



计算机操作系统 课程实验

学院：计算机与信息技术学院

专业：计算机科学与技术

班级：计科 1601

姓名：刘欢

学号：16281044

教师：何永忠

2019 年 3 月 6 日

实验一：操作系统初步

作业要求：

- 本次所有实验都在 Linux 中完成

作业题目：

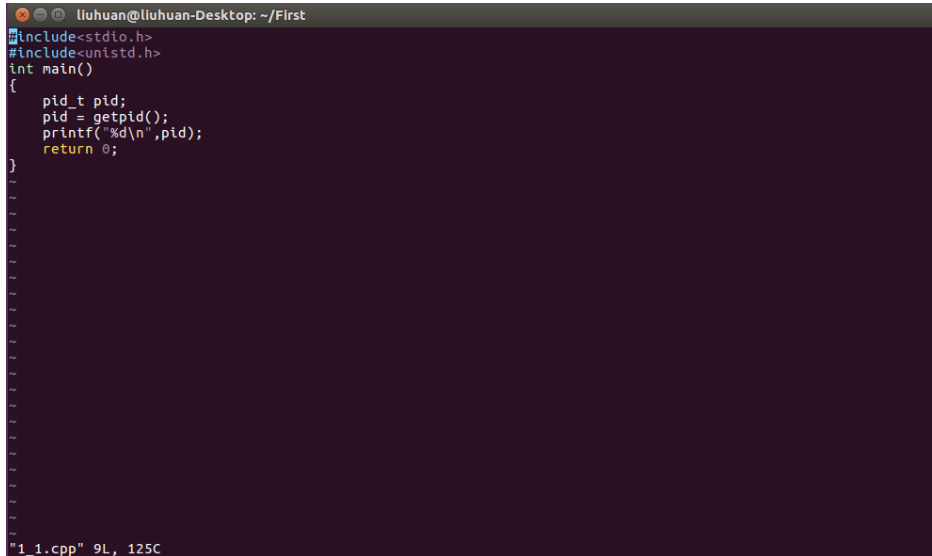
- 一、（系统调用实验）了解系统调用不同的封装形式。
 - 1、参考下列网址中的程序。阅读分别运行用 API 接口函数 `getpid()` 直接调用和汇编中断调用两种方式调用 Linux 操作系统的同一个系统调用 `getpid` 的程序(请问 `getpid` 的系统调用号是多少？linux 系统调用的中断向量号是多少？)。
 - 2、上机完成习题 1.13。
 - 3、阅读 pintos 操作系统源代码，画出系统调用实现的流程图。
- 二、（并发实验）根据以下代码完成下面的实验。
 - 1、编译运行该程序 (`cpu.c`)，观察输出结果，说明程序功能。(编译命令：`gcc -o cpu cpu.c -Wall`) (执行命令：`./cpu`)
 - 2、再次按下面的运行并观察结果：执行命令：`./cpu A & ./cpu B & ./cpu C & ./cpu D` & 程序 `cpu` 运行了几次？他们运行的顺序有何特点和规律？请结合操作系统的特征进行解释。
- 三、（内存分配实验）根据以下代码完成实验。
 - 1、阅读并编译运行该程序(`mem.c`)，观察输出结果，说明程序功能。(命令：`gcc -o mem mem.c -Wall`)
 - 2、再次按下面的命令运行并观察结果。两个分别运行的程序分配的内存地址是否相同？是否共享同一块物理内存区域？为什么？命令：`./mem & ./mem &`
- 四、（共享的问题）根据以下代码完成实验。
 - 1、阅读并编译运行该程序，观察输出结果，说明程序功能。(编译命令：`gcc -o thread thread.c -Wall -pthread`) (执行命令 1：`./thread 1000`)
 - 2、尝试其他输入参数并执行，并总结执行结果的有何规律？你能尝试解释它吗？（例如执行命令 2：`./thread 100000`）（或者其他参数。）
 - 提示：哪些变量是各个线程共享的，线程并发执行时访问共享变量会不会导致意想不到的问题。

实验内容：

◆ 一、系统调用实验

1、

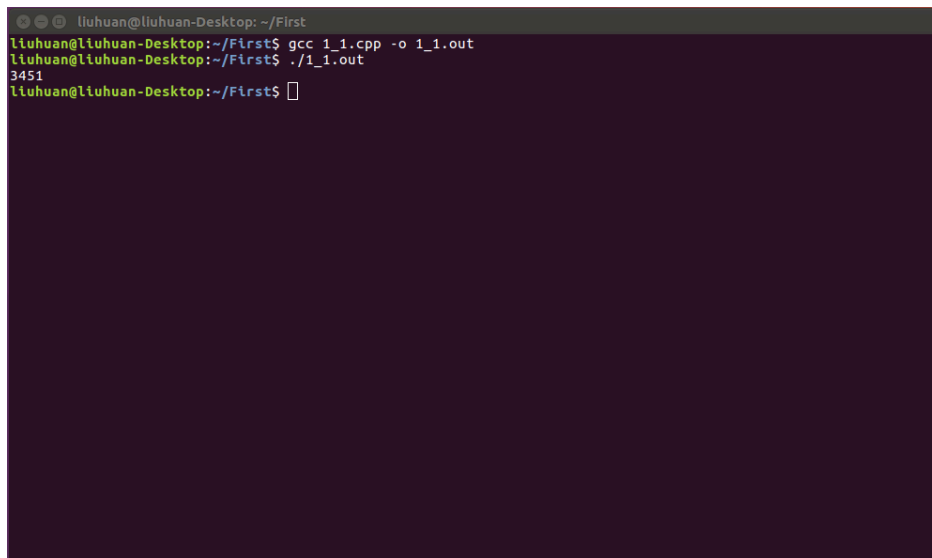
getpid 的系统调用号在 32 位机器下是 14H、Linux 32 位系统的中断向量号是 80H。



```
liuhuan@liuhuan-Desktop: ~/First
#include<stdio.h>
#include<unistd.h>
int main()
{
    pid_t pid;
    pid = getpid();
    printf("%d\n",pid);
    return 0;
}

"1_1.cpp" 9L, 125C
```

图 1 C 源程序



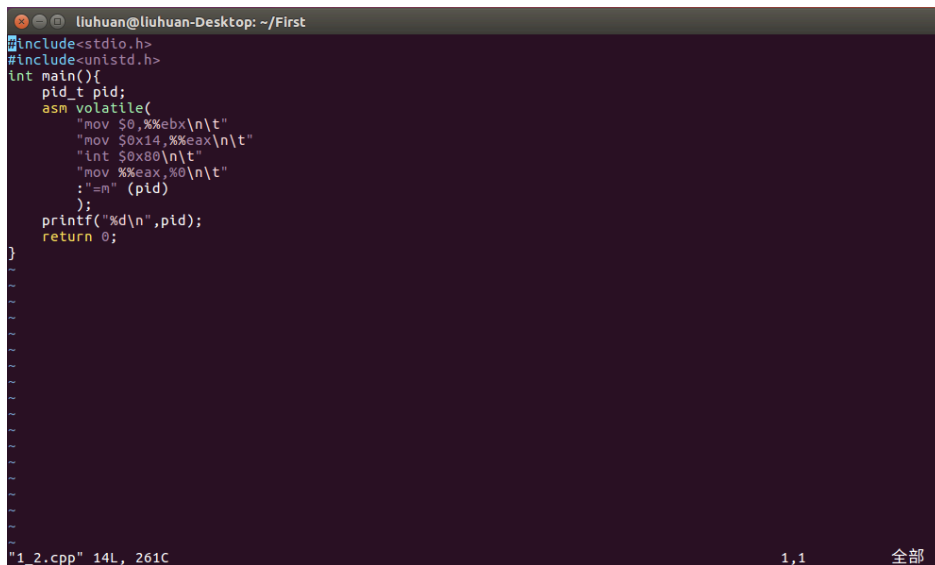
```
liuhuan@liuhuan-Desktop: ~/First
liuhuan@liuhuan-Desktop:~/First$ gcc 1_1.cpp -o 1_1.out
liuhuan@liuhuan-Desktop:~/First$ ./1_1.out
3451
liuhuan@liuhuan-Desktop:~/First$
```

图 2 C 程序编译及运行结果

注：

系统调用号跟操作系统的位数(32bit 或 64bit)和不同的发行版本(debian 和 Ubuntu)有关，例如 Ubuntu 64 位机器上 getpid 的系统调用号是 172，而网上查到 Linux 内核 64 位为 39，32 位为 20。

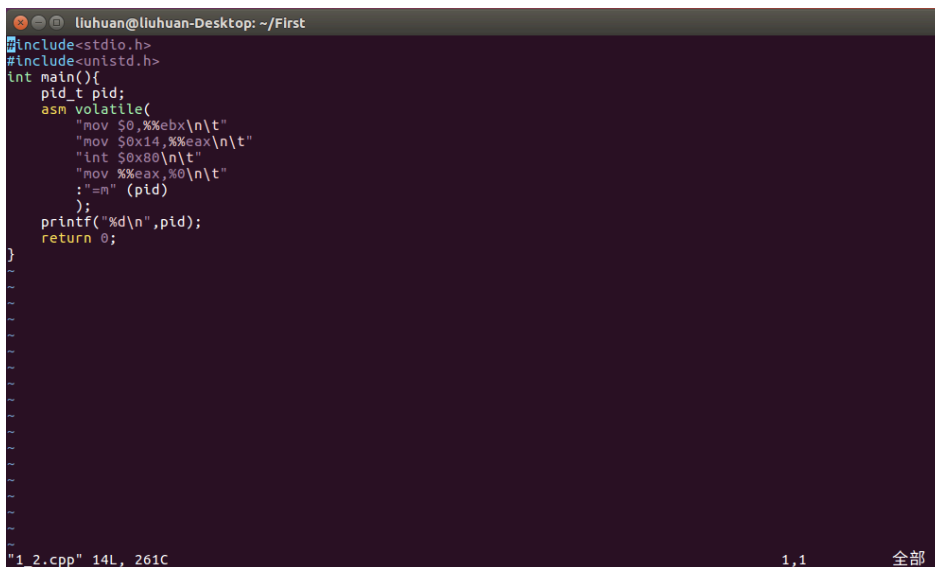
出现不一致的原因是，在 Linux64 位系统中，对 32 位程序进行了兼容操作。之前的 32 位机中，系统通过开放 0x80 搭配系统调用号来实现用户程序使用系统调用功能。但在 64 位机中，已经不使用 int 0x80 作为触发系统调用的机制了，而使用 syscall 指令来触发。但为了保持兼容性，系统仍然支持 int 0x80 进行‘32 位’风格的调用。自然我们的汇编代码使用 20 这个系统调用号，依然可以正确运行在 64 位机器上。



```
#include<stdio.h>
#include<unistd.h>
int main(){
    pid_t pid;
    asm volatile(
        "mov $0,%%ebx\n\t"
        "mov $0x14,%%eax\n\t"
        "int $0x80\n\t"
        "mov %%eax,%0\n\t"
        : "=m" (pid)
    );
    printf("%d\n",pid);
    return 0;
}
```

"1_2.cpp" 14L, 261C 1,1 全部

图 3 系统调用源程序



```
#include<stdio.h>
#include<unistd.h>
int main(){
    pid_t pid;
    asm volatile(
        "mov $0,%%ebx\n\t"
        "mov $0x14,%%eax\n\t"
        "int $0x80\n\t"
        "mov %%eax,%0\n\t"
        : "=m" (pid)
    );
    printf("%d\n",pid);
    return 0;
}
```

"1_2.cpp" 14L, 261C 1,1 全部

图 4 系统调用编译及执行结果

注：

系统调用是指运行在用户空间的程序向操作系统内核请求需要更高权限运行的服务。系统调用提供了用户程序与操作系统之间的接口。操作系统的进程空间可分为用户空间与内核空间，它们需要不同的执行权限，系统调用运行在内核空间。系统调用和普通函数调用非常相似，只是系统调用由操作系统内核提供，运行于内核核心态，普通的库函数调用由函数库或用户自己提供，运行于用户态。

软中断的使用方法，将系统调用号存入 EAX，把函数的其它参数存入其它通用寄存器，最后触发 0x80 号中断。

2、

```
liuhuan@liuhuan-Desktop: ~/First
SECTION .DATA
    hello: db 'Hello World',10
    helloLen: equ $-hello

SECTION .TEXT
    GLOBAL _start
_start:
    mov eax,4
    mov ebx,1
    mov ecx,hello
    mov edx,helloLen
    int 80h
    mov eax,1
    mov ebx,0
    int 80h

"hello.asm" [noeol] 15L, 230C
```

图 5 NASM 格式的汇编代码

注:

1. `eax` 寄存器存放系统写功能号，写的操作是 4 号。
2. `ebx` 寄存器存放文件描述符，1 代表标准输出代表显示器。
3. `ecx` 寄存器存放字符串的首地址，传入 `hello`。
4. `edx` 寄存器存放要输出的字节数量，传入 `helloLen`。
5. 使用 `int 80H` 使用系统调用。
6. `eax` 寄存器存放退出功能号 1。
7. `ebx` 寄存器存放退出的错误码 0，代表无错误。
8. `int 80H` 使用系统调用。

```
liuhuan@liuhuan-Desktop: ~/First
liuhuan@liuhuan-Desktop:~/First$ nasm -f elf64 hello.asm -o hello.o
liuhuan@liuhuan-Desktop:~/First$ ld hello.o -o hello
liuhuan@liuhuan-Desktop:~/First$ ./hello
Hello World
liuhuan@liuhuan-Desktop:~/First$
```

图 6 汇编代码的编译链接与执行

注:

使用 `nasm` 命令对编写的汇编代码进行编译, `-f` 指定输出格式为 Linux 可执行文件常用的 `elf` 格式; 使用 `ld` 命令为生成的可执行文件赋起始地址。如下图 7 所示。

```
liuhuan@liuhuan-Desktop: ~/First
liuhuan@liuhuan-Desktop:~/First$ objdump hello.o -f

hello.o:          文件格式 elf64-x86-64
体系结构: i386:x86-64, 标志 0x00000011:
HAS_RELOC, HAS_SYMS
起始地址 0x0000000000000000

liuhuan@liuhuan-Desktop:~/First$ objdump hello -f

hello:            文件格式 elf64-x86-64
体系结构: i386:x86-64, 标志 0x00000012:
EXEC_P, HAS_SYMS, D_PAGED
起始地址 0x0000000000400084

liuhuan@liuhuan-Desktop:~/First$
```

图 7 使用 objdump 查看可执行文件的头部信息

```
liuhuan@liuhuan-Desktop: ~/First
#include<stdio.h>
int main(){
    printf("Hello World\n");
    return 0;
}

"1_3.cpp" 5L, 75C
```

图 8 C 语言版 HelloWorld 源代码

```
liuhuan@liuhuan-Desktop: ~/First
liuhuan@liuhuan-Desktop:~/First$ gcc 1_3.cpp -o 1_3.o
liuhuan@liuhuan-Desktop:~/First$ ./1_3.o
Hello World
liuhuan@liuhuan-Desktop:~/First$
```

图 9 C 版的编译与执行

3、

阅读 pintos 操作系统代码文件列表：

- PINTOS/src/lib/user/syscall.c
- PINTOS/src/lib/syscall-nr.h
- PINTOS/src/userprog/syscall.c
- PINTOS/src/threads/interrupt.h
- PINTOS/src/threads/interrupt.c
- PINTOS/src/threads/intr-stubs.h
- PINTOS/src/threads/intr-stubs.S

首先，介绍前两个文件的功能和作用，其整体的作用的示意图如图 10 所示。

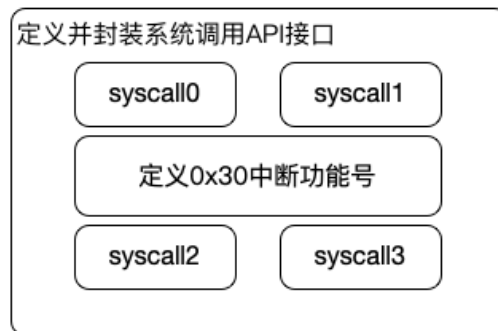


图 10 前两个文件的整体功能

syscall-nr.h 中，定义了 20 个系统调用的功能号，如图 11 所示。这 20 个数字即为中断服务程序区分用户程序所需功能的参考表。

```
/* System call numbers. */
enum
{
    /* Projects 2 and later. */
    SYS_HALT,          /* Halt the operating system. */
    SYS_EXIT,          /* Terminate this process. */
    SYS_EXEC,          /* Start another process. */
    SYS_WAIT,          /* Wait for a child process to die. */
    SYS_CREATE,         /* Create a file. */
    SYS_REMOVE,         /* Delete a file. */
    SYS_OPEN,           /* Open a file. */
    SYS_FILESIZE,       /* Obtain a file's size. */
    SYS_READ,           /* Read from a file. */
    SYS_WRITE,          /* Write to a file. */
    SYS_SEEK,           /* Change position in a file. */
    SYS_TELL,           /* Report current position in a file. */
    SYS_CLOSE,          /* Close a file. */

    /* Project 3 and optionally project 4. */
    SYS_MMAP,           /* Map a file into memory. */
    SYS_MUNMAP,         /* Remove a memory mapping. */

    /* Project 4 only. */
    SYS_CHDIR,          /* Change the current directory. */
    SYS_MKDIR,          /* Create a directory. */
    SYS_READDIR,        /* Reads a directory entry. */
    SYS_ISDIR,          /* Tests if a fd represents a directory. */
    SYS_INUMBER         /* Returns the inode number for a fd. */
};
```

图 11 pintos 的系统调用号

PINTOS/src/lib/user/syscall.c 中，通过内嵌汇编代码的方式定义了系统调用接口：syscall0、syscall1、syscall2 和 syscall3。然后针对 20 个上述功能号及中断服务程序需要的参数数目，封装(注册)了 20 个可供用户程序直接调用的 C 函数，如图 12 所示。

```

/* Projects 2 and later. */
void halt (void) NO_RETURN;
void exit (int status) NO_RETURN;
pid_t exec (const char *file);
int wait (pid_t);
bool create (const char *file, unsigned initial_size);
bool remove (const char *file);
int open (const char *file);
int filesize (int fd);
int read (int fd, void *buffer, unsigned length);
int write (int fd, const void *buffer, unsigned length);
void seek (int fd, unsigned position);
unsigned tell (int fd);
void close (int fd);

/* Project 3 and optionally project 4. */
mapid_t mmap (int fd, void *addr);
void munmap (mapid_t);

/* Project 4 only. */
bool chdir (const char *dir);
bool mkdir (const char *dir);
bool readdir (int fd, char name[READDIR_MAX_LEN + 1]);
bool isdir (int fd);
int inumber (int fd);

```

图 12 封装的系统调用函数

开发者若想继续添加通过软中断实现的系统调用的其他功能,可以通过在上述文件中添加新的功能号定义及其配套函数即可。

在发现了系统调用的入口后,接下来就是分析操作系统如何通过软中断 int 0x30 的方式识别并完成用户需要的 20 个系统调用功能。涉及到的核心文件是其余的 5 个文件。

本着循序渐进的原则方便理解,从 PINTOS/src/userprog/syscall.c 文件开始分析。在这个文件中,只有两个函数 syscall_init(void)和 syscall_handler(struct intr_frame *f)。从函数的命名来看,这两个函数应该是负责系统调用的初始化及系统调用的处理函数。并且处理函数作为初始化函数的参数传入。结合后面代码的功能,总结其基本功能如图 13 所示。

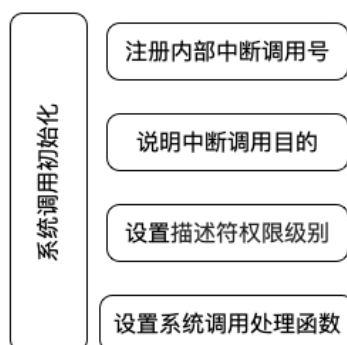


图 13 syscall.c 实现的功能

沿着 syscall.c 中的函数调用,找到了 interrupt.c 中的 register_handler 函数,在这个函数中,通过构造陷阱门修改中断描述符表(IDT),使得 0x30 向量在表中有相应地中断或异常处理程序地入口地址,并允许用户程序使用软中断方式调用系统中断。总结如图 14 所示。

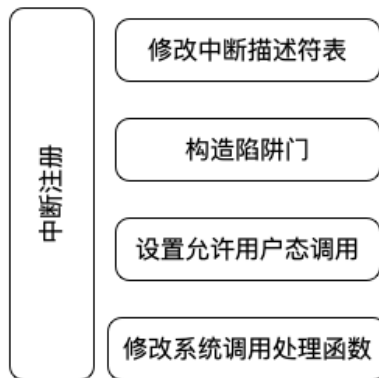


图 14 register_handler 函数的功能

系统中断 int 0x30 的原理到这里基本就结束了，但是目前还不是很清楚，系统中断是如何调用 0x30 对应的中断 handler 呢，那 handler 的参数又是如何传入中断处理函数 syscall_handler 中的呢？

为了解决这个问题，只需要再阅读 z 即可，这个文件是使用汇编编写的，其内容作者做了详尽的注释。尽管不是非常清楚作者使用的各个地址是如何完成功能的，但通读注释，也能总结出这个汇编文件的功能，如图 15 所示。



图 15 intr-stubs.S 的功能

这个汇编文件充当了中断与中断处理之间的桥梁，构造中断处理函数需要的形参，并主动调用中断处理函数。至此，经过一轮的分析，又回到了分析后面 5 个文件的起点 PINTOS/src/userprog/syscall.c。这个文件中定义的 syscall_handler(struct intr_frame *f) 函数，经过一系列的处理，系统中断的处理任务交给了它，一并交给它的还有一个中断帧，中断帧的内容可见 interrupt.h，如图 16 所示。

```

/* Interrupt stack frame. */
struct intr_frame
{
    /* Pushed by intr_entry in intr-stubs.S.
     * These are the interrupted task's saved registers. */
    uint32_t edi; /* Saved EDI. */
    uint32_t esi; /* Saved ESI. */
    uint32_t ebp; /* Saved EBP. */
    uint32_t esp_dummy; /* Not used. */
    uint32_t ebx; /* Saved EBX. */
    uint32_t edx; /* Saved EDX. */
    uint32_t ecx; /* Saved ECX. */
    uint32_t eax; /* Saved EAX. */
    uint16_t gs, :16; /* Saved GS segment register. */
    uint16_t fs, :16; /* Saved FS segment register. */
    uint16_t es, :16; /* Saved ES segment register. */
    uint16_t ds, :16; /* Saved DS segment register. */

    /* Pushed by intrNN_stub in intr-stubs.S. */
    uint32_t vec_no; /* Interrupt vector number. */

    /* Sometimes pushed by the CPU,
     * otherwise for consistency pushed as 0 by intrNN_stub.
     * The CPU puts it just under 'eip', but we move it here. */
    uint32_t error_code; /* Error code. */

    /* Pushed by intrNN_stub in intr-stubs.S.
     * This frame pointer eases interpretation of backtraces. */
    void *frame_pointer; /* Saved EBP (frame pointer). */

    /* Pushed by the CPU.
     * These are the interrupted task's saved registers. */
    void (*eip) (void); /* Next instruction to execute. */
    uint16_t cs, :16; /* Code segment for eip. */
    uint32_t eflags; /* Saved CPU flags. */
    void *esp; /* Saved stack pointer. */
    uint16_t ss, :16; /* Data segment for esp. */
};

```

图 16 中断帧包含的内容

与预期的一样，寄存器的值、中断向量号和错误码等等必要的信息都传递给了 `intr_handler` 函数。虽然系统源代码中并未实现真正满足的功能要求的中断处理函数，但是我们可以通过重构这个函数，结合上 `syscall-nr.h` 中定义的系统调用功能号与寄存器保存值的比对结果，实现定义的各个功能并正确返回结果。

至此分析系统调用的工作全部完成，为了方便理解，整体系统调用的流程图如图 17 所示。

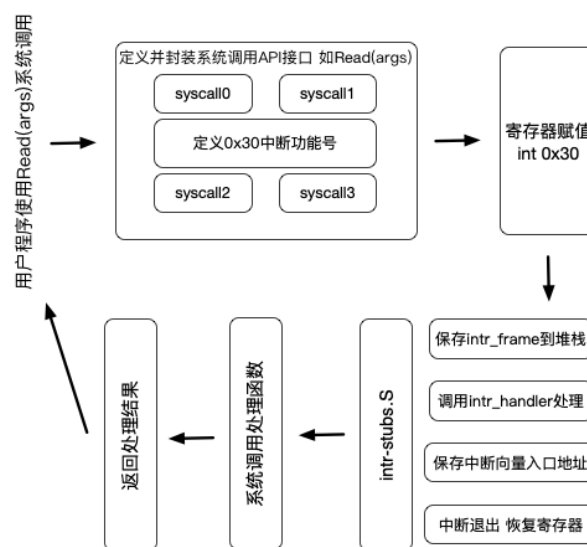
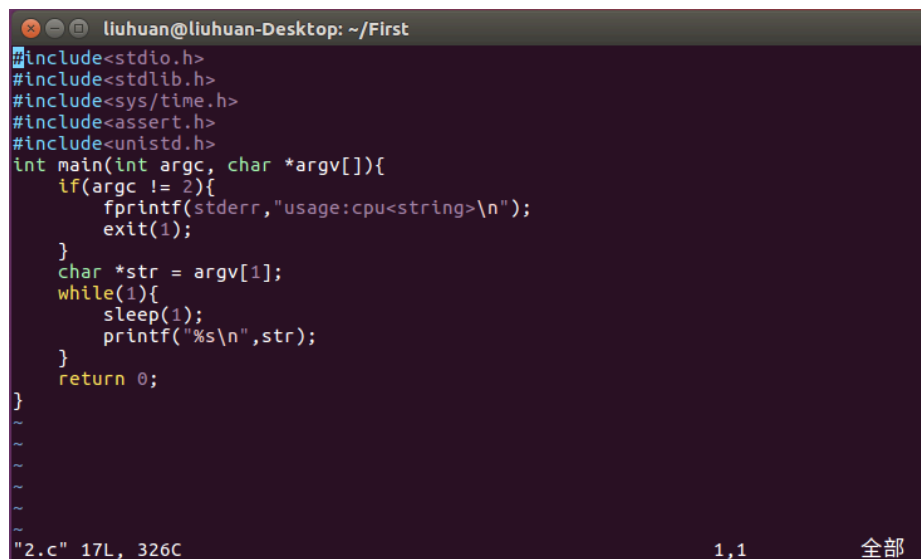


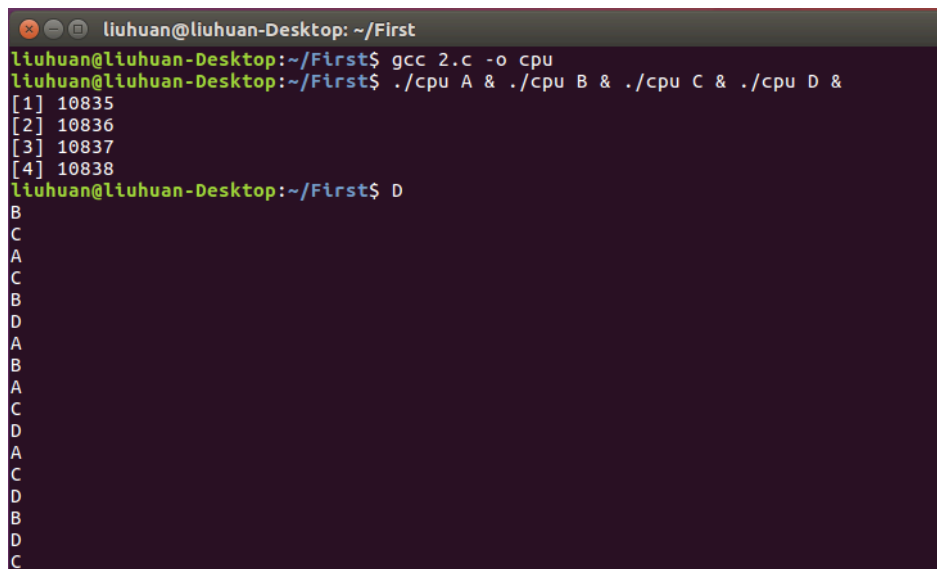
图 17 pintos 系统调用流程

◆ 二、并发实验



```
liuhuan@liuhuan-Desktop: ~/First
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#include<assert.h>
#include<unistd.h>
int main(int argc, char *argv[]){
    if(argc != 2){
        fprintf(stderr, "usage:cpu<string>\n");
        exit(1);
    }
    char *str = argv[1];
    while(1){
        sleep(1);
        printf("%s\n",str);
    }
    return 0;
}
~
~
~
~
~
"2.c" 17L, 326C 1,1 全部
```

图 18 程序源代码



```
liuhuan@liuhuan-Desktop: ~/First
liuhuan@liuhuan-Desktop:~/First$ gcc 2.c -o cpu
liuhuan@liuhuan-Desktop:~/First$ ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 10835
[2] 10836
[3] 10837
[4] 10838
liuhuan@liuhuan-Desktop:~/First$ D
B
C
A
C
B
D
A
B
A
C
D
A
C
D
B
D
C
```

图 19 编译运行结果

注：

程序功能是，若运行时未后缀参数或后缀参数两个及以上，使用标准错误输出设备，即屏幕输出程序的使用方法“usage:cpu<string>”；否则，循环输出输入的参数到屏幕。通过观察，字符输出的顺序没有特定的排列规律，程序输出时有时会单个依次输出字符，有时会同时输出两个。使用的是 Ubuntu16.04 系统，虚拟环境包含 4 个核心，程序运行可以并行执行，同一时刻程序可能运行于不同核心，故可能出现同时输出的情况，这体现了操作系统可以提高硬件的效率。

◆ 三、内存分配实验



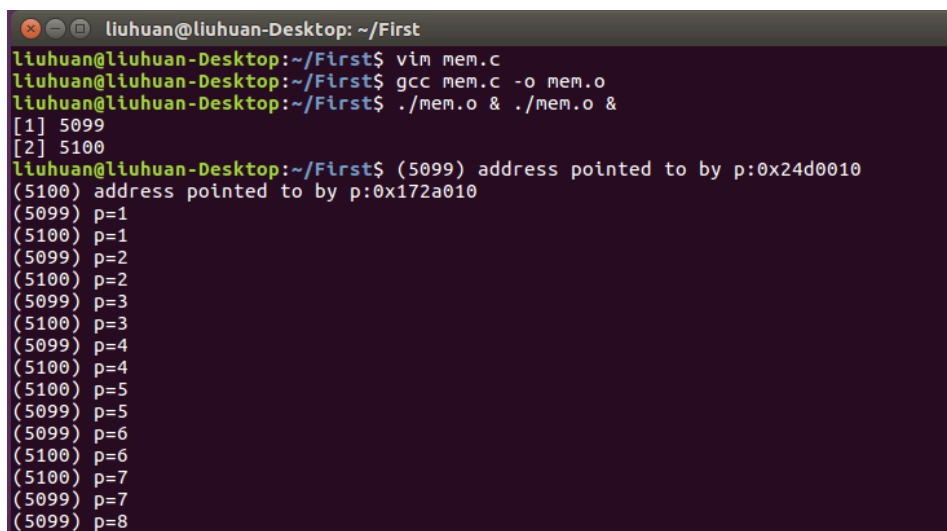
```
liuhuan@liuhuan-Desktop: ~/First
1 #include<unistd.h>
2 #include<stdio.h>
3 #include<stdlib.h>
4 #include<assert.h>
5 int main(int argc, char *argv[])
6 {
7     int *p = malloc(sizeof(int)); //a1
8     assert(p!=NULL);
9     printf("(%d) address pointed to by p:%p\n",getpid(),p); //a2
10    *p = 0; //a3
11    while(1){
12        sleep(1);
13        *p = *p + 1;
14        printf("(%d) p=%d\n",getpid(),*p); //a4
15    }
16    return 0;

```

NORMAL mem.c c 100% 16: 1

"mem.c" 16L, 370C

图 20 实验源代码



```
liuhuan@liuhuan-Desktop: ~/First
liuhuan@liuhuan-Desktop:~/First$ vim mem.c
liuhuan@liuhuan-Desktop:~/First$ gcc mem.c -o mem.o
liuhuan@liuhuan-Desktop:~/First$ ./mem.o & ./mem.o &
[1] 5099
[2] 5100
liuhuan@liuhuan-Desktop:~/First$ (5099) address pointed to by p:0x24d0010
(5100) address pointed to by p:0x172a010
(5099) p=1
(5100) p=1
(5099) p=2
(5100) p=2
(5099) p=3
(5100) p=3
(5099) p=4
(5100) p=4
(5100) p=5
(5099) p=5
(5099) p=6
(5100) p=6
(5100) p=7
(5099) p=7
(5099) p=8

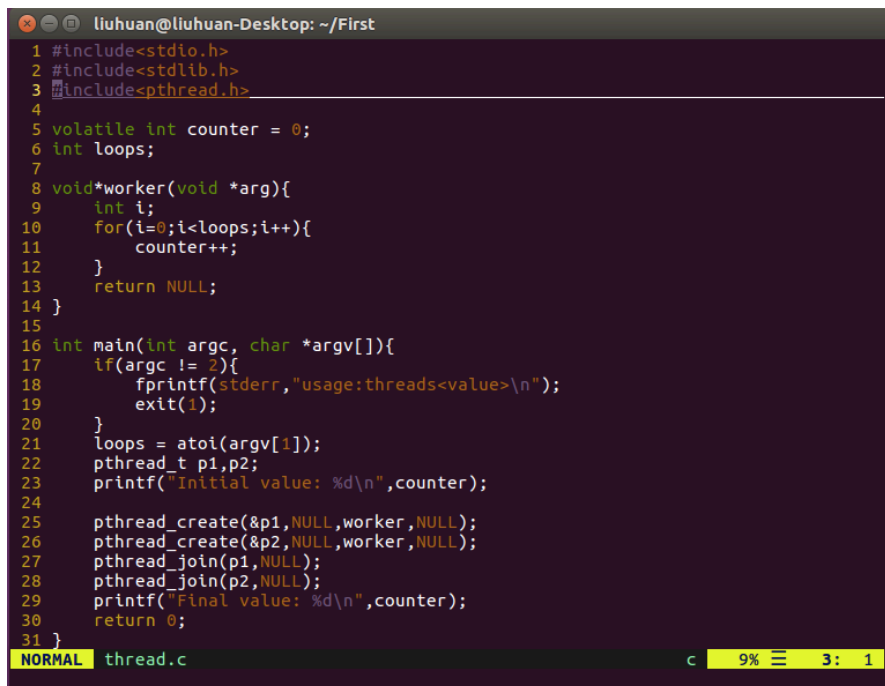
```

图 21 编译及运行结果

注：

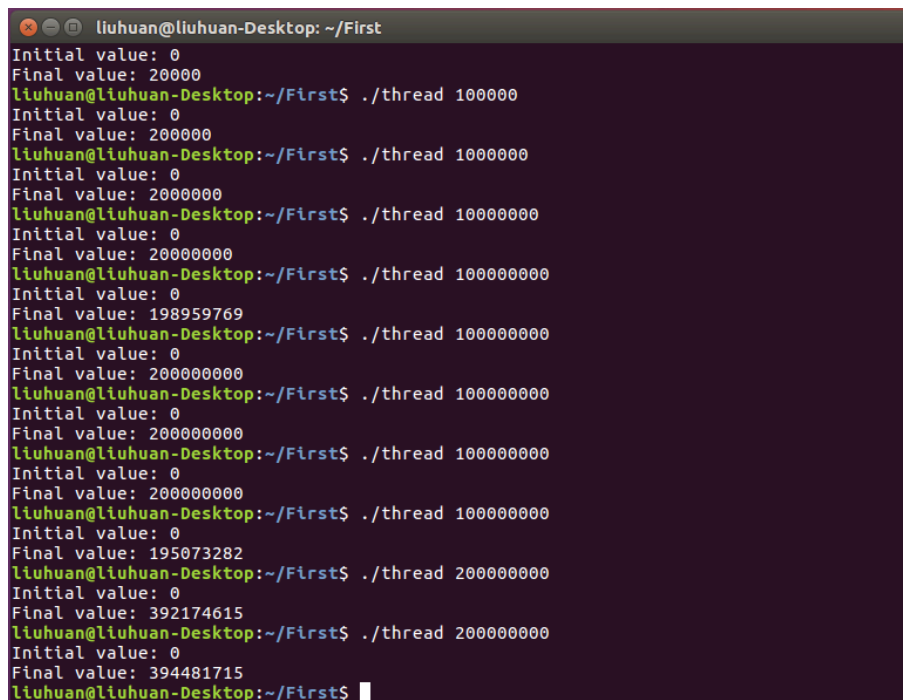
为程序分配的内存地址不相同，但共享同一块物理内存区域。在多任务操作系统中，每一个进程都运行在自己的沙盒中，这个沙盒是虚拟地址空间。这些虚拟地址通过页表映射到物理内存，页表由操作系统维护并被处理器引用。每个进程都拥有属于自己的页表。所以两个程序运行时分配的内存地址不同，但共享同一块内存区域，只是从进程的角度，自己独占一个沙盒。其实实际上它们都使用的是同一块存储器“内存条”。

◆ 四、共享的问题



```
liuhuan@liuhuan-Desktop: ~/First
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<pthread.h>
4
5 volatile int counter = 0;
6 int loops;
7
8 void*worker(void *arg){
9     int i;
10    for(i=0;i<loops;i++){
11        counter++;
12    }
13    return NULL;
14 }
15
16 int main(int argc, char *argv[]){
17     if(argc != 2){
18         fprintf(stderr,"usage:threads<value>\n");
19         exit(1);
20     }
21     loops = atoi(argv[1]);
22     pthread_t p1,p2;
23     printf("Initial value: %d\n",counter);
24
25     pthread_create(&p1,NULL,worker,NULL);
26     pthread_create(&p2,NULL,worker,NULL);
27     pthread_join(p1,NULL);
28     pthread_join(p2,NULL);
29     printf("Final value: %d\n",counter);
30     return 0;
31 }
```

图 22 实验源代码



```
liuhuan@liuhuan-Desktop: ~/First
Initial value: 0
Final value: 20000
liuhuan@liuhuan-Desktop:~/First$ ./thread 100000
Initial value: 0
Final value: 200000
liuhuan@liuhuan-Desktop:~/First$ ./thread 1000000
Initial value: 0
Final value: 2000000
liuhuan@liuhuan-Desktop:~/First$ ./thread 10000000
Initial value: 0
Final value: 20000000
liuhuan@liuhuan-Desktop:~/First$ ./thread 100000000
Initial value: 0
Final value: 198959769
liuhuan@liuhuan-Desktop:~/First$ ./thread 1000000000
Initial value: 0
Final value: 2000000000
liuhuan@liuhuan-Desktop:~/First$ ./thread 10000000000
Initial value: 0
Final value: 2000000000
liuhuan@liuhuan-Desktop:~/First$ ./thread 100000000000
Initial value: 0
Final value: 2000000000
liuhuan@liuhuan-Desktop:~/First$ ./thread 1000000000000
Initial value: 0
Final value: 2000000000
liuhuan@liuhuan-Desktop:~/First$ ./thread 10000000000000
Initial value: 0
Final value: 195073282
liuhuan@liuhuan-Desktop:~/First$ ./thread 2000000000
Initial value: 0
Final value: 392174615
liuhuan@liuhuan-Desktop:~/First$ ./thread 20000000000
Initial value: 0
Final value: 394481715
liuhuan@liuhuan-Desktop:~/First$
```

图 23 编译及试验结果

注:

运行程序过程中, 不断提高输入参数的值, 发现当参数值较小时, 输出值为输入参数的二倍。但当输入大于等于 1000000000 时, 固定程序的输入参数, 发现输出具有一定的随机性。怀疑可能是主函数的两个线程的并发共享变量出现了问题。

当输入参数值较小时, 由于 CPU 运行速度极快, 几乎两个线程创建进程获得 CPU 后就执行完毕, 使得程序的并发存在的问题被掩饰起来。当输入参数较大时, 需要 CPU 进行长时间的运算, 此时线程 1 执行过程中线程 2 也开始执行, 两个线程运行过程出现了时间重合。

两个线程在运行过程中由于对变量 counter 的读写操作不是严格按照“先读后写再读”的顺序进行的，可能导致累加过程，因为覆盖性写入出现累加丢失的情况。

共享变量是 loops 和 counter。