实验一：操作系统初步实验报告

计科 1601 班 马静雯 16281046

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

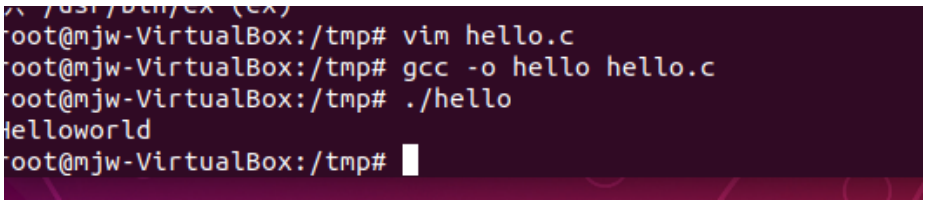
请问 getpid 的系统调用号是多少？39

linux 系统调用的中断向量号是多少？中断 0x80。

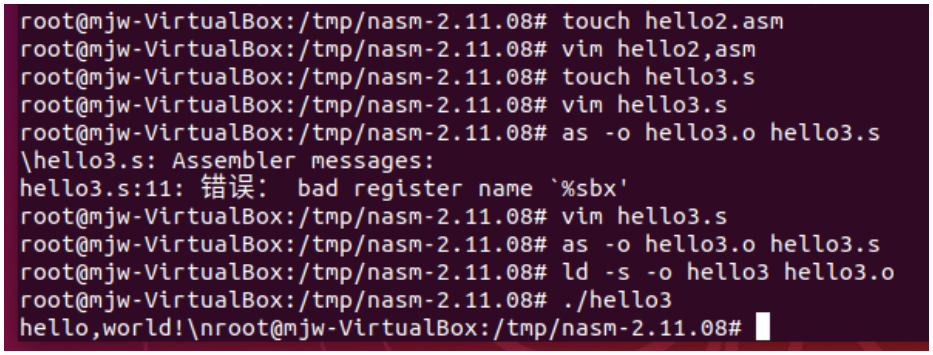
2、上机完成习题 1.13。

命令：printf(“Hello World!\n”)可归入一个 {C 标准函数、GNU C 函数库、Linux API} 中哪一个或者哪几个？请分别用相应的 linux 系统调用的 C 函数形式和汇编代码两种形式来实现上述命令。

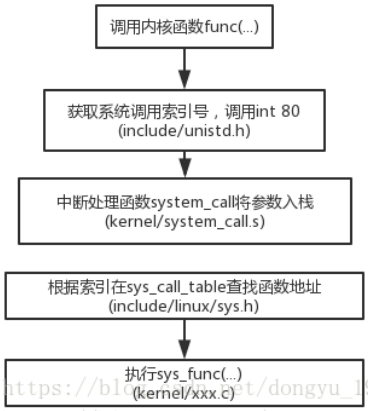
C 函数形式：



汇编形式：



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）

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。

事实上，它所做的是调用 Spin()，一个反复检查时间并一旦运行一秒钟就返回的函数。然后，它打印出用户在命令行上输入的字符串，并一直重复。将单个 CPU（或一小组 CPU）转换为看似无限数量的 CPU，从而允许许多程序看起来一次运行，这就是我们所谓的虚拟化 CPUCPU 运行了 4 次，运行的顺序是随机的。

```
root@mjw-VirtualBox:/tmp/nasm-2.11.08# ./hello3
hello,world!\nroot@mjw-VirtualBox:/tmp/nasm-2.11.08# touch 3.c
root@mjw-VirtualBox:/tmp/nasm-2.11.08# vim 3.c
root@mjw-VirtualBox:/tmp/nasm-2.11.08# gcc -o cpu 3.c -Wall
root@mjw-VirtualBox:/tmp/nasm-2.11.08# ./cpu
usage: cpu <string>
root@mjw-VirtualBox:/tmp/nasm-2.11.08# ./cpu A &;./cpu B & ; ./cpu C & ; ./cpu D &
bash: 未预期的符号 `;' 附近有语法错误
root@mjw-VirtualBox:/tmp/nasm-2.11.08# ./cpu A & ./cpu B & ./cpu C &./cpu D &
[1] 12374
[2] 12375
[3] 12376
[4] 12377
root@mjw-VirtualBox:/tmp/nasm-2.11.08# B
A
C
D
B
A
C
D
A
C
B
D
C
A
D
B
A
C
D
B
C
```

三、（内存分配实验）根据以下代码完成实验。

要求：

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)

```
root@mjw-VirtualBox:/tmp# touch mem.c
root@mjw-VirtualBox:/tmp# vim mem.c
root@mjw-VirtualBox:/tmp# gcc -o mem mem.c -Wall
mem.c: In function ‘main’:
mem.c:13:2: warning: implicit declaration of function ‘whhile’; did you mean ‘wr
ite’? [-Wimplicit-function-declaration]
  whhile(1)
  ^~~~~~
  write
mem.c:13:11: error: expected ‘;’ before ‘{’ token
  whhile(1)
           ^
           ;
  {
  ~
root@mjw-VirtualBox:/tmp# vim mem.c
root@mjw-VirtualBox:/tmp# gcc -o mem mem.c -Wall
root@mjw-VirtualBox:/tmp# ./mem
(12515)address pointed to by p: 0x5620d7878260
(12515) p: 1
(12515) p: 2
(12515) p: 3
(12515) p: 4
(12515) p: 5
```

程序功能:该程序做了几件事。首先,它分配一些内存。然后,它打印出内存地址(a2),然后将数字 0 放入新分配的内存的第一个位置。 最后，它循环：延迟一秒并递增存储在 p 中保存的地址的值。 对于每个 print 语句，它还会打印出正在运行的程序的进程标识符。 该 PID 在每个运行过程中都是唯一的。

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 &

```
root@mjw-VirtualBox:/tmp# ./mem & ./mem &
[1] 13635
[2] 13636
root@mjw-VirtualBox:/tmp# (13635)address pointed to by p: 0x55af6b165260
(13636)address pointed to by p: 0x558275abd260
(13635) p: 1
(13636) p: 1
(13635) p: 2
(13636) p: 2
(13636) p: 3
(13635) p: 3
(13636) p: 4
(13635) p: 4
(13636) p: 5
(13635) p: 5
(13635) p: 6
(13636) p: 6
(13636) p: 7
(13635) p: 7
```

实际上，这正是这里发生的事情，因为操作系统虚拟化了内存。 每个进程访问自己的私有虚拟地址空间（有时只称为其地址空间(address space)），操作系统以某种方式映射到机器的物理内存。一个正在运行的程序中的内存引用不会影响其他进程(或 OS 本身）的地址空间；就运行程序而言，它拥有所有的物理内存。 然而，现实是物理内存是由操作系统管理的共享资源。

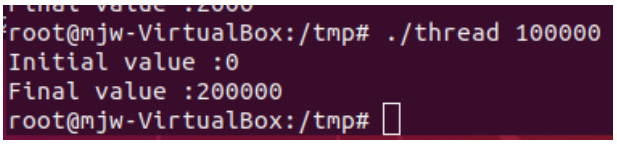
四、（共享的问题）根据以下代码完成实验。

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）

程序功能：主程序使用 Pthread .create()创建两个线程。 您可以将线程视为在与其他函数相同的内存空间中运行的函数，其中一次激活多个函数。

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。）



当两个线程完成时，计数器的最终值为 2000 和 200000，因为每个线程将计数器递增 1000 次和 100000 次。 实际上，当循环的输入值设置为 N 时，我们期望程序的最终输出为 2N