实验一：操作系统初步

16281025

一、（系统调用实验）了解系统调用不同的封装形式。

要求：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？)。

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

int main()
{
    pid_t pid;

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

    return 0;
}
```

```c
#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;
```

答：中断向量号 80H，系统调用号 14H

2、上机完成习题 1.13。·

C 语言：

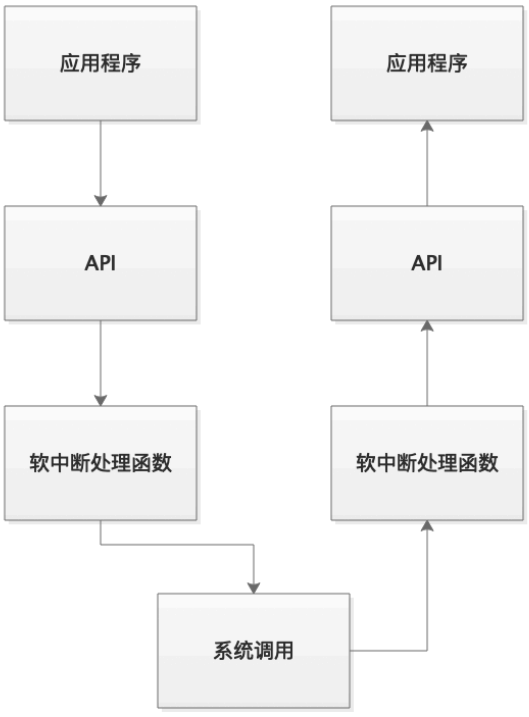
```
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("Hello World\n");
return 0;
}
```

汇编：

```
_main:                                  ## @main
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $32, %rsp
    leaq L_.str(%rip), %rax
    movl    $0, -4(%rbp)
```

```
        movl    %edi, -8(%rbp)
        movq    %rsi, -16(%rbp)
        movq    %rax, %rdi
        movb    $0, %al
        callq   _printf
        xorl  %ecx, %ecx
        movl    %eax, -20(%rbp)             ## 4-byte Spill
        movl    %ecx, %eax
        addq    $32, %rsp
        popq    %rbp
        retq
        .cfi_endproc
                                                ## -- End function
        .section   __TEXT,__cstring,cstring_literals
L_.str:                                         ## @.str
        .asciz    "Hello World\n"
```

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。

要求：

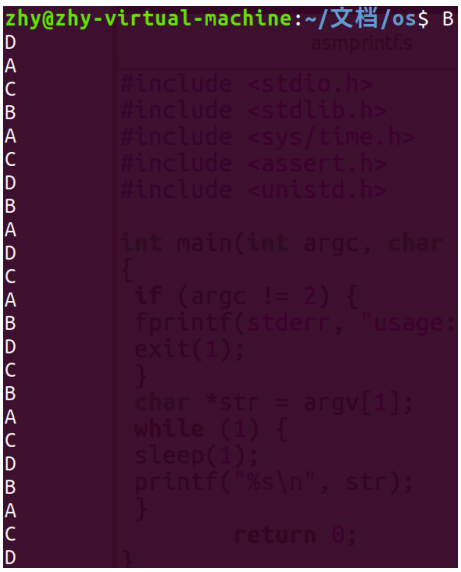
1、 编译运行该程序（cpu.c），观察输出结果，说明程序功能。

(编译命令： gcc -o cpu cpu.c –Wall)（执行命令：./cpu）

功能：每隔 1 秒输出一次传入的参数字符串，若参数格式不正确则输出提示。

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。

运行结果：



运行了四次，可以看到字符的输出并非按命令给出的顺序，而是 BDAC 的顺序开始运行，每一组输出也并非相同顺序，可以体现出操作系统对该四个进程进行了线程调度，导致其并非严格意义上并行执行，在调度过程中导致了四个进程的等待时长产生细微的差异。

三、（内存分配实验）根据以下代码完成实验。

要求：

2、 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)

程序功能：首先输出程序 pid 和申请的内存空间地址，然后对该空间内的数据进行自增并输出。

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 &

运行结果：

```
zhy@zhy-virtual-machine:~/文档/os$ (8681) address pointed to by p: 0x5623c2c7a010
(8682) address pointed to by p: 0x55c1723b5010
(8681) p: 1
(8682) p: 1
(8681) p: 2
(8682) p: 2
(8682) p: 3
(8681) p: 3
(8681) p: 4
(8682) p: 4
(8681) p: 5
(8682) p: 5
(8681) p: 6
(8682) p: 6
(8681) p: 7
(8682) p: 7
(8681) p: 8
(8682) p: 8
(8682) p: 9
(8681) p: 9
(8682) p: 10
(8681) p: 10
(8682) p: 11
(8681) p: 11
```

结论：

两个同时运行的程序所分配到的地址并不相同（理论上可以相同但多次试验都不同），它们并不公用内存因为用户态下只能访问自己内存空间内的地址，程序所获取到的指针地址只是用户态下的偏移地址，其物理地址并不相同，所以对该内存地址内的数据进行自增时，无论指针地址是否相同，都不会相互影响，体现了操作系统内存模型的线程安全。

四、（共享的问题）根据以下代码完成实验。

要求：

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）

```
zhy@zhy-virtual-machine:~/文档/os$ ./thread 1000
Initial value : 0
Final value : 2000
```

程序功能：输入参数 n，创建两个线程，对同一内存地址分别进行 n 次自增。

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。）

```
zhy@zhy-virtual-machine:~/文档/os$ ./thread 100000
Initial value : 0
Final value : 147924
```

其结果在输入参数的 1-2 倍之间。同时访问共享的内存会发生意外，可能丢失修改，如同时读取了值并自增，两次自增操作表现为只自增了一次。

3、 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。

Loops 和 counter 是共享的。同时访问共享的内存会发生意外，可能丢失修改，如同时读取了值并自增，两次自增操作表现为只自增了一次。