

# 实验报告

课程名称：\_\_\_\_操作系统\_\_\_\_

学生姓名：\_\_\_\_郭鑫玥\_\_\_\_

学生学号：\_\_\_\_16281284\_\_\_\_

学生班级：\_\_\_\_安全 1601\_\_\_\_

2019 年 3 月 8 日

# 实验一：操作系统初步

## 目录

1 系统调用实验.....	3
1.1 实验目的 .....	3
1.2 实验内容.....	3
1.3 问题总结 .....	8
2 并发实验.....	9
2.1 实验目的 .....	9
2.2 实验内容.....	10
2.3 问题总结 .....	12
3 内存分配实验.....	13
3.1 实验目的 .....	13
3.2 实验内容.....	13
3.3 问题总结 .....	14
4 共享实验.....	15
4.1 实验目的 .....	15
4.2 实验内容.....	16
4.3 问题总结 .....	17
5 实验总结.....	18

# 1 系统调用实验

## 1.1 实验目的

了解系统调用不同的封装形式。

## 1.2 实验内容

1.2.1 运行用 API 接口函数 `getpid()` 直接调用方式调用 Linux 操作系统的系统调用 `getpid` 的程序。

编写 c 程序, 功能为利用 `getpid()` 函数获得进程号同时打印输出进程号。

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

int main()
{
    pid_t pid;

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

    return 0;
}
```

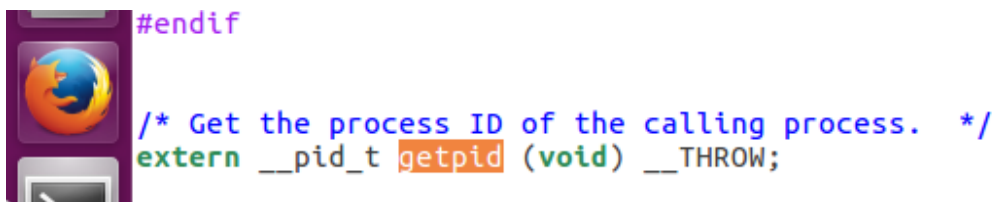
利用 gcc 命令编译运行，得到进程号。

```
yue@yue-virtual-machine: ~/文档/操作系统/lab1
yue@yue-virtual-machine:~/文档/操作系统/lab1$ gcc -o 1.1.1 1.1.1.c
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./1.1.1
28261
yue@yue-virtual-machine:~/文档/操作系统/lab1$
```

根据查询可得到 `getpid` 的系统调用号是 39。

38	setitimer	sys_setitimer	<a href="#">kernel/itimer.c</a>
39	getpid	sys_getpid	<a href="#">kernel/sys.c</a>
40	sendfile	sys_sendfile64	<a href="#">fs/read_write.c</a>
41	socket	sys_socket	<a href="#">net/socket.c</a>

在 /usr/include/unistd.h 中可以找到该函数



```
#endif

/* Get the process ID of the calling process. */
extern __pid_t getpid (void) __THROW;
```

### 1.2.2 运行汇编中断调用方式调用 Linux 操作系统的系统调用 getpid 的程序。

编写 c 程序，在 c 语言中内嵌汇编代码实现系统调用。此处系统调用是一个软中断，既然是中断那么就具有中断号和中断处理程序两个属性，Linux 使用 0x80 号中断作为系统调用的入口，而中断处理程序的地址放在中断向量表里，通过代码可知 getpid 的中断向量号为 14。

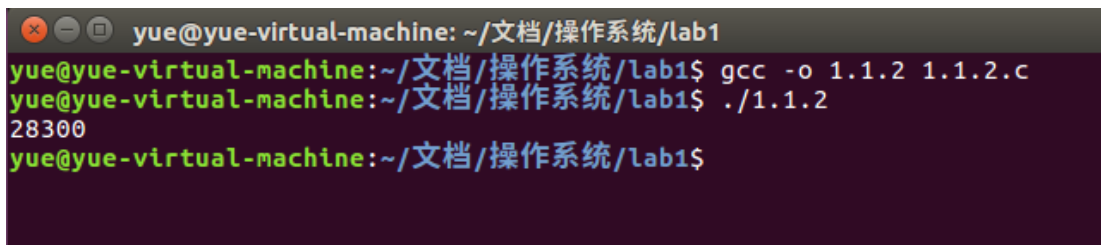
```
#include <stdio.h>
#include <unistd.h>

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;
}
```

利用 gcc 命令编译运行，得到进程号。



```
yue@yue-virtual-machine: ~/文档/操作系统/lab1
yue@yue-virtual-machine:~/文档/操作系统/lab1$ gcc -o 1.1.2 1.1.2.c
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./1.1.2
28300
yue@yue-virtual-machine:~/文档/操作系统/lab1$
```

分别运行程序 1.1.1 和 1.1.2 可获得两个相邻的进程号。

```
yue@yue-virtual-machine: ~/文档/操作系统/lab1
yue@yue-virtual-machine:~/文档/操作系统/lab1$ gcc -o 1.1.2 1.1.2.c
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./1.1.2
28300
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./1.1.1
28303
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./1.1.2
28304
yue@yue-virtual-machine:~/文档/操作系统/lab1$
```

### 1.2.3 完成习题 1.13。

1. 以 C 函数形式实现 printf

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

编译运行 1.1.3.c，可见打印输出“Hello World”

```
yue@yue-virtual-machine: ~/文档/操作系统/lab1
yue@yue-virtual-machine:~/文档/操作系统/lab1$ gcc -o 1.1.3 1.1.3.c
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./1.1.3
Hello World
yue@yue-virtual-machine:~/文档/操作系统/lab1$
```

2. 以汇编代码形式实现 printf

```
printf.asm (~/文档/操作系统/lab1) - gedit
打开(O)  [?]

section .data
msg db "Hello World",0xA
len equ $-msg

section .text

global _start

_start:
    mov eax,4
    mov ebx,1
    mov ecx,msg
    mov edx,len
    int 0x80

    mov eax,1
    xor ebx,ebx
    int 0x80
```

编写好上述代码，然后将其保存为 printf.asm，先使用 NASM 编译 hello.asm 程序：nasm -f elf64 printf.asm。之后使用 ld -s -o printf printf.o 指令连接程序，最后运行程序。

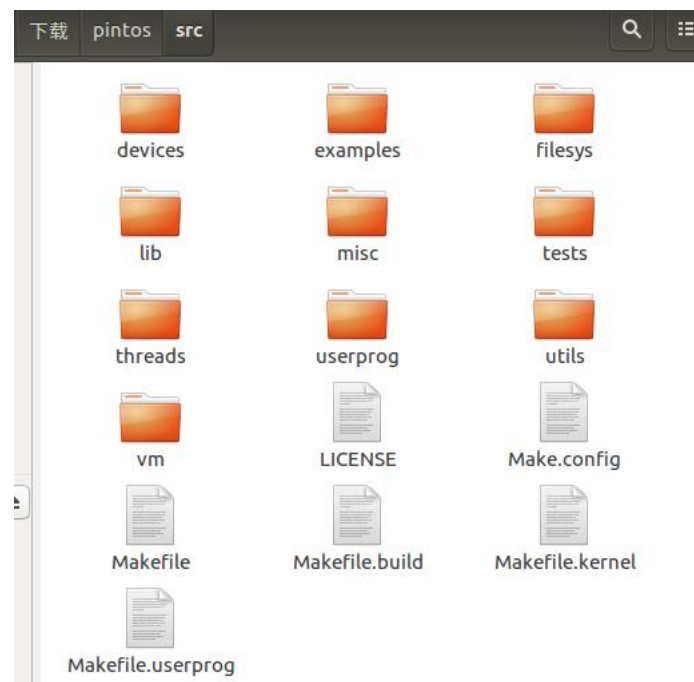
可见程序输出：“Hello World”

```
yue@yue-virtual-machine: ~/文档/操作系统/lab1
yue@yue-virtual-machine:~/文档/操作系统/lab1$ nasm -f elf64 printf.asm
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ld -o printf printf.o
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ls
1.1.1 1.1.1.c 1.1.2 1.1.2.c 1.1.3 1.1.3.c printf printf.asm printf.o
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./printf
Hello World
yue@yue-virtual-machine:~/文档/操作系统/lab1$
```

#### 1.2.4 画出系统调用实现的流程图

Pintos 是 80\*86 架构的简单操作系统框架。支持内核线程，加载和运行用户程序以及文件系统。

下载 Pintos 解压缩后可以看到 src 下的目录：



在 lib/user/syscall.h 中可以看到定义了 20 个系统调用函数

```

syscall.h (~/.下载/pintos/src/lib/user) - gedit
#define EXIT_SUCCESS 0 /* Successful execution. */
#define EXIT_FAILURE 1 /* Unsuccessful execution. */

/* 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);

#endif /* lib/user/syscall.h */

```

在 syscall.c 中可见有四个函数：

```

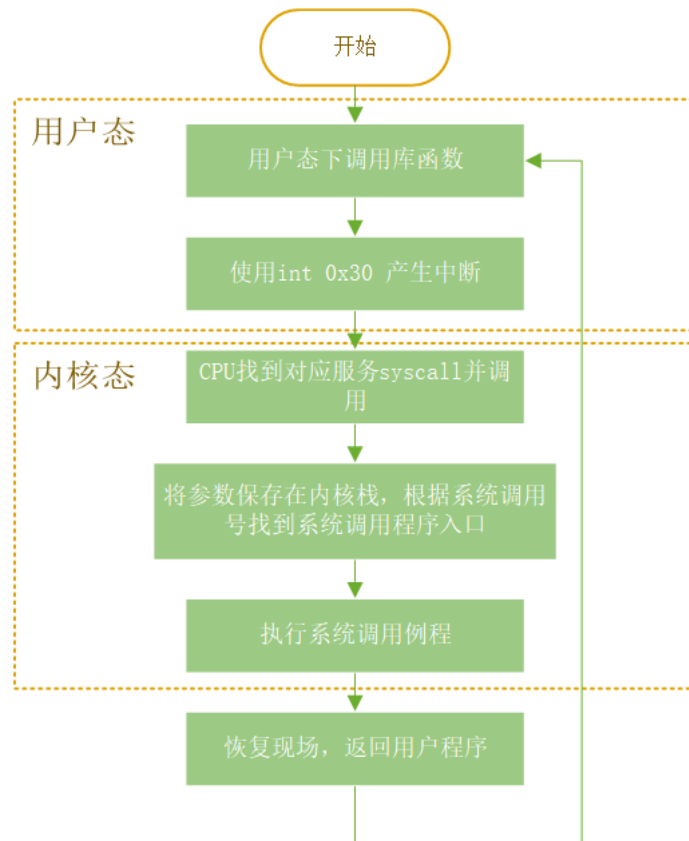
#define syscall0(NUMBER)
({
    int retval;
    asm volatile
        ("pushl %[number]; int $0x30; addl $4, %%esp"
         : "=a" (retval)
         : [number] "i" (NUMBER)
         : "memory");
    retval;
})

/* Invokes syscall NUMBER, passing argument ARG0, and returns the
   return value as an `int'. */
#define syscall1(NUMBER, ARG0)
({
    int retval;
    asm volatile
        ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp"
         : "=a" (retval)
         : [number] "i" (NUMBER),
           [arg0] "g" (ARG0)
         : "memory");
    retval;
})

/* Invokes syscall NUMBER, passing arguments ARG0 and ARG1, and
   returns the return value as an `int'. */

```

系统调用流程图：



## 1.3 问题总结

### 1. 系统调用号的查询

在 Linux 中，每个系统调用被赋予一个系统调用号。这样，通过这个独一无二的号就可以关联系统调用。当用户空间的进程执行一个系统调用的时候，这个系统调用号就被用来指明到底是要执行哪个系统调用。

在 linux 查看 32 位的系统调用号：cat /usr/include/asm/unistd\_32.h

在 linux 查看 64 位的系统调用号：cat /usr/include/asm/unistd\_64.h

### 2. linux 下的系统调用

系统调用可以通过 `syscall()` 函数发起，或者调用每个对应的一个 C 函数，这些函数定义在 `<syscall.h>` 或者 `<unistd.h>` 头文件中。应用程序通过应用编程接口 (API) 而不是直接通过系统调用来编程。因为应用程序使用的这种编程接口实际



上并不需要和内核提供的系统调用一一对应。一个 API 定义了一组应用程序使用的编程接口。它们可以实现成一个系统调用，也可以通过调用多个系统调用来实现，而完全不使用任何系统调用也不存在问题。

用户空间的程序无法直接执行内核代码。它们不能直接调用内核空间中的函数，因为内核驻留在受保护的地址空间上。如果进程可以直接在内核的地址空间上读写的话，系统安全就会失去控制。所以，通知内核的机制是靠软件中断实现的。首先，用户程序为系统调用设置参数。其中一个参数是系统调用编号，如本实验中的 `getpid` 为 39。参数设置完成后，程序执行“系统调用”指令。x86 系统上的软中断由 `int` 产生。这个指令会导致一个异常：产生一个事件，这个事件会致使处理器切换到内核态并跳转到一个新的地址，并开始执行那里的异常处理程序。此时的异常处理程序实际上就是系统调用处理程序。

Linux 下的系统调用主要通过以下几个步骤：

- (1) 将你的系统调用号放进 EAX 中。
- (2) 设置系统调用参数，并且依次将参数放进 EBX、ECX、EDX、ESI、EDI 和 EBP。
- (3) 调用相关中断（对应 Linux 来说是 80h）。
- (4) 最后的调用结果会返回到 EAX 中保存。

## 2 并发实验

### 2.1 实验目的

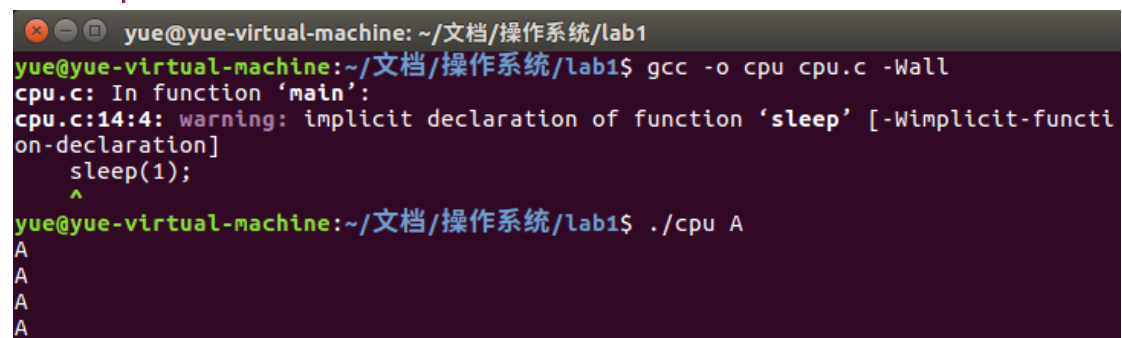
了解体会操作系统并发性。

## 2.2 实验内容

### 2.2.1 编译运行 cpu.c，观察输出结果，说明程序功能。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.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;
}
```



```
yue@yue-virtual-machine: ~/文档/操作系统/lab1
yue@yue-virtual-machine:~/文档/操作系统/lab1$ gcc -o cpu cpu.c -Wall
cpu.c: In function 'main':
cpu.c:14:4: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
     sleep(1);
     ^
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./cpu A
A
A
A
A
```

编译并运行 cpu.c 文件，可见连续输出指定字符。该程序功能为每隔一秒输  
出一次命令行参数，若没有正确接受参数则提示用户正确输入参数。仅运行一个  
cpu 程序时：循环输出“A”

## 2.2.2 同时运行多个 cpu 程序

```
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 30879
[2] 30880
[3] 30881
[4] 30882
yue@yue-virtual-machine:~/文档/操作系统/lab1$ A
C
D
B
C
A
D
B
C
A
D
```

同时运行四个 cpu 程序时，可见同时有四个进程：30879、30880、30881、30882，在运行时可以看到是四个程序交替运行，四个进程轮流运行，顺序并无一定的规律。同时进行任务 A、B、C、D，CPU 会轮流地分配给 A、B、C、D 使用，只是之间的时间间隔太小，用户不能感觉出来，就觉得是多个任务同时进行一样，这就是并发性。在本实验中，通过 sleep 函数加长时间间隔。

```
root@yue-virtual-machine: /home/yue/文档/操作系统/lab1
yue@yue-virtual-machine:~/文档/操作系统/lab1$ su
密码:
root@yue-virtual-machine:/home/yue/文档/操作系统/lab1# cat /proc/sys/kernel/randomize_va_space
2
root@yue-virtual-machine:/home/yue/文档/操作系统/lab1# echo 0 > /proc/sys/kernel/randomize_va_space
root@yue-virtual-machine:/home/yue/文档/操作系统/lab1# cat /proc/sys/kernel/randomize_va_space
0
root@yue-virtual-machine:/home/yue/文档/操作系统/lab1#
```

通过命令行去随机化操作后，再次运行程序，可以到的以下结果：

```
[1] 14813
[2] 14814
[3] 14815
[4] 14816
yue@yue-virtual-machine:~/文档/操作系统/lab1$ B
A
C
D
B
A
C
D
B
A
C
D
B
A
C
D
B
A
C
```

程序均已“BACD”的顺序输出，可见此时已经不再是之前的随机算法。

## 2.3 问题总结

### 1. gcc 编译时 -Wall 的含义

-Wall 选项意思是编译后显示所有警告；-w 的意思是关闭编译时的警告，也就是编译后不显示任何 warning，因为有时在编译之后编译器会显示一些例如数据转换之类的警告，这些警告是我们平时可以忽略的。-W 选项类似-Wall，会显示警告，但是只显示编译器认为会出现错误的警告。

### 2. 操作系统的并发和并行

并行性和并发性是既相似又有区别的两个概念。并行性是指两个或多个事件在同一时刻发生。而并发性是指两个或多个事件在同一时间间隔发生。在多道程序环境下，并发性是指在一段时间内宏观上有多个程序在同时运行，例如本例中 A、B、C、D 四个 cpu 程序在同时运行，但在单处理机系统中，每一时刻却仅能

有一道程序执行，故微观上这些程序只能是分时地交替执行。

## 3 内存分配实验

### 3.1 实验目的

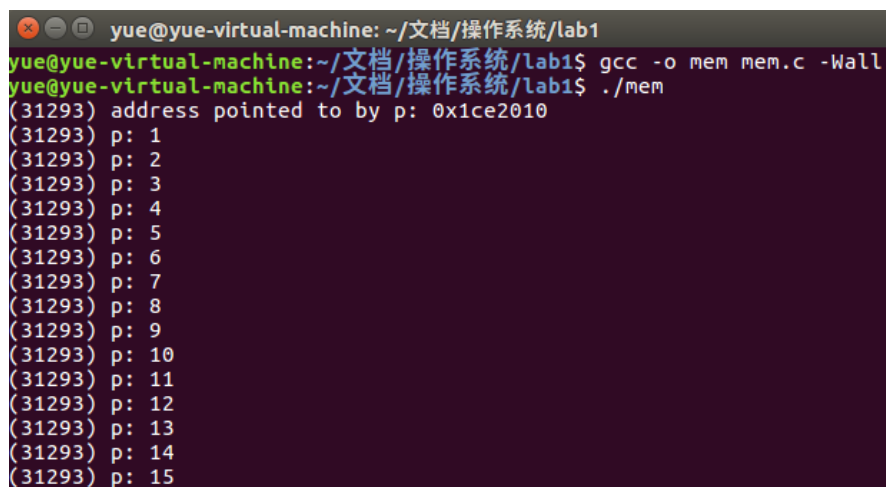
了解操作系统内存分配原理逻辑。

### 3.2 实验内容

3.2.1 编译运行 mem.c，观察输出结果，说明程序功能。

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

int main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int)); // a1
    assert(p != NULL);
    printf("(%) address pointed to by p: %p\n",
        getpid(), p); // a2
    *p = 0; // a3
    while (1) {
        sleep(1);
        *p = *p + 1;
        printf("(%) p: %d\n", getpid(), *p); // a4
    }
    return 0;
}
```



```
yue@yue-virtual-machine: ~/文档/操作系统/lab1
yue@yue-virtual-machine:~/文档/操作系统/lab1$ gcc -o mem mem.c -Wall
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./mem
(31293) address pointed to by p: 0x1ce2010
(31293) p: 1
(31293) p: 2
(31293) p: 3
(31293) p: 4
(31293) p: 5
(31293) p: 6
(31293) p: 7
(31293) p: 8
(31293) p: 9
(31293) p: 10
(31293) p: 11
(31293) p: 12
(31293) p: 13
(31293) p: 14
(31293) p: 15
```

编译并运行 mem.c 文件，可见运行 ./mem 时程序输出运行进程的进程号以及

所指向地址的初始位置，并循环使下一指针所指向的内容加 1，以检验是否共享同一块物理内存区域。

仅运行一个 mem.c 程序时：初始状态  $p=0x1ce2010$   $*p=0$ ，可见该进程循环使得  $p$  指向的值逐次累加 1。

### 3.2.2 运行两个 mem 程序



```
yue@yue-virtual-machine: ~/文档/操作系统/lab1
[1] 31314
[2] 31315
yue@yue-virtual-machine:~/文档/操作系统/lab1$ (31315) address pointed to by p: 0x1c3e010
(31314) address pointed to by p: 0x2200010
(31314) p: 1
(31315) p: 1
(31314) p: 2
(31315) p: 2
(31314) p: 3
(31315) p: 3
(31314) p: 4
(31315) p: 4
(31314) p: 5
(31315) p: 5
(31314) p: 6
(31315) p: 6
(31315) p: 7
(31314) p: 7
(31314) p: 8
```

同时运行两个 mem.c 程序时，可以看到，此时拥有两个进程：31314 和 31315。

其中进程 31315 分配的内存地址为：0x1c3e010，进程 31314 分配的内存地址为 0x2200010，所以两个分别运行的程序分配的内存地址不相同。

之后两个进程开始进入循环部分，可见进程 31314 每循环一次，指针  $p$  所指向的内容加 1，此后进程 31315 每循环一次，指针  $p$  在原有的基础上所指向内容加 1，并未受到 31314 进程的影响。所以可知两个分别运行的程序不共享同一块物理内存区域。

## 3.3 问题总结

### 1. C 语言中 $*p=*p+1$ 的含义，以及各类指针操作辨析

`*p=*p+1` 的效果：指针不移动，但指针所指的数据有+1 效果。

`*p++` 的效果：指针向下移动一个单位（对于 `int` 变量的数组，指针移动了 4 字节），指针内的内容并未变化。

`*p=*(p+1)` 的效果：p 指向的内存地址的下一个地址的数据

## 2. 输出地址的方法

`%p` 格式符会输出指针本身的值，也就是指针指向的地址值。该输出为 16 进制形式，具体输出值取决于指针指向的实际地址值。`%p` 一般仅用于 `printf` 及同类函数中。形式为 `printf("%p", varp)`；其中后续参数 `varp` 为某一个指针变量。

# 4 共享实验

## 4.1 实验目的

了解操作系统线程共享的问题。

## 4.2 实验内容

### 4.2.1 编译运行 thread.c，观察输出结果，说明程序功能。

```
volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value : %d\n", counter);
    return 0;
}
```

在 Linux 系统下，与线程相关的函数都定义在 pthread.h 头文件中。

创建线程函数是 pthread\_create 函数。线程等待函数 pthread\_join 函数，调用该函数的线程将挂起等待，直到线程终止。thread.c 程序功能是将输入的命令行参数传入，创建两个线程，并在指定线程运行 worker 函数完成计数。由此判断全局变量在不同的线程当中访问全局变量是否是共享的



```
yue@yue-virtual-machine: ~/文档/操作系统/lab1
yue@yue-virtual-machine:~/文档/操作系统/lab1$ gcc -o thread thread.c -Wall -pthread
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./thread 1000
Initial value : 0
Final value : 2000
yue@yue-virtual-machine:~/文档/操作系统/lab1$
```

执行测试，发现得到的统计值为输入值的两倍。



```
yue@yue-virtual-machine: ~/文档/操作系统/lab1
yue@yue-virtual-machine:~/文档/操作系统/lab1$ gcc -o thread thread.c -Wall -pthread
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./thread 1000
Initial value : 0
Final value : 2000
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./thread 100
Initial value : 0
Final value : 200
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./thread 3000
Initial value : 0
Final value : 6000
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./thread 0
Initial value : 0
Final value : 0
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./thread 722
Initial value : 0
Final value : 1444
yue@yue-virtual-machine:~/文档/操作系统/lab1$ ./thread 100000
Initial value : 0
Final value : 200000
yue@yue-virtual-machine:~/文档/操作系统/lab1$
```

多次测试发现仍然符合该规律，final value 的值总是为输入参数的二倍。在这个函数中，worker 函数是两个线程共有的运行函数，全局变量 loops，counter 是两个线程共享的，loops 值等于输入的参数值，而两个线程执行时，由结果可见 counter 值是可累加的，所以可知全局变量在多个线程中是共享的

虽然线程共享全局变量相对于进程通信会给线程通信带来巨大的方便，但是不做控制的进行访问全局变量也是致命的，带来巨大程序 bug，并且难以发现。当多个线程同时对一个全局变量操作，会出现资源竞争问题，从而导致数据结不正确，即遇到线性安全问题。

## 4.3 问题总结

### 1. 进程与线程的区别

进程：程序的一个动态运行实例，承担分配系统资源的实例。（Linux 实现进程的主要目的是资源独占）

线程：在进程的內部运行（进程的地址空间）运行的一个分支，也是调度的基本单位。（Linux 实现线程的主要目的是资源共享）。线程所有的资源由进程提

供。

## 2. gcc 中-pthread 和-lpthread 的区别

编译选项中指定 -pthread 会附加一个宏定义 -D\_REENTRANT, 该宏会导致 libc 头文件选择那些 thread-safe 的实现; 链接选项中指定 -pthread 则同 -lpthread 一样, 只表示链接 POSIX thread 库。由于 libc 用于适应 thread-safe 的宏定义可能变化, 因此在编译和链接时都使用 -pthread 选项而不是传统的 -lpthread 能够保持向后兼容, 并提高命令行的一致性。

## 5 实验总结

通过本次实验, 我对于操作系统的系统调用, 并发性, 内存分配, 共享问题等有了进一步的理解, 同时也对于 linux 系统掌握了基本的操作。在这四个不同的实验过程中, 出现了很多知识上的短缺和疑惑, 通过查询学习即时的填补了知识空缺。在完成实验后更深的理解了课堂上所讲的概念, 尤其直观的看到关于并发的交替进行和内存分配等比较抽象的问题, 也加深了记忆。此外由于之前欠缺了汇编方面的知识, 在本次实验中通过了解学习我也掌握了一些关于汇编知识的运用。希望之后能够继续加强实验动手能力, 加油~