实验一：操作系统初步

1 实验要求与步骤

1.1 了解系统调用不同的封装形式

1.1.1 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？)。

直接调用代码：

```
#include <stdio.h>
#include <unistd.h>
int main()
{
  pid_t pid;
  pid = getpid();
  printf("%d\n",pid);
  return 0;
}
```

汇编中断调用代码：

```
 #include<stdio.h>
 #include <unistd.h>
 int main()
 { pit_d 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;
 }
```

程序调用结果：

```
laybing@laybing-VirtualBox:~$ ./test1
1725
laybing@laybing-VirtualBox:~$ ./test2
1726
```

getpid 的系统调用号是 1725，系统调用的中断向量号是 1726。

1.1.2 上机完成习题 1.13。

printf(“Hello World!\n”)

分别用相应的 Linux 系统调用 C 函数形式和汇编代码两种形式来实现上述命令

C 函数代码及运行结果：

```
#include<stdio.h>
int main()
{
  printf("Hello World!\n");
  return 0;
}
```

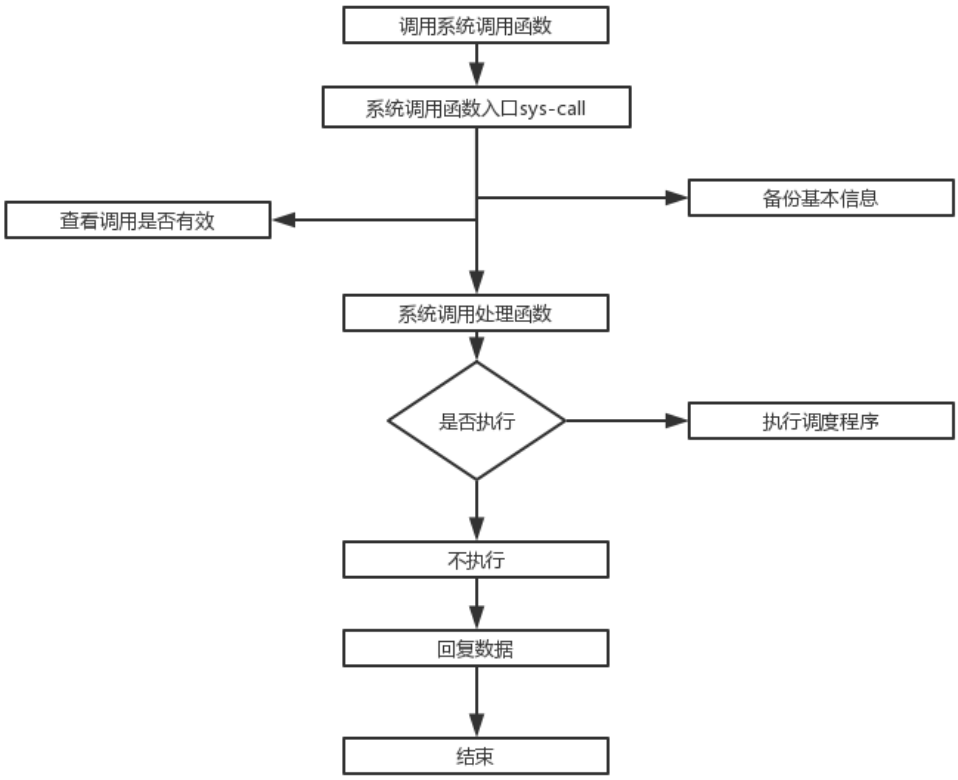
```
laybing@laybing-VirtualBox:~$ ./test3
Hello World!
```

汇编代码及运行结果:

```
section .data                           ;section declaration
msg     db      "Hello, world!",0xA     ;our dear string
len     equ     $ - msg                 ;length of our dear string
section .text                           ;section declaration
                    ;we must export the entry point to the ELF linker or
   global _start    ;loader. They conventionally recognize _start as their
                    ;entry point. Use ld -e foo to override the default.
_start:
;write our string to stdout
       mov     eax,4   ;system call number (sys_write)
       mov     ebx,1   ;first argument: file handle (stdout)
       mov     ecx,msg ;second argument: pointer to message to write
       mov     edx,len ;third argument: message length
       int     0x80    ;call kernel
;and exit
       mov     eax,1   ;system call number (sys_exit)
       xor     ebx,ebx ;first syscall argument: exit code
       int     0x80    ;call kernel
```

```
laybing@laybing-VirtualBox:~/nasm-2.11.08$ ls hello
hello
laybing@laybing-VirtualBox:~/nasm-2.11.08$ ./hello
Hello, world!
```

1.1.3 阅读 pintos 操作系统源代码，画出系统调用实现的流程图。



1.2（并发实验）根据以下代码完成下面的实验。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"
Int main(int argc, char *argv[])
{
   if (argc != 2) {
      fprintf(stderr, "usage: cpu <string>\n");
      exit(1);
```

```
    }
    char *str = argv[1];
    while (1) {
        spin(1);
        printf("%s\n", str);
    }
    return 0;
}
```

1.2.1 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）

```
laybing@laybing-VirtualBox:~$ ./cpu
usage: cpu <string>
```

1.2.2 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。

```
laybing@laybing-VirtualBox:~$ ./cpu A & ./cpu B & ./cpu c & ./cpu D
[1] 4552
[2] 4553
[3] 4554
B
A
D
c
B
A
D
c
B
A
D
c
B
A
D
c
B
A
D
c
B
D
A
c
```

运行结果顺序与程序执行顺序不是对应的，并且每轮运行结果的顺序会有改变。

并发环境下，由于程序的封闭性被打破，程序与计算不再一一对应，一个程序副本可以有多个计算。

1.3（内存分配实验）根据以下代码完成实验。

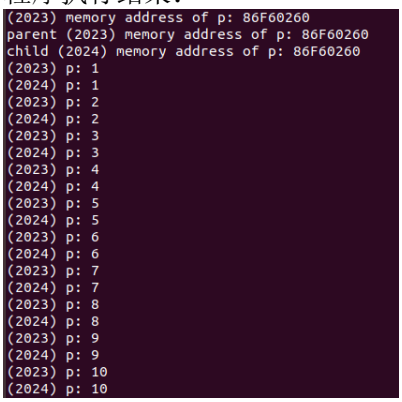
```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int)); // a1
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n",getpid(), p); // a2
    *p = 0; // a3
    while (1) {
        sleep(1);
        *p = *p + 1;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

```
    return 0;
}
```

1.3.1 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）
程序执行结果：

首先，通过 malloc()函数申请了部分内存。然后打印出了内存地址，再将数字 0 赋值给最新申请的内存地址中。最后，每延迟一秒，打印该程序的进程号，该进程号是唯一的，以及循环递增存储在该地址上的值。

1.3.2 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 &
程序执行结果：



每个进程都是从相同的地址开始分配内存，独立地去更新该地址的数值的。
操作系统中真正发生的事情是虚拟内存。每个进程都访问它们自己的私有的虚拟地址空间，操作系统将这些虚拟地址空间以某种方式映射到机器的物理内存。一个运行的程序引用的内存不会影响其他进程的地址空间；就正在运行的程序而言，它独自占有所有的物理内存。然而，事实却是物理内存是一个由操作系统管理的共享资源。

1.4（共享的问题）根据以下代码完成实验。

```
#include <stdio.h>
#include <stdlib.h>
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;
}
```

1.4.1 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 ）

```
laybing@laybing-VirtualBox:~$ ./thread 1000
Initial value : 0
Final value : 2000
```

程序结果输出两个 worker 函数循环中共享的计数器的增长次数。

1.4.2 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。）

```
laybing@laybing-VirtualBox:~$ ./thread 10000
Initial value : 0
Final value : 20000
laybing@laybing-VirtualBox:~$ ./thread 100000
Initial value : 0
Final value : 200000
laybing@laybing-VirtualBox:~$ ./thread 1000000
Initial value : 0
Final value : 2000000
laybing@laybing-VirtualBox:~$ ./thread 1000000000
Initial value : 0
Final value : 1996198727
laybing@laybing-VirtualBox:~$ ./thread 1000000000
Initial value : 0
Final value : 1995508327
```

不断增大输入的值，当输入的值为 1000000000，程序给出的结果不是 2000000000，而是 1995508327。又一次运行，我们不仅得到了一个不一样的错误的结果。

解释：上面程序共享的计数器递增的关键部分包涵三条指令：一个是从内存加载计数器的值到寄存器，一个时增加，一个是将它存入内存。因为这三条指令不是原子执行的（即一次执行所有），所有出了问题。

1.4.3 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。

全局变量是各个线程共享的，线程并发执行时访问共享变量会导致原子性问题，可见性问题，有序性问题等。