变量存储在内核栈中。内核数据段在逻辑地址空间中的位置是固定的，通常是从 `0x00001000` 开始的一段连续的地址空间。而内核栈的位置是动态的，它的大小是在内核初始化时确定的，通常是从高地址向低地址扩展。

5. 在本实验 3.4 中遇到了共享的物理页被多个进程重复释放，导致操作系统报告错误并中断运行的问题，暂时是通过在申请共享的物理页时增加物理页在 `mem_map` 中的引用计数来解决的。请读者换一个思路，通过实现一个简化版本的关闭共享内存的系统调用函数 `shmdt` 来解决此问题，其函数原型可以为：

```
void shmdt(int key, const void* startaddr)
```

参数 `key` 是共享内存的键值，即为共享内存数组的下标；参数 `startaddr` 是共享内存的逻辑地址。在每个生产者进程和消费者进程退出前，都必须调用此函数关闭共享内存，最终确保无论是一个生产者与一个消费者同步运行，还是一个生产者与多个消费者同步运行都不会再出现应用程序退出时操作系统报告错误的情况。

提示：

1) 读者需要实现一个将共享的物理页从当前进程的二级页表映射中移除的内核函数，其原型可以为：

```
void cancel_mapping(const void* linearaddr)
```

参数 `linearaddr` 是需要取消映射的物理页的线性地址，根据此线性地址找到对应的页表项，然后将页表项置为 0 即可。

2) 之前的源代码中只是用全局的 `vector` 数组保存了共享内存的物理页的基址，现在需要为共享内存定义一个结构体，其中除了保存物理页的基址外，还需要保存共享内存的引用计数（可以参考信号量的引用计数），再使用此结构体定义一个全局的共享内存数组。当进程调用 `shmdt` 函数关闭共享内存时，首先将共享内存的引用计数减 1，然后调用 `cancel_mapping` 函数将共享的物理页从二级页表映射中移除。由于 `shmdt` 的第二个参数 `startaddr` 是共享内存的逻辑地址，需要为其加上 `current->start_code` 转换为线性地址后，才能作为 `cancel_mapping` 函数的参数。最后，判断如果共享页的引用计数的值大于 0，说明仍然有其他进程在使用此共享内存，就结束（不释放物理页）；否则，需要调用 `free_page` 函数释放物理页（只释放这一次）。

先定义一个共享内存的结构体，包含物理页的基址和引用计数两个成员变量。再定义一个全局的共享内存数组，用于保存所有的共享内存，在申请共享内存时，需要遍历这个数组，找到一个空闲的位置，将物理页的基址和引用计数保存到数组中。

6. 假设一台使用 x86 处理器的物理裸机有 4GB 的物理内存，如果读者需要设计一个操作

系统，要求在操作系统初始化的时候使用全局描述符表将代码段和数据段的段基址设定为 0，段界限为 4GB，然后将所有物理内存使用二级页表映射到线性地址空间，并且确保内存的物理地址与线性地址一致。请读者说明在完成初始化后，全局描述符表中有几个描述符，各个描述符的值是多少？CS、DS 寄存器的值是多少？页目录的物理页框号是多少？页目录中 1024 个页表项的值可能是多少？第一个页表中 1024 个页表项的值可能是多少？页目录和页表一共占用了多少个物理页？

代码段：基址为 0，段界限为 0xFFFF FFFF，访问权限为可执行、非一致性代码段，DPL 为 0。

数据段：基址为 0，段界限为 0xFFFF FFFF，访问权限为可读写向上扩展的数据段，DPL 为 0。

CS 寄存器的值为 0，DS 寄存器的值为 8。

每个页目录和页表都有 1024 个页表项，所以页目录中 1024 个页表项的值可能是：

指向页表的物理地址，访问权限为可读写，存在位为 0（表示该页表不存在）或 1（表示该页表存在）。

第一个页表中 1024 个页表项的值可能是：

指向物理页的物理地址，访问权限为可读写，存在位为 1（表示该物理页已经被映射）。

所以页目录和页表一共占用了 $1 + 1024 = 1025$ 个物理页。

任务十四 页面置换算法

1. 实验目的

掌握 OPT、FIFO、LRU、LFU、Clock 等页面置换算法；

掌握可用空间表及分配方法；

掌握边界标识法以及伙伴系统的内存分配方法和回收方法。

2. 实验内容

页面置换算法

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（Windows 控制台程序），然后使用 VSCode 将个人项目克隆到本地磁盘中并打开。

3.2 查看最佳页面置换算法 (OPT) 和先进先出页面置换算法 (FIFO) 的执行过程

最佳页面置换算法

当需要换出一个页面时，淘汰那个以后不再需要的或最远的将来才会用到的页面。

先进先出页面置换算法

当需要换出一个页面时，淘汰那个最先调入主存的页面，或者说在主存中驻留时间最长的那一页。

- ①运行 task “生成项目 (make)” 生成应用程序项目，确保没有语法上的错误。
- ②按 Ctrl+Shift+C，在项目所在目录启动控制台 CMD 窗口。
- ③输入命令 `chcp 65001` 调整编码格式，防止输出中文时出现乱码。Win7 用户还需要在控制台窗口中右键点击窗口名称，选择“设置”—“字体”tab，设置字体为“Lucida Console”。
- ④输入可执行文件 exe 的名字后按下回车运行程序，查看 OPT 和 FIFO 的执行过程，其中，“#”表示未引用的内存块。执行结果如下图所示：

```

*****最佳页面置换算法:*****
页面引用串为: 7 0 1 2 0 3 0 4 2 3 0 1 1 7 0 1 0 3
访问页面7后, 内存中的页面号为: 7 # # # #
访问页面0后, 内存中的页面号为: 7 0 # # #
访问页面1后, 内存中的页面号为: 7 0 1 # #
访问页面2后, 内存中的页面号为: 7 0 1 2 #
访问页面0后, 内存中的页面号为: 7 0 1 2 #
访问页面3后, 内存中的页面号为: 7 0 1 2 3
访问页面0后, 内存中的页面号为: 7 0 1 2 3
访问页面4后, 内存中的页面号为: 4 0 1 2 3 淘汰页面号为:7
访问页面2后, 内存中的页面号为: 4 0 1 2 3
访问页面3后, 内存中的页面号为: 4 0 1 2 3
访问页面0后, 内存中的页面号为: 4 0 1 2 3
访问页面1后, 内存中的页面号为: 4 0 1 2 3
访问页面1后, 内存中的页面号为: 4 0 1 2 3
访问页面7后, 内存中的页面号为: 7 0 1 2 3 淘汰页面号为:4
访问页面0后, 内存中的页面号为: 7 0 1 2 3
访问页面1后, 内存中的页面号为: 7 0 1 2 3
访问页面0后, 内存中的页面号为: 7 0 1 2 3
访问页面3后, 内存中的页面号为: 7 0 1 2 3
缺页次数为: 7
OPT缺页率为: 0.389
*****先进先出页面置换算法:*****
页面引用串为: 7 0 1 2 0 3 0 4 2 3 0 1 1 7 0 1 0 3
访问页面7后, 内存中的页面号为: 7 # # # #
访问页面0后, 内存中的页面号为: 7 0 # # #
访问页面1后, 内存中的页面号为: 7 0 1 # #
访问页面2后, 内存中的页面号为: 7 0 1 2 #
访问页面0后, 内存中的页面号为: 7 0 1 2 #
访问页面3后, 内存中的页面号为: 7 0 1 2 3
访问页面0后, 内存中的页面号为: 7 0 1 2 3
访问页面4后, 内存中的页面号为: 4 0 1 2 3 淘汰页面号为:7
访问页面2后, 内存中的页面号为: 4 0 1 2 3
访问页面3后, 内存中的页面号为: 4 0 1 2 3
访问页面0后, 内存中的页面号为: 4 0 1 2 3
访问页面1后, 内存中的页面号为: 4 0 1 2 3
访问页面1后, 内存中的页面号为: 4 0 1 2 3
访问页面7后, 内存中的页面号为: 4 7 1 2 3 淘汰页面号为:0
访问页面0后, 内存中的页面号为: 4 7 0 2 3 淘汰页面号为:1
访问页面1后, 内存中的页面号为: 4 7 0 1 3 淘汰页面号为:2
访问页面0后, 内存中的页面号为: 4 7 0 1 3
访问页面3后, 内存中的页面号为: 4 7 0 1 3
缺页次数为: 9
FIFO缺页率为: 0.500

```

图 37

3.3 完成最近最久未使用页面置换算法(LRU)

最近最久未使用页面置换算法

当需要换出一个页面时, 淘汰那个在最近一段时间里较久未被访问的页面。它是根据程序执行时所具有的局部性来考虑的, 即那些刚被使用过的页面可能马上还要被使用, 而那些在较长时间内未被使用的页面一般可能不会马上使用。

仔细阅读 main.c 中的源代码, 并仿照已经实现的 OPT 和 FIFO 算法来实现最近最久未使用页面置换算法(LRU)。正确实现 LRU 算法后的执行结果应该如下图所示。

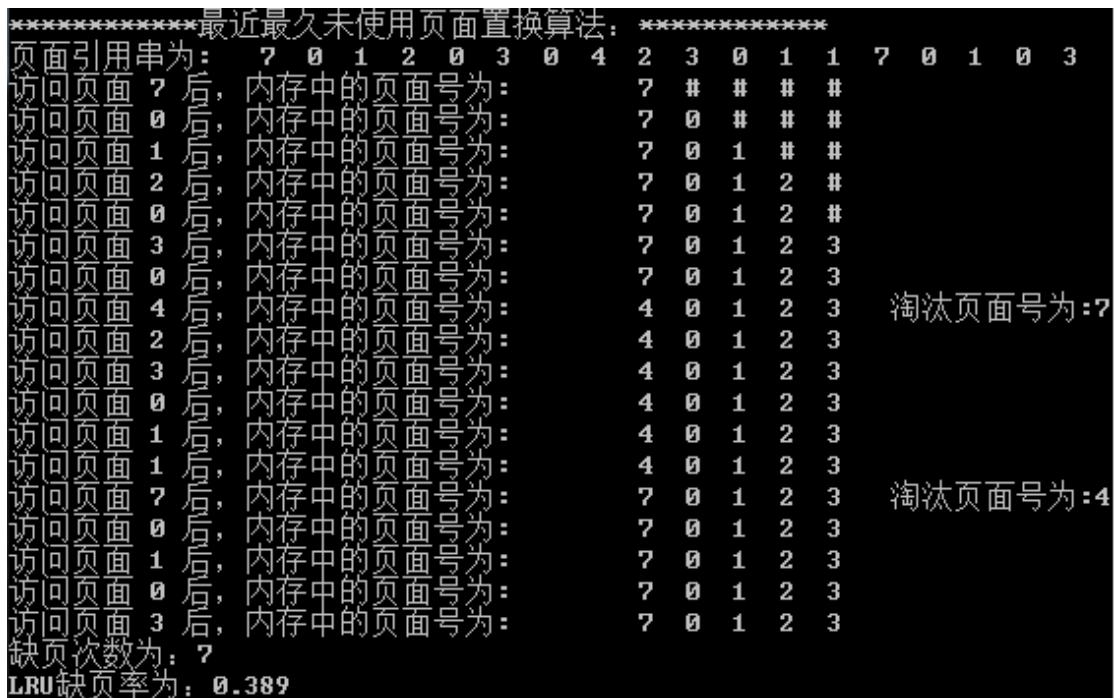


图 38

3.4 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，方便教师通过平台查看读者提交的作业。

4. 运行结果

共修改一个文件

```

... @@ -196,11 +196,26 @@ void Fifo(int *BlockOfMemory, int *PageNumOfRef, int BlockCount, int PageNumRefCoun
196 196     printf("FIFO缺页率为: %.3f\n", (float)MissCount / PageNumRefCount);
197 197 }
198 198
199 199 //3 为页表中的缺号
200 200 //k 为引用串中的页号
201 201 + int DistanceLru(int *BlockOfMemory, int *PageNumOfRef, int j, int i, int PageNumRefCount)
202 202 + {
203 203     int k, index;
204 204     for(int k = 0; k < i; k++)
205 205         if(BlockOfMemory[j] == PageNumOfRef[k])
206 206             index = k;
207 207     return index;
208 208 + }
209 209 +
210 210 //最近最久未使用页面置换算法
211 211 void Lru(int *BlockOfMemory, int *PageNumOfRef, int BlockCount, int PageNumRefCount)
212 212 {
213 213     int i;
214 214     int i, k;
215 215     int MinIndex1, MinIndex2;
216 216     int MissCount = 0;
217 217     int ReplaceIndex = 0;
218 218     int EmptyBlockCount = BlockCount;
219 219
220 220     printf("*****最近最久未使用页面置换算法: *****\n");
221 221
222 222     @@ -213,9 +228,33 @@ void Lru(int *BlockOfMemory, int *PageNumOfRef, int BlockCount, int PageNumRefCo
223 223     {
224 224         MissCount++;

```

图 39

5. 流程图

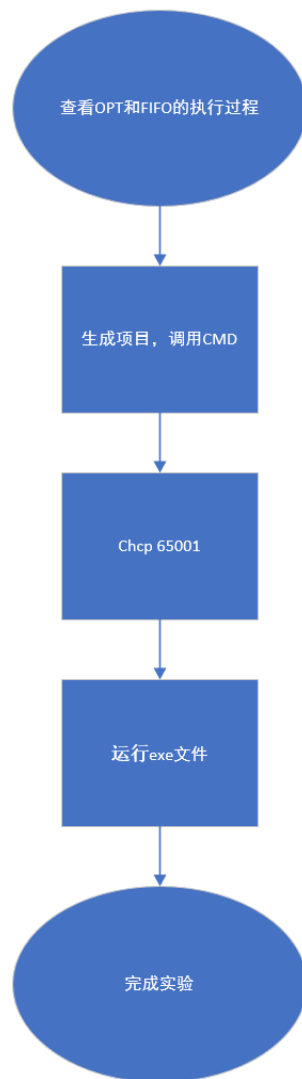


图 40

6. 实验体会

通过本次实验，我了解了内存的控制方法。通过占用回收的方式模拟内存控制，调用各类函数实现内存块的增删。

7. 思考与练习

无

任务十五 动态内存分配 - 边界标识法

1. 实验目的

掌握 OPT、FIFO、LRU、LFU、Clock 等页面置换算法；
掌握可用空间表及分配方法；
掌握边界标识法以及伙伴系统的内存分配方法和回收方法。

2. 实验内容

动态内存分配-边界标识法

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（Windows 控制台程序），然后使用 VSCode 将个人项目克隆到本地磁盘中并打开。

3.2 边界标识法的设计实现

边界标识法是操作系统中用以进行动态分区分配的一种存储管理方法。系统将所有的空闲块链接在一个双重循环链表结构的可利用空间表中；分配可按首次拟合进行，也可按最佳拟合进行。系统的特点在于：在每个内存区的头部和底部两个边界上分别设有标识，以标识该区域为占用块或空闲块，使得在回收用户释放的空闲块时易于判别在物理位置上与其相邻的内存区域是否为空闲块，以便将所有地址连续的空闲存储区组合成一个尽可能大的空闲块。

1) 分配算法

可以采用首次拟合法进行分配，即从表头指针所指节点起，在可利用空间表中进行查找，找到第一个容量不小于请求分配的存储量的空闲块时，即可进行分配。

2) 回收算法

一旦作业释放占用块，系统需立即回收以备新的请求产生时进行再分配。为了使地址相邻的空闲块结合成一个尽可能大的节点，则首先需要检查刚释放的占用块的左、右邻居是否为空闲块。若释放块的左、右邻居均为占用块，则处理最为简单，只要将此新的空闲块作为一个结点插入到可利用空闲表中即可；若只有左邻居是空闲块，则应与左邻区合并成一个结点；若只有右邻居是空闲块，则应与右邻区合并成一个结点；若左右邻居都是空闲块，则应将 3 块合起来成为一个结点留在可利用空间表中。

3) 实现

运行 task “生成项目(make)” 生成应用程序项目，确保没有语法错误。

①按 Ctrl+Shift+C，在项目所在目录启动控制台 CMD 窗口。

②输入命令 `chcp 65001` 调整编码格式，防止输出中文时出现乱码。Win7 用户还需要在控制台窗口中右键点击窗口名称，选择“设置”—》“字体” tab，设置字体为“Lucida Console”。

③输入可执行文件 `exe` 的名字后按下回车运行程序，查看运行结果。其提供的功能项如下图所示：

```
选择功能项：<0-退出  1-分配主存  2-回收主存  3-显示主存>:
```

图 41

程序启动后会停留在此界面等待用户输入功能项序号 0-3 中的一个。系统会根据用户输入的序号执行相应的功能，然后继续停留在此界面等待用户的输入，直到用户按 0 退出应用程序。

例如，首先使用功能 3 查看当前内存的分配情况，然后使用功能 1，输入内存长度 20 来分配内存，接着分配长度为 30 和 500 的内存，然后使用功能 3 查看内存的分配情况，如果已分配的内存达到内存的最大值时，再次分配内存时，系统会提示“无法继续分配内存”。当然还可以使用功能 2 来回收内存（注意：在释放内存时，首地址的输入使用十六进制）。如果要回收的内存左右有空闲区，就会与之合并，否则就作为一个结点插入到可利用空间表中，读者可在使用这些功能的同时，加深对程序的理解。

```
选择功能项：<0-退出  1-分配内存  2-回收内存  3-显示内存>: 3
空间首地址  空间大小  块标志<0:空闲,1:占用>  前驱地址  后继地址
0x395b30      1000              0      0x395b30  0x395b30
选择功能项：<0-退出  1-分配内存  2-回收内存  3-显示内存>: 1
所需内存长度: 20
选择功能项：<0-退出  1-分配内存  2-回收内存  3-显示内存>: 1
所需内存长度: 30
选择功能项：<0-退出  1-分配内存  2-回收内存  3-显示内存>: 1
所需内存长度: 500
选择功能项：<0-退出  1-分配内存  2-回收内存  3-显示内存>: 3
空间首地址  空间大小  块标志<0:空闲,1:占用>  前驱地址  后继地址
0x395b30      450              0      0x395b30  0x395b30
0x399870       20              1
0x399690       30              1
0x397750      500              1
```

图 42

```
选择功能项：<0-退出  1-分配内存  2-回收内存  3-显示内存>: 2
输入要回收分区的首地址: 0x399870
选择功能项：<0-退出  1-分配内存  2-回收内存  3-显示内存>: 3
空间首地址  空间大小  块标志<0:空闲,1:占用>  前驱地址  后继地址
0x399870       20              0      0x395b30  0x395b30
0x395b30      450              0      0x399870  0x399870
0x399690       30              1
0x397750      500              1
```

图 43

3.3 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，方便教师通过平台查看读者提交的作业。

4. 运行结果

共修改 1 个文件

▼ main.c

...

...

@@ -152,7 +152,7 @@ void reclaimBoundTag(Space * pav,Space sp){

152 152 next = sp + sp->size;//后一个空间..

153 153 nTag = next->tag;

154 154 }

155 -

155 + if(pTag!=-1&&pTag!=0) pTag=1;

156 156 if ((*pav != NULL && pTag == 1 && nTag == 1) || *pav == NULL){前后都被占用，直接插入在表头.

157 157 Space foot = FOOT_LOC(sp);

158 158 foot->tag = sp->tag = 0;

...

...

图 44

5. 流程图

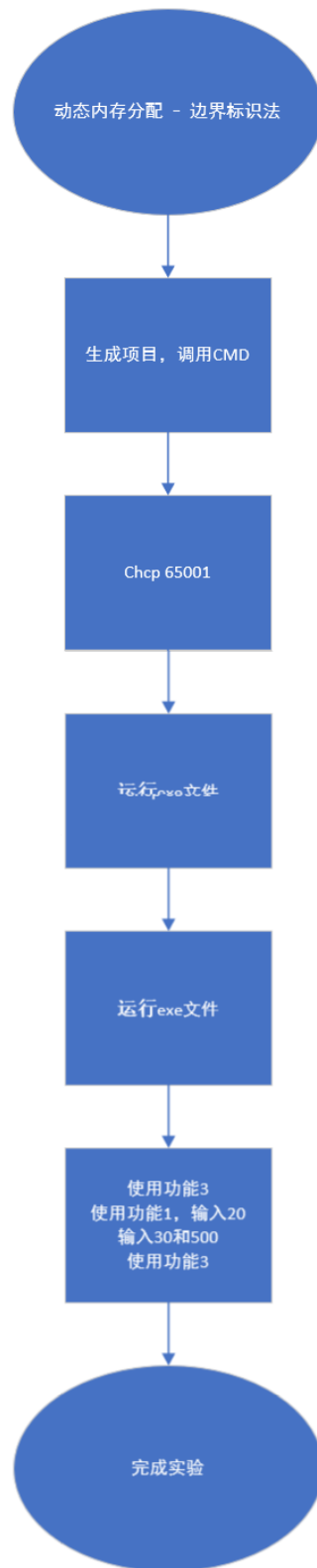


图 45

6. 实验体会

掌握不同的页面置换算法，如 OPT、FIFO、LRU、LFU、Clock 等，可以更好地优化内存管理，提高系统的性能和效率。OPT 算法是一种理论上最优的页面置换算法，但实际上很难实现。FIFO 算法是一种简单而常用的页面置换算法，但是它没有考虑页面的访问频率和重要性，可能会导致性能下降。LRU 算法是一种比较常用的页面置换算法，它根据页面最近的访问时间来判断哪些页面应该被置换出去。LFU 算法是一种根据页面访问频率来判断哪些页面应该被置换出去的算法。

7. 思考与练习

任务十六 动态内存分配 -伙伴系统

1. 实验目的

掌握 OPT、FIFO、LRU、LFU、Clock 等页面置换算法；
掌握可用空间表及分配方法；
掌握边界标识法以及伙伴系统的内存分配方法和回收方法。

2. 实验内容

动态内存分配 -伙伴系统

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（Windows 控制台程序），然后使用 VSCode 将个人项目克隆到本地磁盘中并打开。

3.2 伙伴系统的设计实现

伙伴系统是操作系统中用到的另一种动态存储管理方法。它和边界标识法类似，当用户提出申请时，分配一块大小“恰当”的内存区给用户；反之，在用户释放内存区时即回收。所不同的是，在伙伴系统中，无论是占用块或空闲块，其大小均为 2 的 k 次幂（k 为某个正整数）。例如：当用户申请 n 个字节的内存区时，分配的占用块大小为 2^k 个字节（ $2^{k-1} < n \leq 2^k$ ）。由此，在可利用空间表中的空闲块大小也只能是 2 的 k 次幂。

1) 分配算法

当用户提出大小为 n 个字节的内存请求时，首先在可利用表上寻找节点大小与 n 相匹配的子表，若此子表非空，则将子表中任意一个结点分配之即可；若此子表为空，则需从

更大的非空子表中去查找，直至找到一个空闲块，则将其中一部分分配给用户，而将剩余部分插入相应的子表中。

2) 回收算法

在回收空闲块时，应首先判断其伙伴是否为空闲块，若否，则只要将释放的空闲块简单插入在相应子表中即可；若是，则需在相应子表中找到其伙伴并删除之，然后再判断合并后的空闲块的伙伴是否是空闲块。依此重复，直到归并所得空闲块的伙伴不是空闲块时，再插入到相应的子表中去。

3) 实现

选择功能项：<0-退出 1-分配主存 2-回收主存 3-显示主存>:

图 46

选择功能项：<0-退出 1-分配主存 2-回收主存 3-显示主存>:1
输入作业所需长度: 16
选择功能项：<0-退出 1-分配主存 2-回收主存 3-显示主存>:3
可利用空间:

块的大小	块的首地址	块标志<0:空闲 1:占用>	前驱地址	后继地址
32	0x3e41b8	0	0x3e41b8	0x3e41b8
64	0x3e43b8	0	0x3e43b8	0x3e43b8
128	0x3e47b8	0	0x3e47b8	0x3e47b8
256	0x3e4fb8	0	0x3e4fb8	0x3e4fb8
512	0x3e5fb8	0	0x3e5fb8	0x3e5fb8

已利用空间:

占用块块号	占用块的首地址	块大小	块标志<0:空闲 1:占用>
0	0x3e3fb8	32	1

图 47

4) 练习

编程实现伙伴系统的内存回收算法。完成算法后，可以按照下面的案例进行测试：

- a) 先分配一个大小为 20 的内存块，然后使用功能 3 可以看到刚刚分配的内存块在已利用空间中的块号为 0，接下来回收块号为 0 的内存块，会导致所有的空闲块归并为一个空闲块。此时查看内存的显示情况，在可利用空间显示了一个内存块。

选择功能项：<0-退出 1-分配主存 2-回收主存 3-显示主存>:2
输入要回收块的块号: 0
选择功能项：<0-退出 1-分配主存 2-回收主存 3-显示主存>:3
可利用空间:

块的大小	块的首地址	块标志<0:空闲 1:占用>	前驱地址	后继地址
1024	0x3e4c90	0	0x3e4c90	0x3e4c90

图 48

- b) 连续分配 5 次块大小为 20 的内存块，查看内存的显示情况，然后按照下面的顺序回收内存块，回收块号为 0 的内存块、回收块号为 2 的内存块、回收块号为 1 的内存块、回收块号为 3 的内存块、回收块号为 4 的内存块，并在每次回收之后查看内存的显示情况，完成所有的内存块回收之后，可利用空间恢复为一个内存块。

输入要回收块的块号: 2
 选择功能项: <0-退出 1-分配主存 2-回收主存 3-显示主存>:3
 可利用空间:

块的大小	块的首地址	块标志<0:空闲 1:占用>	前驱地址	后继地址
32	0x3e5090	0	0x3e5690	0x3e4c90
32	0x3e4c90	0	0x3e5090	0x3e5690
32	0x3e5690	0	0x3e4c90	0x3e5090
64	0x3e5890	0	0x3e5890	0x3e5890
256	0x3e5c90	0	0x3e5c90	0x3e5c90
512	0x3e6c90	0	0x3e6c90	0x3e6c90

已利用空间:

占用块块号	占用块的首地址	块大小	块标志<0:空闲 1:占用>
1	0x3e4e90	32	1
3	0x3e5290	32	1
4	0x3e5490	32	1

图 49

输入要回收块的块号: 1
 选择功能项: <0-退出 1-分配主存 2-回收主存 3-显示主存>:3
 可利用空间:

块的大小	块的首地址	块标志<0:空闲 1:占用>	前驱地址	后继地址
32	0x3e5090	0	0x3e5690	0x3e5690
32	0x3e5690	0	0x3e5090	0x3e5090
64	0x3e4c90	0	0x3e5890	0x3e5890
64	0x3e5890	0	0x3e4c90	0x3e4c90
256	0x3e5c90	0	0x3e5c90	0x3e5c90
512	0x3e6c90	0	0x3e6c90	0x3e6c90

已利用空间:

占用块块号	占用块的首地址	块大小	块标志<0:空闲 1:占用>
3	0x3e5290	32	1
4	0x3e5490	32	1

选择功能项: <0-退出 1-分配主存 2-回收主存 3-显示主存>:2
 输入要回收块的块号: 3
 选择功能项: <0-退出 1-分配主存 2-回收主存 3-显示主存>:3
 可利用空间:

块的大小	块的首地址	块标志<0:空闲 1:占用>	前驱地址	后继地址
32	0x3e5690	0	0x3e5690	0x3e5690
64	0x3e5890	0	0x3e5890	0x3e5890
128	0x3e4c90	0	0x3e4c90	0x3e4c90
256	0x3e5c90	0	0x3e5c90	0x3e5c90
512	0x3e6c90	0	0x3e6c90	0x3e6c90

已利用空间:

占用块块号	占用块的首地址	块大小	块标志<0:空闲 1:占用>
4	0x3e5490	32	1

选择功能项: <0-退出 1-分配主存 2-回收主存 3-显示主存>:2
 输入要回收块的块号: 4
 选择功能项: <0-退出 1-分配主存 2-回收主存 3-显示主存>:3
 可利用空间:

块的大小	块的首地址	块标志<0:空闲 1:占用>	前驱地址	后继地址
1024	0x3e4c90	0	0x3e4c90	0x3e4c90

图 50

4. 运行结果

共修改 1 个文件

main.c

```
...    ...    @@ -15,6 +15,7 @@ typedef struct _Node{
15    15        int bflag;                // 块标志, 0: 空闲, 1: 占用
16    16        int bsize;                // 块大小, 值为2的幂次k
17    17        struct _Node *rlink;    // 头部域, 指向后继节点
18    18        + int kval;
19    19    }Node;
20    20
21    21    // 可利用空间表的头结点
22    22    @@ -25,7 +26,7 @@ typedef struct HeadNode{
23    23
24    24
25    25
26    26    Node* AllocBuddy(FreeList *avail, unsigned long n);
27    27    Node* buddy(Node* n);
28    28    - void Reclaim(FreeList *pav, Node* *p);
29    29    + void Reclaim(FreeList *pav, Node* p);
30    30
31    31    void SetColor(int color);
32    32    void print_free(FreeList p);
33    33    void print_used(Node* p[]);
34    34
35    35    ...    ...    @@ -165,12 +166,82 @@ Node* buddy(Node* n)
36    36
37    37    }
38    38
39    39
40    40    // 伙伴系统的回收算法 将p所指的释放块回收可利用空间表pav中
41    41    - void Reclaim(FreeList *pav, Node* *p)
42    42    + void Reclaim(FreeList *pav, Node* p)
43    43    {
44    44
45    45
46    46
47    47
48    48    //
49    49    //      TODO: 在此添加代码
50    50    //
51    51
52    52    + Node *q = NULL;
53    53    + Node *pre;
54    54    + while (1)
```

图 51

5. 流程图

无

6. 实验体会

除了页面置换算法，还有可用空间表及分配方法、边界标识法以及伙伴系统的内存分配方法和回收方法等内存管理技术。可用空间表及分配方法可以帮助我们更好地管理内存空间，避免内存碎片的产生。边界标识法是一种简单而常用的内存分配方法，它可以将内存空间分割成大小相等的块，并使用边界标识来记录每个块的状态。伙伴系统是一种高效的内存分配方法，它可以将内存空间分割成大小相等的块，并使用二叉树来管理这些块，从而实现高效的内存分配和回收。

7. 思考与练习

1. 在 Linux011 内核项目中，打开 lib/malloc.c 文件，仔细阅读其中的源代码及注释，查看一下 linux 0.11 系统动态分配和回收内存的方法。其使用的方法与本实验介绍的边界标识法和伙伴系统法不同。请读者尝试分别用边界标识法和伙伴系统算法来修改 malloc 函数和 free 函数来修改系统的动态分配内存和回收内存的方法。



图 52

使用边界标识法来实现 malloc 和 free 函数，需要修改内存块的数据结构，以记录每个空闲块的起始位置和大小。malloc 函数则需要遍历空闲块链表，找到一个大小满足要求的空闲块，并将其分配出去。free 函数则需要将一个已分配的块插入到空闲块链表中，并合并相邻的空闲块。

使用伙伴系统算法来实现，则需要将内存划分为大小为 2 的幂次方的块，并使用一个伙伴系统来管理这些块。malloc 函数则需要在伙伴系统中找到一个大小满足要求的块，并将其分配出去。free 函数则需要将一个已分配的块插入到伙伴系统中，并合并相邻的空闲块。

2. x86 平台为实现改进型的 Clock 页面置换算法提供了硬件支持，它实现了二级映射机制，它定义的页表项的格式如下：

其中位 5 是访问标志，当处理器访问页表项映射的页面时，页表项的这个标志就会置为 1。位 6 是页面修改标志，当处理器对一个页面执行写操作时，该项就会被置为 1。

在 Linux 0.11 的内核中写一个系统调用函数来实现改进型的 Clock 页面置换算法的模拟。参考实验八的第 3.5 节，在此函数中将二级映射表打印出来，然后实现一个算法来根据页表项中的访问位和修改位计算出应该置换出的页面。在此函数中不必完全实现 Clock 页面置换算法，只要能判断出应该置换的页面即可。再编写一个 Linux 0.11 应用程序，在应用程序中调用此系统调用函数。

在系统调用表中添加一个新的系统调用号后，具体的实现如下：

```
// 系统调用号
#define __NR_CLOCK_PAGE_SWAP 333

// 页表项
typedef struct {
    unsigned int present : 1;
    unsigned int rw : 1;
    unsigned int user : 1;
    unsigned int accessed : 1;
    unsigned int dirty : 1;
    unsigned int unused : 7;
    unsigned int frame : 20;
} page_table_entry;
```



```

// 页目录项
typedef struct {
    unsigned int present : 1;
    unsigned int rw : 1;
    unsigned int user : 1;
    unsigned int unused : 9;
    unsigned int page_table_base : 20;
} page_directory_entry;

// 页目录表
page_directory_entry *page_directory = (page_directory_entry *)0x1000;

// 页表
page_table_entry *page_table = (page_table_entry *)0x2000;

// 二级映射表
unsigned char *second_level_map = (unsigned char *)0x3000;

// 系统调用函数
int sys_clock_page_swap(void) {
    int i, j;
    unsigned int address;
    unsigned char access_bit, dirty_bit;
    unsigned int page_frame_to_swap = 0;

    // 打印二级映射表
    for (i = 0; i < 1024; i++) {
        if (page_directory[i].present == 1) {
            for (j = 0; j < 1024; j++) {
                if (page_table[i * 1024 + j].present == 1) {
                    address = (i << 22) | (j << 12);
                    printf("Virtual address: %x, Physical address: %x\n",
address, address | (page_table[i * 1024 + j].frame << 12));
                }
            }
        }
    }
}

// 遍历页表，计算每个页面的访问位和修改位
for (i = 0; i < 1024; i++) {
    if (page_directory[i].present == 1) {
        for (j = 0; j < 1024; j++) {
            if (page_table[i * 1024 + j].present == 1) {
                access_bit = second_level_map[i * 1024 + j] & 0x1;
            }
        }
    }
}

```

```

        dirty_bit = (second_level_map[i * 1024 + j] >> 1) & 0x1;
        // 根据访问位和修改位计算出应该置换出的页面
        if (access_bit == 0 && dirty_bit == 0) {
            page_frame_to_swap = page_table[i * 1024 + j].frame;
            break;
        }
    }
}

// 返回应该置换出的页面的页框号
return page_frame_to_swap;
}

```

测试的应用程序实现如下：

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>

#define __NR_CLOCK_PAGE_SWAP 333

int main() {
    int page_frame_to_swap;

    // 调用系统调用函数
    page_frame_to_swap = syscall(__NR_CLOCK_PAGE_SWAP);

    printf("Page frame to swap: %d\n", page_frame_to_swap);

    return 0;
}

```

第三部分 设备管理（任务十七—十八）

任务十七 字符显示的控制

1. 实验目的

加深对操作系统设备管理基本原理的认识，掌握键盘中断、扫描码等概念；
掌握 Linux 对键盘设备和显示器终端的处理过程。

2. 实验内容

字符显示的控制

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并打开。

3.2 字符显示的控制

①将 kernel/chr_drv/keyboard.s 文件中的第 274 行注释掉。

②在 include/asm/system.h 文件的第三行前添加一个全局变量的声明：

```
extern int judge;
```

此变量是用作判断标志。初始值为0，当F12键被按下时judge被设置为1，F12键再次被按下时，judge又被设置为0。如此循环。

③在 kernel/chr_drv/tty_io.c 文件的第61行添加 judge 的定义：

```
int judge = 0;
```

④并修改 copy_to_cooked 函数，在第196行 GETCH(tty->read_q, c); 语句的后面添加如下代码：

```
if(c=='L')
{
    if(judge) judge=0;
    else judge=1;
    break;
}
```

这样，在取到字符后，如果字符为“L”，表示F12被按下（按键F12的接通码会转换为ASCII码76，即字母“L”）。

⑤修改 kernel/chr_drv/console.c 文件中的 con_write 函数，在第615行后面添加如下代码，如果字符是英文字母或者阿拉伯数字，就将之替换为“*”号：

```

if(judge)
{
    if((c>='a' && c<='z') || (c>='A' && c<='Z') || (c>='0' && c<='9'))
        c=42;
}

```

⑥生成项目，确保没有语法错误。

⑦然后按F5启动调试。

⑧执行ls命令，可以正常看到当前目录下的所有文件。

⑨按F12，此时再执行ls命令，可以看到，所有的英文字符和阿拉伯数字全都变成了“*”。再按F12，再执行ls命令，可以看到，一切又回归到正常。

⑩修改步骤3中的源代码，将字母“L”换成其他字母（F1-F12分别对应‘A’-‘L’），重新调试，试试效果。

3.3 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

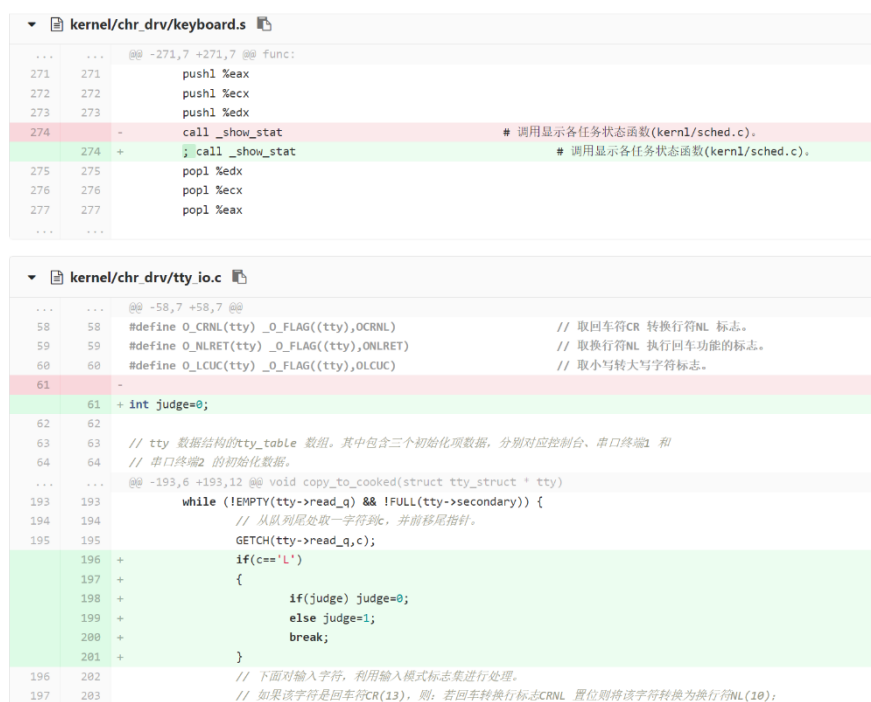
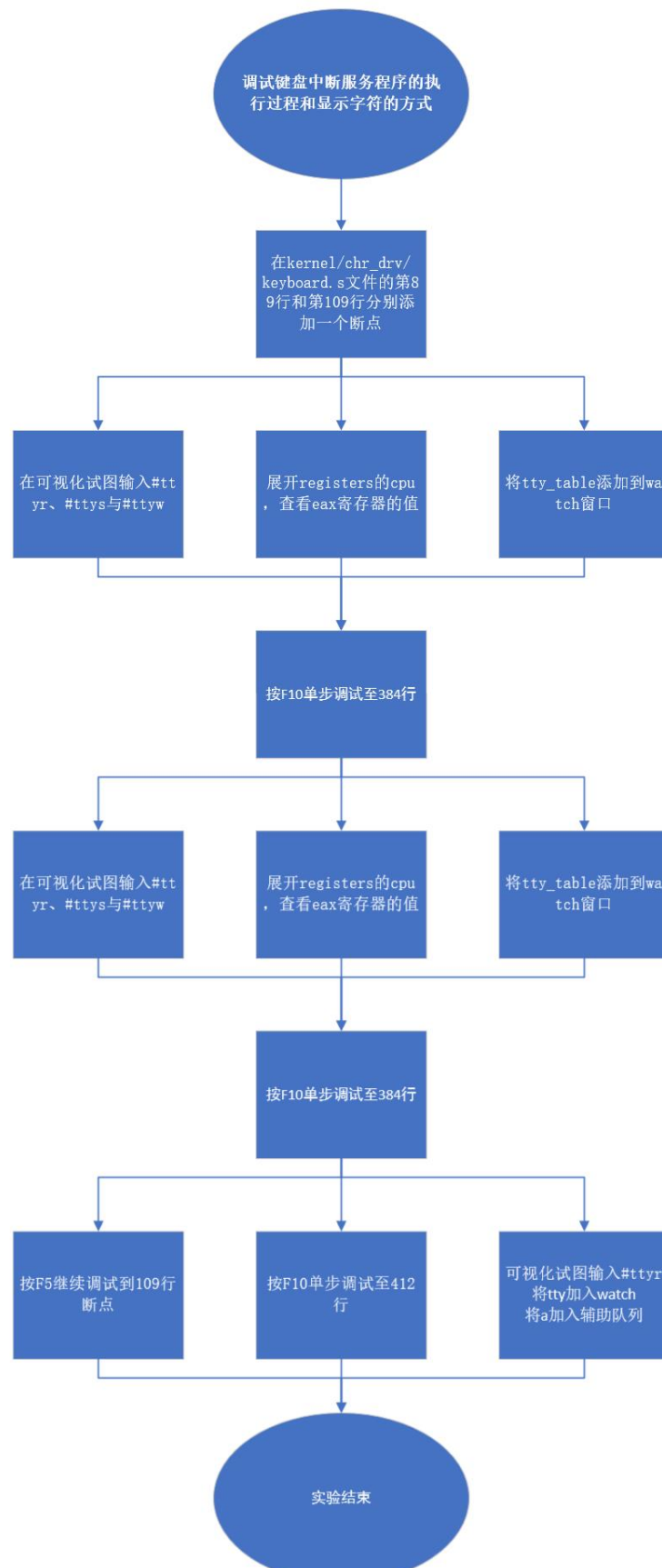
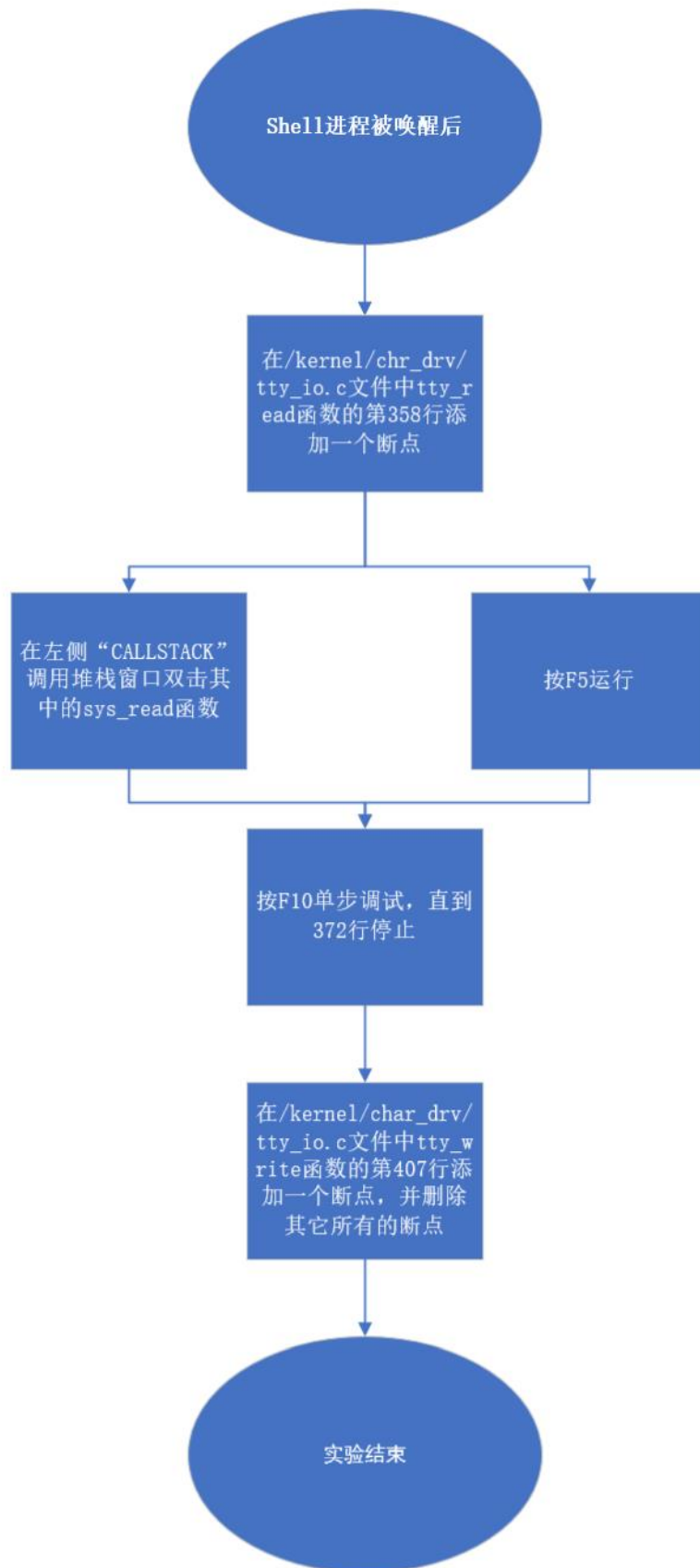


图 53

5. 流程图





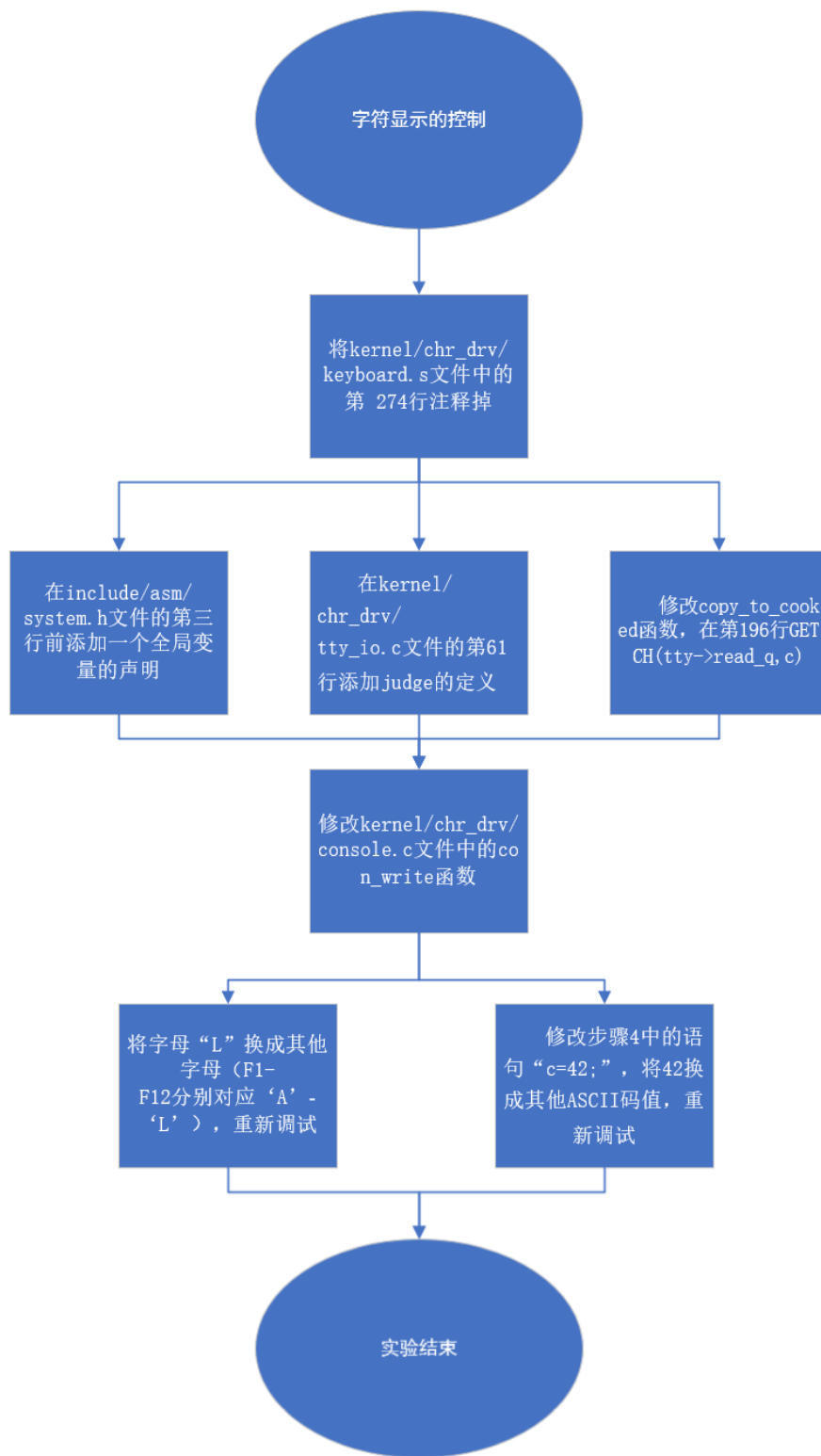


图 54

6. 实验体会

通过本次实验，我深刻认识到设备管理是操作系统的一个重要功能之一。设备管理的主要任务是对计算机硬件设备进行管理，包括对输入设备（如键盘、鼠标等）和输出设备（如显示器、打印机等）的管理。

7. 思考与练习

无

任务十八 实现贪吃蛇游戏

1. 实验目的

加深对操作系统设备管理基本原理的认识，掌握键盘中断、扫描码等概念；掌握 Linux 对键盘设备和显示器终端的处理过程。

2. 实验内容

实现贪吃蛇游戏

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并打开。

3.2 按照下面的步骤完成实验

①在文件include/asm/system.h的第三行添加一个全局变量的声明：

```
extern int fflag;
```

此变量是用作判断标志。初始值为0，表示贪吃蛇游戏未启动，当“q”键被按下时设置为1，表示贪吃蛇游戏启动。

②在文件kernel/chr_drv/tty_io.c中的第62行添加全局变量的定义：

```
int fflag = 0;
```

③并在第195行的后面(在copy_to_cooked函数中“GETCH(tty->read_q,c);”语句的后面)添加如下代码。其作用是当q键被按下时，将标志设置为1，启动贪吃蛇游戏。“w、s、a、d”键分别代表上下左右四个方向，分别将标志设置为2、3、4、5。

```
if(c=='q') //q启动游戏
```



```

{
    fflag = 1;
    PUTCH(c, tty->secondary);
    break;
}
if(c == 'w') //w上
{
    fflag = 2;
    PUTCH(c, tty->secondary);
    break;
}
if(c == 's')//s下
{
    fflag = 3;
    PUTCH(c, tty->secondary);
    break;
}
if(c == 'a')//a左
{
    fflag = 4;
    PUTCH(c, tty->secondary);
    break;
}
if(c == 'd')//d右
{
    fflag = 5;
    PUTCH(c, tty->secondary);
    break;
}
}

```

④在kernel/chr_drv/console.c文件的第598行后面(con_write函数的前面)添加如下代码。函数snake_move将蛇头字符设置在显示内存中由 pos 指定的位置，从而显示在屏幕上。snake_stop函数中使用全局变量jiffies完成一个计时循环(jiffies可参考实验六中的说明)，从而让贪吃蛇停止一定的时间。

//将蛇头字符设置在显示内存中由 pos 指定的位置，从而显示在屏幕上

```

void snake_move()
{
    __asm__ ("movb _attr,%ah\n\t"
             "movw %%ax,%l\n\t"
             ::"a" ('+'),"m" (*(short *)pos)
             );
}

```

//让贪吃蛇停留一定的时间

```

void snake_stop()

```

```

{
    int i;
    for(i = jiffies + 50; jiffies <= i;)
        ;
}

```

⑤ 在 kernel/chr_drv/console.c 文件的第 634 行后面 (con_write 函数中 “GETCH(tty->write_q, c);” 语句的后面, 第一个 switch 语句的前面) 添加如下代码, 根据标志位来启动游戏, 并且让贪吃蛇向指定的方向移动。需要说明的是, 向右移动是通过一个 while 循环来实现不停地向右移动的。

```

switch(fflag)
{
    case 0: break;
    case 1: //初始化
        csi_J(2); //清屏
        x = 0; y = 0;
        gotoxy(x, y); //将当前光标设置为屏幕左上角的坐标
        set_cursor(); //将光标移动到屏幕左上角
    case 5: //向右移动
        while(fflag == 5 || fflag == 1)
        {
            delete_line(); //删除光标所在的行
            x < video_num_columns-1 ? x++ : x; //光标坐标向右移动一位
            gotoxy(x, y); //更新光标位置
            set_cursor(); //设置显示器光标的位置
            snake_move(); //蛇向右移动
            snake_stop(); //停留
        }
        break;
    case 3: //向下移动
        while(fflag == 3)
        {
            delete_line(); //删除光标所在的行
            y < video_num_lines-1 ? y++ : y; //光标坐标向下移动一位
            gotoxy(x, y); //更新光标位置
            set_cursor(); //设置显示器光标的位置
            snake_move(); //蛇向下移动
            snake_stop(); //停留
        }
        break;
}
if(fflag != 0)
{
    continue;
}

```

- ⑥生成项目，确保没有语法错误。
- ⑦按F5启动调试。
- ⑧按“q”启动贪吃蛇游戏。游戏启动后，贪吃蛇从屏幕的左上角出现并自动向右移动。此时可以按“s”将贪吃蛇的移动方向改为向下。再按“d”即可将贪吃蛇的移动方向改为向右。

3.3 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，方便教师通过平台查看读者提交的作业。

4. 运行结果

共修改 3 个文件

正在显示 3 个修改的文件 ▾ 包含 106 行增加 和 1 行删除

include/asm/system.h

11

2

3

4

5

...

1

2

3

4

5

...

//// 切换到用户模式运行。

// 该函数利用iret 指令实现从内核模式切换到用户模式（初始任务0）。

+ extern int fflag;

#define move_to_user_mode() \

__asm__ ("movl %%esp,%%eax\n\t" \

"pushl \$0x17\n\t" \

...

kernel/chr_drv/console.c

...

595

596

597

598

599

600

601

602

603

@@ -595,6 +595,21 @@ static void restore_cur(void)

{

gotoxy(saved_x, saved_y);

}

+ void snake_move()

+ {

+ __asm__ ("movb _attr,%%ah\n\t"

+ "movw %%ax,%l\n\t"

+ :: "a" ('+'), "m" (*(short *)pos)

+);

图 55

5. 流程图

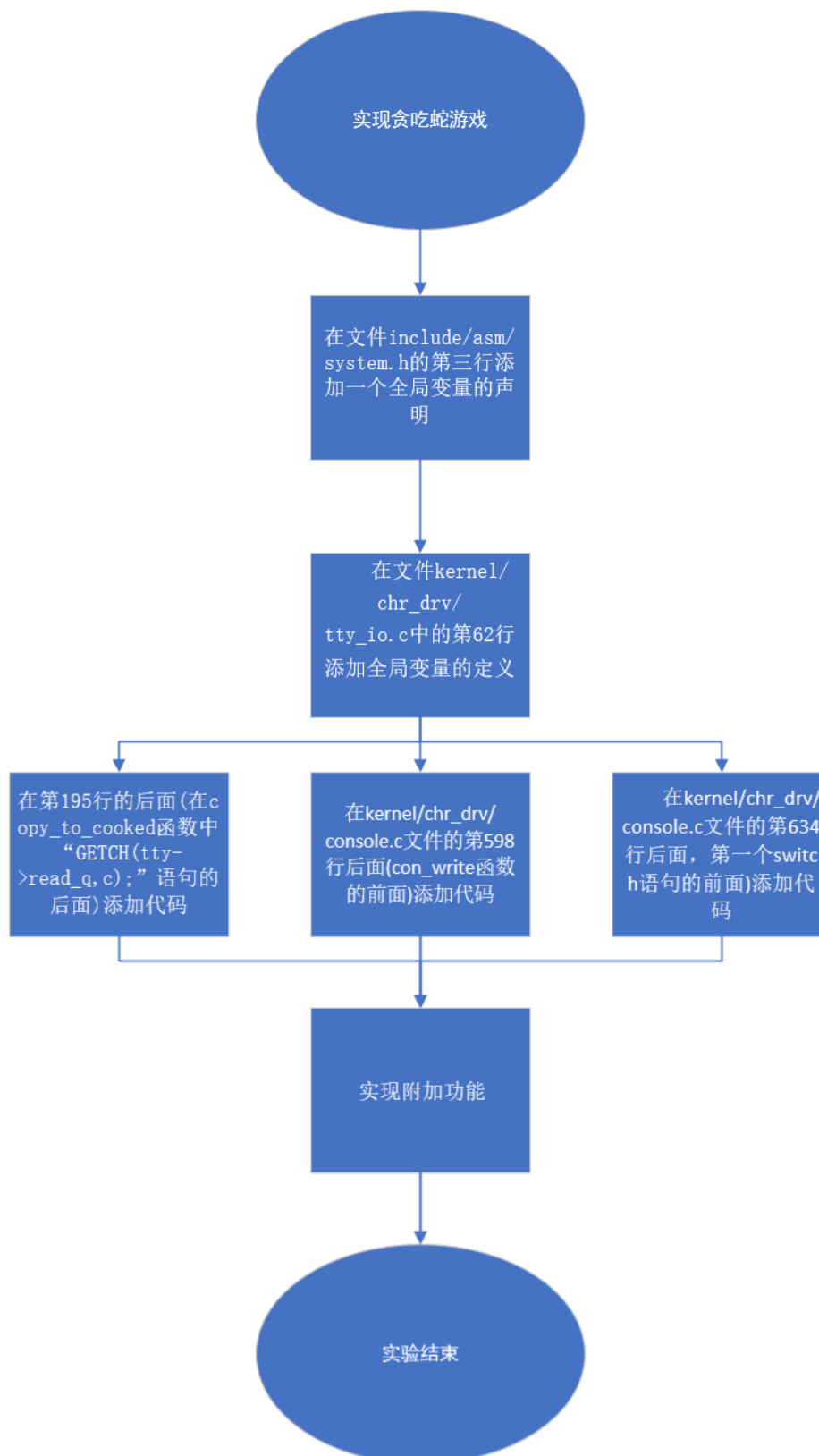


图 56

6. 实验体会

在深化我的学习体验过程中，我系统地研究了操作系统设备管理的基础机制，并通过广泛的文献阅读和实际操作，我深化了对键盘中断和扫描码等核心概念的理解。同时，我也掌握了 Linux 系统对键盘设备和显示器终端的高效管理方式。

在字符控制输入的学习上，我通过一系列练习熟练掌握了输入输出字符的基本流程，这让我更加明确了字符控制在 Linux 系统中的核心地位。这种实践性的学习让我深刻感受到，理论学习与实践操作的结合对于深化理解至关重要。

特别是在最后的项目实践中，我致力于实现贪吃蛇游戏的部分功能，通过不断的尝试和调整，我体验到了边学边做的乐趣。这一过程不仅增强了我的实践能力，也让我更加坚信，在计算机学习中，动手实践是理解和掌握知识的最佳途径。通过实际操作，我能够更直接地发现问题、解决问题，从而加深对知识点的理解和记忆。

7. 思考与练习

目前只实现了将贪吃蛇向右移动和向下移动的基础功能，请读者在充分理解之前添加的代码的基础上，为贪吃蛇游戏添加下面的功能，使其具有一定的可玩性：

- a) 添加贪吃蛇向上和向左移动的功能。
- b) 当蛇头“+”移动到屏幕的边缘时，就会在与之相反的边缘出现，继续同方向移动。
- c) 在屏幕上的某些位置出现“#”字符，当蛇头“+”与“#”相遇后，“#”消失，并且在另外一个位置再出现一个“#”。同时，蛇的尾部就多出一个“*”，作为蛇身，吃的“#”越多，蛇身就越长。
- d) 蛇身越长，贪吃蛇移动的速度越快。
- e) 当蛇头“+”撞到蛇身“*”后，结束游戏。
- f) 同时出现 2 个贪吃蛇，实现双人对战，甚至多人对战。

(a) 实现向上和向左移动的功能，仿照向下和向右移动的代码，将光标坐标和蛇的移动方向改为向上或向左。

(b) 贪吃蛇从边缘出现的功能，在移动蛇头的时候判断是否到达了屏幕的边缘，如果是，则将蛇头的坐标设置为相反边缘的坐标。

(c) 出现“#”字符的功能，在屏幕的某些位置随机生成“#”字符，并在蛇头移动的时候判断是否与“#”字符相遇，如果是，则将“#”字符删除，并在另外一个位置生成一个新的“#”字符，并在蛇的尾部添加一个“*”字符，表示蛇身变长。

(d) 蛇身越长速度越快的功能，在移动蛇头的时候，根据蛇身的长度来调整移动的速度，蛇身越长，速度越快。

(e) 蛇头撞到蛇身结束游戏的功能，在移动蛇头的时候判断是否与蛇身相遇，如果是，则游戏结束。

(f) 双人对战或多人对战的功能，在程序中添加多个贪吃蛇，并在移动蛇头的时候分别判断每个贪吃蛇的移动方向和坐标。

第四部分 文件管理（任务十九一二十）

任务十九 proc 文件系统的实现

1. 实验目的

掌握proc文件系统的实现原理。

掌握文件、目录、索引节点等概念。

2. 实验内容

编写代码，实现 proc 文件系统

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并打开。

3.2 实现 proc 文件系统

①在 MINIX 1.0 文件系统中为 proc 文件增加一个新文件类型。在 include/sys/stat.h 文件的第 25 行后面添加新文件类型的宏定义，其代码如下：

```
#define S_IFPROC 0050000
```

新文件类型宏定义的值应该在 0010000 到 0100000 之间，但后四位八进制数必须是 0，而且不能和已有的任意一个 S_IFXXX 相同

②为新文件类型添加相应的测试宏。在第 37 行后面添加测试宏，代码如下：

```
#define S_ISPROC(m) (((m)& S_IFMT) == S_IFPROC)
```

③psinfo 结点要通过 mknod 系统调用建立，所以要让它支持新的文件类型。在 fs/namei.c 文件中只需修改 mknod 函数中的一行代码即可，即将第 647 行代码修改为如下：

```
if(S_ISBLK(mode) || S_ISCHR(mode) || S_ISPROC(mode))
```

④在 init/main.c 文件的第 13 行包含头文件 stat.h：

```
#include<sys/stat.h>
```

⑤在第 42 行后面添加如下代码，定义 mkdir 和 mknod 两个系统调用函数：

```
_syscall2(int, mkdir, const char*, name, mode_t, mode)
```

```
_syscall3(int, mknod, const char*, filename, mode_t, mode, dev_t, dev)
```

⑥在第 234 行复制句柄语句的后面添加如下代码，调用 mkdir 函数创建/proc 目录，调用

mknod 函数创建 psinfo 节点:

```
mkdir("/proc", 0755);  
mknod("/proc/psinfo", S_IFPROC|0444, 0);
```

⑦找到 fs/read_write.c 文件中的 sys_read 函数, 此函数中有一系列的 if 判断语句, 再添加一个 if 语句, 代码如下:

```
if(S_ISPROC(inode->i_mode))  
    return psread(inode->i_zone[0], buf, count, &file->f_pos);
```

⑧由于这里调用了 psread 函数, 所以需要在第 32 行添加如下代码声明此函数:

```
extern int psread(int dev, char * buf, int count, off_t * f_pos);
```

⑨在VSCode的“文件资源管理器”窗口中, 右键点击“fs”文件夹节点, 在弹出的快捷菜单中选择“Reveal in File Explorer”, 打开项目所在文件夹。

⑩将“学生包”本实验对应文件夹下的proc.c文件拖动到步骤1中打开的fs文件夹下, 修改文件名称为procfs.c。

生成项目, 确保没有语法错误。

按 F5 启动调试, 待 Linux 0.11 启动以后, 输入命令: `ls -l /proc` 可以查看/proc 目录的属性信息, 如下图所示:

```
total 0  
?r--r--r--  1 root    root          0 ??? ??  ??? psinfo
```

图 57

输入命令: `cat /proc/psinfo` 即可得到当前所有进程的信息, 如下图所示:

pid	state	father	counter	start_time
0	1	-1	0	0
1	1	0	28	1
4	1	1	1	71
3	1	1	24	65
6	0	4	12	785

图 58

结束调试。

3.3 调试 proc 文件系统的工作过程

①在 fs/procfs.c 文件中 psread 函数的第一个 if 语句处 (第 70 行) 添加一个断点。

②按 F5 启动调试, 待 Linux 0.11 启动后, 输入命令“`cat /proc/psinfo`”后按回车, 会在刚刚添加的断点处中断

③按 F10 单步调试到第 81 行, 这个过程中调用了五次 addTitle 函数将标题添加到了 psbuffer 缓冲区中。接下来第 85 行开始的 for 循环, 会遍历进程控制块数组, 将有效的进程信息写入 psbuffer 缓冲区中。

⑤在 psread 函数的最后一个 for 循环语句处 (第 102 行) 添加一个断点。按 F5 继续执行, 程序就会命中此断点。此 for 循环的目的是将 psbbuffer 缓冲区内的数据逐字节的读取到用户空间中的 buf 缓冲区中。

⑥在 psread 函数最后的 return 语句处添加一个断点, 按 F5 继续调试, 程序会在此处中断。

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

```

#include <unistd.h>
#include <fcntl.h>
int main(int argc, char* argv[])
{
    char buf[513] = {'\0'};
    int nread;
    int fd = open(argv[1], O_RDONLY, 0);
    while(nread = read(fd, buf, 512))
    {
        buf[nread] = '\0';
        puts(buf);
    }
    return 0;
}

```

3.4 简化 fs/procfs.c 文件中的源代码

文件 fs/procfs.c 中 psread 函数的源代码调用了 itoa 和 addTitle 函数将内容写入 psbuffer 缓冲区,造成源代码比较复杂。读者可以使用 sprintf 函数替换掉 itoa 和 addTitle 函数,源代码就会简单很多。但是 Linux 0.11 没有实现 sprintf 函数,读者可以参考 init/main.c 文件中第 200 行的 printf 函数,在 fs/procfs.c 文件中实现一个 sprintf 函数。

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

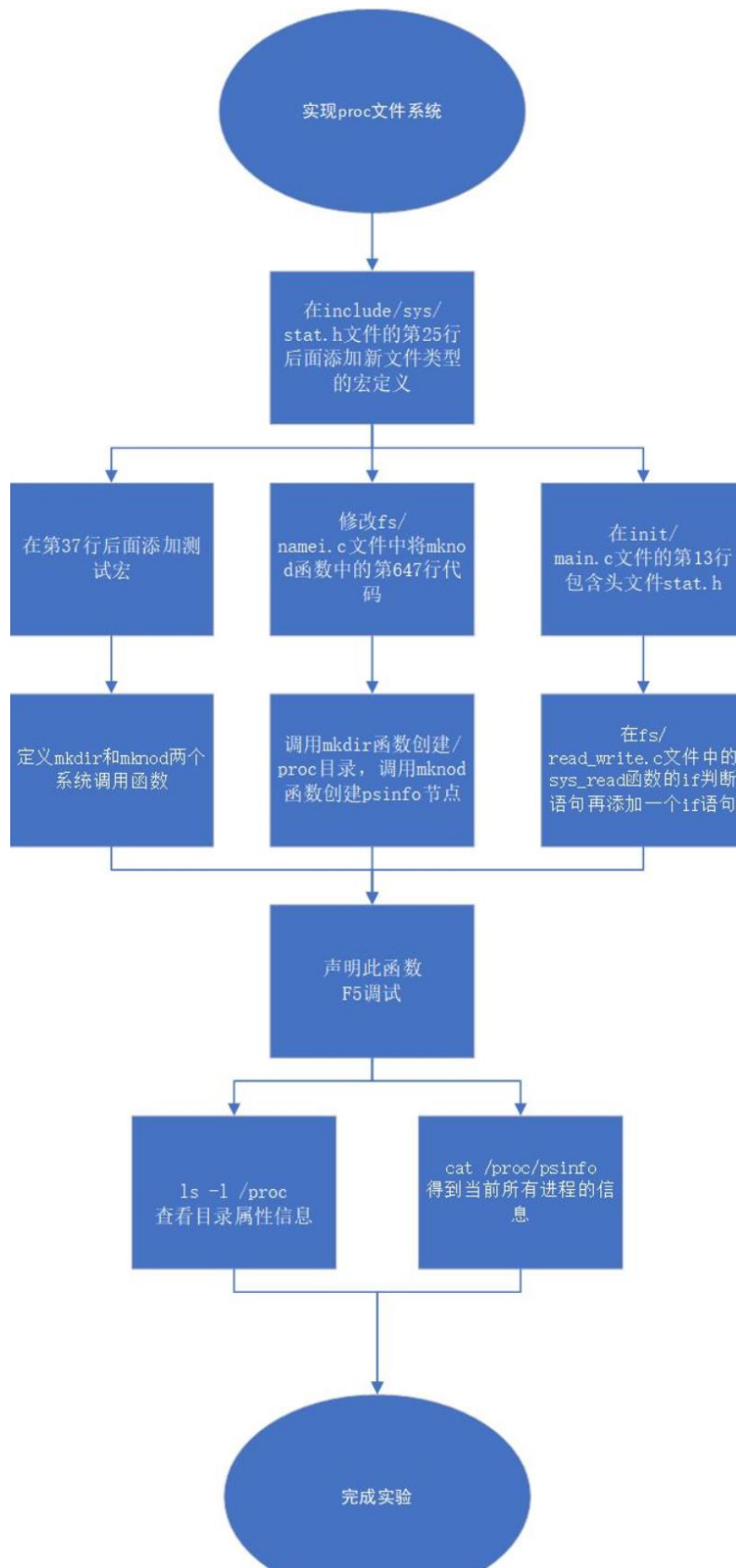
共修改个文件 6 个文件

▼  include/sys/stat.h 		
...	...	@@ -23,6 +23,7 @@ struct stat {
23	23	#define S_IFREG 0100000 // 常规文件。
24	24	#define S_IFBLK 0060000 // 块特殊（设备）文件，如磁盘dev/fd0。
25	25	#define S_IFDIR 0040000 // 目录文件。
	26	+ #define S_IFPROC 0050000
26	27	#define S_IFCHR 0020000 // 字符设备文件。
27	28	#define S_IFIFO 0010000 // FIFO 特殊文件。
28	29	// 文件属性位：
...	...	@@ -34,6 +35,7 @@ struct stat {
34	35	#define S_ISDIR(m) (((m) & S_IFMT) == S_IFDIR) // 是否目录文件。
35	36	#define S_ISCHR(m) (((m) & S_IFMT) == S_IFCHR) // 是否字符设备文件。
36	37	#define S_ISBLK(m) (((m) & S_IFMT) == S_IFBLK) // 是否块设备文件。
	38	+ #define S_ISPROC(m) (((m) & S_IFMT) == S_IFPROC)
37	39	#define S_ISFIFO(m) (((m) & S_IFMT) == S_IFIFO) // 是否FIFO 特殊文件。
38	40	
39	41	#define S_IRWXU 00700 // 宿主可以读、写、执行/搜索。
...	...	

▼  init/main.c 		
...	...	@@ -10,7 +10,7 @@
10	10	// 标准符号常数与类型文件。定义了各种符号常数和类型，并声明了各种函数。
11	11	// 如果定义了__LIBRARY__，则还包括系统调用号和内嵌汇编代码_syscall0()等。
12	12	#include <time.h> // 时间类型头文件。其中最主要定义了tm 结构和一些有关时间的函数原形。
13	-	-
	13	+ #include<sys/stat.h>
14	14	/*
15	15	* we need this inline - forking from kernel space will result
16	16	* in NO COPY ON WRITE (!!!), until an execve is executed. This

图 59

5. 流程图



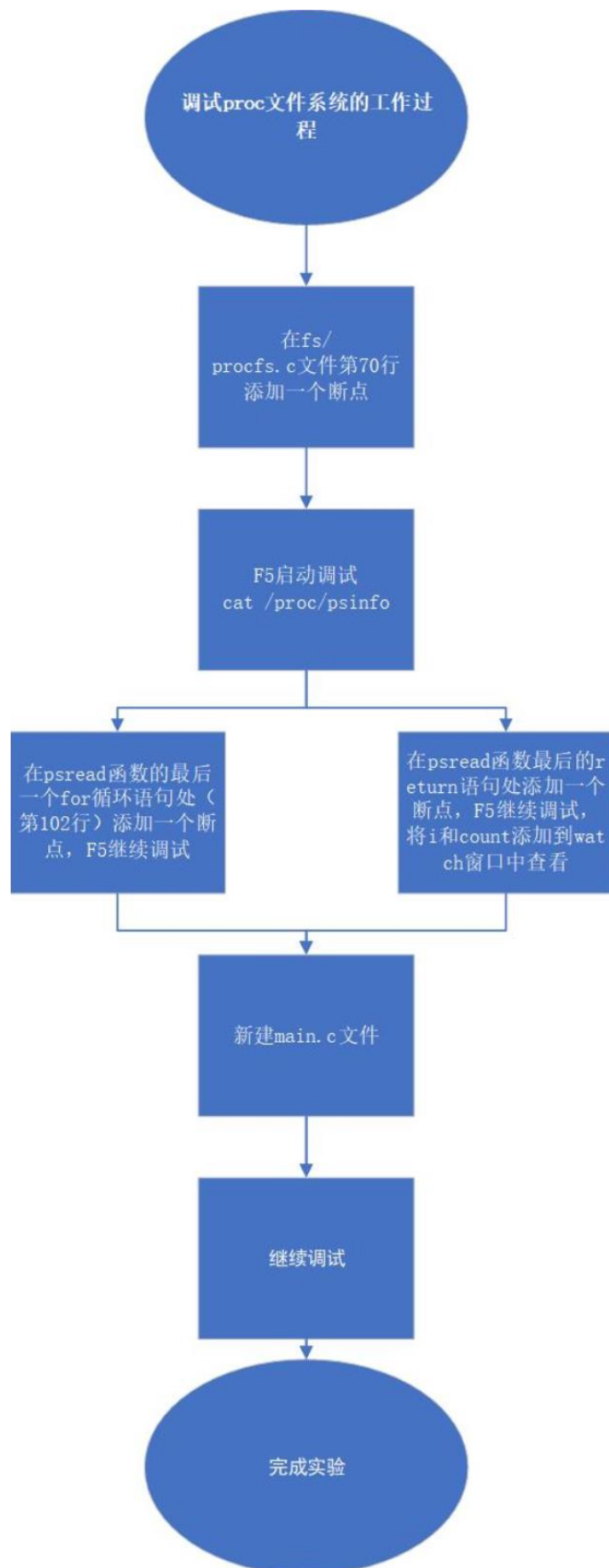


图 60

6. 实验体会

在这次的实验经历中，我深入探索了 proc 文件系统的运作原理，并增进了对文件、目录和索引节点等核心概念的理解。proc 文件系统作为一个独特的存在，它并非基于物理磁盘，而是在系统内存中构建，为用户提供了一个实时访问系统状态信息的窗口，如进程详情、内存占用状况等。

在 Linux 环境下，proc 文件系统展现为一种虚拟文件系统，其内部的文件与目录并非物理实体，而是由内核在需要时动态创建。这一特性使得 proc 文件系统能够高效地为用户提供实时的系统信息。

通过对 proc 文件系统的学习，我掌握了其在内存中如何模拟文件和目录的创建过程，以及如何通过标准的文件操作接口读取和更新这些“虚拟”文件的内容。同时，我也对文件、目录和索引节点这些基本文件系统元素有了更深入的认识。文件作为数据的载体，可以包含文本、图片、音频等多种形式；目录则是一种特殊的文件，用于组织和管理其他文件和目录；而索引节点则存储了文件的元数据信息，如访问权限、所有者以及文件大小等。

这次实验不仅让我对操作系统的文件系统部分有了更为全面的了解，还锻炼了我修改和扩展系统源码的能力。由于实验中的源代码部分打印指令尚未实现，我通过直接修改源码的方式添加了必要的指令，并在修改过程中特别注意了代码之间的逻辑关联和依赖关系。这一经验对于我未来进行操作系统相关的开发工作将大有裨益。

7. 思考与练习

1. 请读者模仿 psinfo 节点的实现方法，实现一个保存物理内存信息的 meminfo 节点，该节点保存的数据可以参考实验八打印输出的物理内存的信息。

```

#include <linux/proc_fs.h>
#include <linux/seq_file.h>

static int meminfo_show(struct seq_file *m, void *v)
{
    unsigned long block_size_bytes, total_size, available_size;
    // 读取物理内存信息
    // ...

    // 将物理内存信息写入meminfo节点
    seq_printf(m, "Block size: %lu\n", block_size_bytes);
    seq_printf(m, "Total size: %lu\n", total_size);
    seq_printf(m, "Available size: %lu\n", available_size);

    return 0;
}

static int meminfo_open(struct inode *inode, struct file *file)
{
    return single_open(file, meminfo_show, NULL);
}

static const struct file_operations meminfo_fops = {
    .owner      = THIS_MODULE,
    .open       = meminfo_open,
    .read       = seq_read,
    .llseek     = seq_lseek,
    .release    = single_release,
};

static int __init meminfo_init(void)
{
    // 创建meminfo节点
    proc_create("meminfo", 0, NULL, &meminfo_fops);

    return 0;
}

static void __exit meminfo_exit(void)
{
    // 删除meminfo节点
    remove_proc_entry("meminfo", NULL);
}

module_init(meminfo_init);
module_exit(meminfo_exit);

```

图 61

2. 实现一个可以读取 jiffies 时钟滴答值的 tickinfo 节点。时钟滴答 jiffies 的说明可以参考实验六预备知识的内容。考虑到 jiffies 的值改变的太快了（10ms 变一次），能够直接将 Linux 0.11 中全局变量 jiffies 的值转换为字符串并放入一个缓冲区中供用户读取吗？在将 jiffies 的值转换为字符串的过程中，其值是否有可能发生改变呢？如何解决这个问题。

按照以下步骤实现一个可以读取 jiffies 时钟滴答值的 tickinfo 节点：

- ①创建一个新的/sys 文件系统的节点，命名为/sys/kernel/tickinfo
 - ②在该节点下创建一个名为 jiffies 的文件，用于读取 jiffies 的值
 - ③在 jiffies 文件中实现读取 jiffies 值的函数，该函数可以直接读取全局变量 jiffies 的值并将其转换为字符串，然后将其放入一个缓冲区中供用户读取
- 不能直接将其转换为字符串并放入缓冲区中供用户读取。因为在转换的过程中，jiffies 的

值可能会发生改变，导致读取的字符串不准确。

可以使用自旋锁来保护 jiffies 的值，确保在转换的过程中 jiffies 的值不会被改变。具体实现方法如下：

- ①在 jiffies 文件中使用自旋锁来保护 jiffies 的值，即在读取 jiffies 值之前先获取自旋锁，读取完毕后再释放自旋锁。
- ②在内核中更新 jiffies 值的地方也需要使用自旋锁来保护 jiffies 的值，以确保在更新 jiffies 值的同时不会被其他进程或中断干扰。

3. 在读取节点 psinfo 的过程中，函数 psread 会将所有进程的信息转换为字符串并放入 psbuffer 缓冲区中，但是在循环遍历所有 64 个进程的过程中，是否会出现进程的信息发生变化的情况呢？如果会发生变化，读者是否可以参考前一个练习解决 jiffies 的值变化太快的方法来解决此问题。

在循环遍历所有 64 个进程的过程中，可能会出现进程的信息发生变化的情况。例如，进程的状态、CPU 时间片、内存使用情况等都可能在遍历过程中发生变化。

要避免这种情况，可以采用类似于解决 jiffies 值变化太快的方法，即在每次遍历时记录当前时间戳，并将其与上次记录的时间戳进行比较，只有当时间戳发生变化时才更新进程信息，否则直接使用上一次的缓存数据。这样可以避免在遍历过程中获取到过时的进程信息。

任务二十 MINIX 1.0 文件系统的实现

1. 实验目的

通过查看 MINIX 1.0 文件系统的硬盘信息，理解 MINIX 1.0 的硬盘管理方式。

学习 MINIX 1.0 文件系统的实现方法。

改进 MINIX 1.0 文件系统的实现方法，加深对 MINIX 1.0 文件系统的理解。

2. 实验内容

编写 Windows 控制台程序，打印输出 MINIX 1.0 文件系统的目录树

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（Windows 控制台程序），然后使用 VSCode 将个人项目克隆到本地磁盘中并打开。

3.2 打印输出 MINIX 1.0 文件系统的目录树

- ①打开一个新的 VSCode，并使用 VSCode 中“file”菜单中的“open folder”打开一个在前面的实验中克隆到本地的 Linux011 内核项目文件夹。
- ②在 VScode “文件资源管理器”窗口中的任意一个文件夹或文件节点上点击右键，然后在弹出的快捷菜单中选择“Reveal in File Explorer”，在打开的文件夹中，将 `harddisk.img` 文件复制到 `C:\minix` 目录下（该路径是 `main` 函数中需要打开的 `harddisk.img` 文件的 `path` 字符串变量指定的磁盘位置）。
- ③再次打开在本实验中通过领取任务在本地创建的 Windows 控制台项目。
- ④生成项目，确保没有语法错误。
- ⑤打开本项目的文件夹，将文件夹下的 `minix.exe` 文件复制到 `C:\minix` 目录下。
- ⑥启动 Windows 控制台，进入 `C:\minix` 目录，然后执行命令“`minix.exe > a.txt`”。此命令会将应用程序打印输出的目录树重定向到文本文件 `a.txt` 中。
- ⑦打开 `a.txt` 文件查看 MINIX 1.0 文件系统的目录树。

3.3 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

5. 流程图

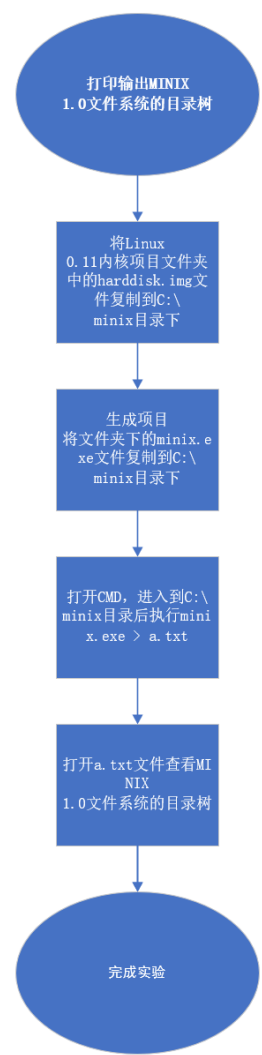


图 62

6. 实验体会

在深入探索 MINIX 1.0 文件系统的实验中，我主要研究了其硬盘管理方式及实现细节。MINIX 1.0 文件系统以磁盘块为基础，每个块固定为 1024 字节，文件则通过这些块的有序组合来存储。

在学习的过程中，我专注于理解文件系统管理模块的核心作用，它负责处理文件的创建、删除、读写等关键操作。这一模块由文件系统缓存、文件系统操作、以及文件系统内存管理等子模块组成，它们共同协作，确保文件系统的稳定运行。

为了优化 MINIX 1.0 文件系统的性能，我尝试引入了多级索引结构的概念。这一改进不仅增强了文件系统的访问效率，也加深了我对 MINIX 1.0 文件系统内部机制的理解。通过实践，我认识到在改进文件系统时，必须综合考虑性能、可靠性和安全性等多方面的因素。

此外，我还通过 makefile 模拟了 minix 系统的构建过程，并实现了对逻辑块的增删查改等操作。尽管这些新增的内容目前对我来说还不够熟悉，但它们为我未来深入学习 Linux 系统和 MINIX 文件系统奠定了坚实的基础。我期待在后续的学习中，能够更深入地理解这些代码段，并将其运用到实际的系统开发中。

7. 思考与练习

1. 参考 load_inode_bitmap 函数写出一个可以将硬盘镜像文件中的整个逻辑块位图都加载到内存中的函数。
2. 打印输出类似于 Linux 0.11 内核命令“df”的输出内容。
3. 将“/usr/root”文件夹下的“hello.c”文件的内容打印输出。
4. 将“/usr/src/linux-0.11.org”文件夹下的“memory.c”的内容打印输出。注意，此文件大于 7KB，所以需要使用二次间接块才能访问所有的数据。
5. 删除“/usr/root/hello.c”文件。
6. 删除“/usr/root/shoe”文件夹。
7. 新建“/usr/root/dir”文件夹。
8. 新建“/usr/root/file.txt”文件，并设置初始大小为 10KB。

参考 load_inode_bitmap 函数写出一个可以将硬盘镜像文件中的整个逻辑块位图都加载到内存中的函数

```
126 void load_all()
127 {
128     int n;
129     for(n = 0; n < sblock.s_ninodes; i++)
130     {
131         load_inode_bitmap();
132     }
133 }
```

图 63

打印输出类似于 Linux 0.11 内核命令“df”的输出内容。

将“/usr/root”文件夹下的“hello.c”文件的内容打印输出

```
char* path = "C:\\usr\\root\\hello.c";  
fd = fopen(path, "rb");  
if(fd==NULL)  
    printf("open file failed!\n");
```

图 64