**实验一: 操作系统初步**

16281254 黄春浦 安全1601

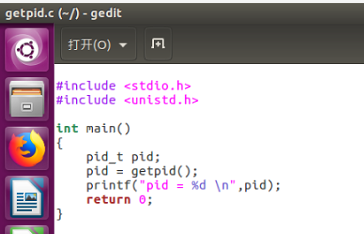
**一、（系统调用实验）了解系统调用不同的封装形式。要求如下：**

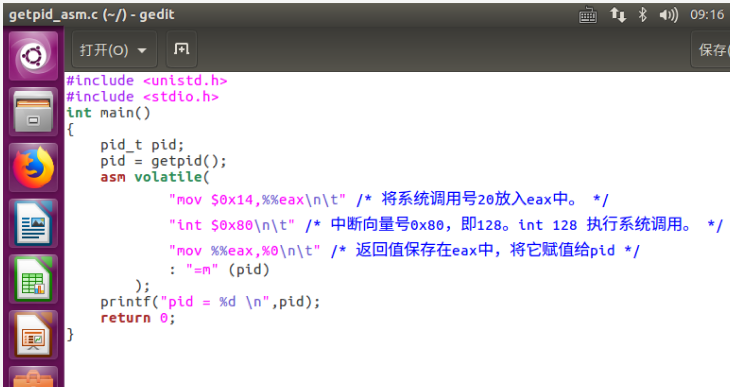
**问题1：**参考下列网址中的程序。阅读分别运行用API接口函数getpid()直接调用和汇编中断调用两种方式调用Linux操作系统的同一个系统调用getpid的程序(请问getpid的系统调用号是多少？linux系统调用的中断向量号是多少？)；

**答：**

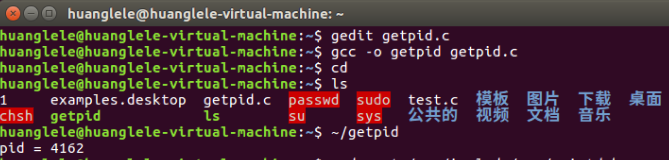
PID（Process Identification）：操作系统里指进程识别号，也就是进程标识符。操作系统里每打开一个程序都会创建一个进程ID，即PID。每个进程有唯一的PID编号，进程终止后PID标识符就会被系统回收，可能会被继续分配给新运行的程序。Getpid()函数可以返回进程识别码。

在自己的Linux系统中分别编写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的程序如下：

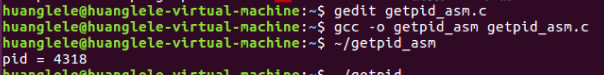




编译，并运行后可以得到API接口函数getpid()直接调用结果如下：

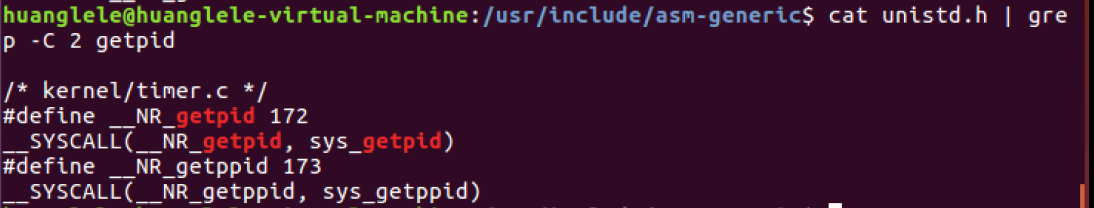


汇编中断调用结果如下：



**对问题1的解释：**

* 根据实验执行结果，程序成功完成了系统调用获取pid的操作，我们通过内嵌汇编代码可以清晰的看出调用**系统调用的工作过程**：首先将0x14（十进制20）赋值给eax寄存器，表示需要调用的系统调用号；然后，执行int 0x80（十进制128）来执行系统调用；之后eax寄存器保存了返回值，将它分别赋值给输出pid变量,随即完成整个汇编代码的系统调用，回到C代码中将pid输出；而C语言代码则是主要通过函数调用call来调用getpid()函数，该函数中封装了系统调用函数。
* 因此，我们可以得到，在getpid的**系统调用号**是20，linux系统调用的**中断向量号**是0x80(即128)；实际上，20是32位系统中getpid的系统调用号，64位中为39，在虚拟机上查找系统中的调用号，发现为172，如下：



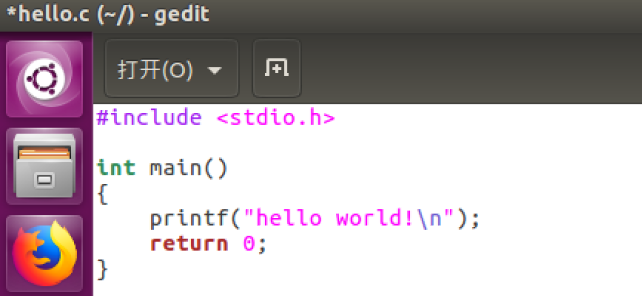
但是，我们的64位机中，兼容了32位机，并且不使用int 0x80进行中断，而是使用system\_call，在汇编代码中，由于汇编语言的编译器是32位的，故可以成功获得结果。

* **软中断int**的用法：系统程序需要核心态特权才能运行，此时用函数调用的办法是无法调用系统API函数的。解决这个问题的方法是使用软中断，当应用程序需要调用API时，就先设置功能号(如AX=0H)，然后触发软中断(如**INT 80H**)，根据系统程序设置好中断向量表。这样，应用程序就可以间接找到系统API了。
* 系统调用的过程：函数库提供的封装函数接口（API）->**system\_call**(是所有系统调用的入口，通过查看系统调用表(sys\_call\_table)找到所调用的内核函数入口地址)，这个入口会根据系统调用号（eax入栈）,调用对应的系统调用例程；
* **Int**与函数调用**call**的区别在于： 在系统调用中，使用使用INT和IRET指令，内核和应用程序使用的是不同的堆栈，因此存在堆栈的切换，从用户态切换到内核态，从而可以使用特权指令操控设备；而在函数调用中，使用CALL和RET指令，调用时没有堆栈切换。
* **系统调用：**Linux内核中设置了一组用于实现各种系统功能的子程序，称为系统调用。它是用户态进程访问硬件的一种方式，它通过中断（int 0x80）由用户态陷入内核态。用户可以通过系统调用命令在自己的应用程序中调用它们。系统调用由操作系统核心提供，运行于核心态；普通的函数调用由函数库或用户自己提供，运行于用户态。Linux核心提供了一些C语言函数库，习惯上把这些函数也称为系统调用。

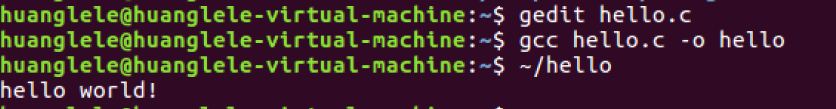
**问题2：**上机完成习题1.13；

**答：**

* 使用C语言在自己的Linux系统中编写习题1.13的程序hello.c如下：

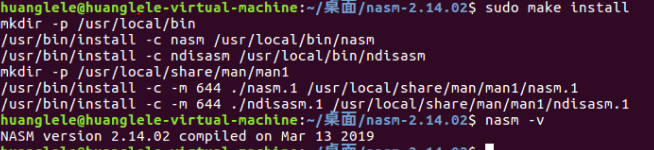


编译运行结果为在屏幕上打印出“hello world!”，如下：

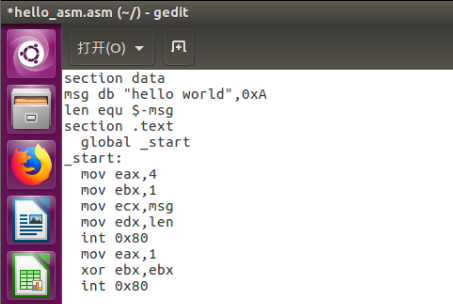


* 使用汇编语言

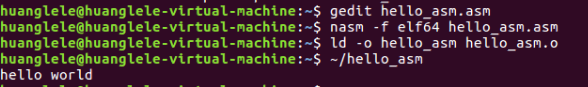
首先确认系统中安装了nasm汇编工具，如下：



在自己的Linux系统中编写习题1.13的程序源代码hello\_asm.asm如下



编译运行结果为在屏幕上打印出“hello world!”，其中，hello\_asm.o为汇编源码得到的对象文件，hello\_asm为链接到的可执行文件，如下：

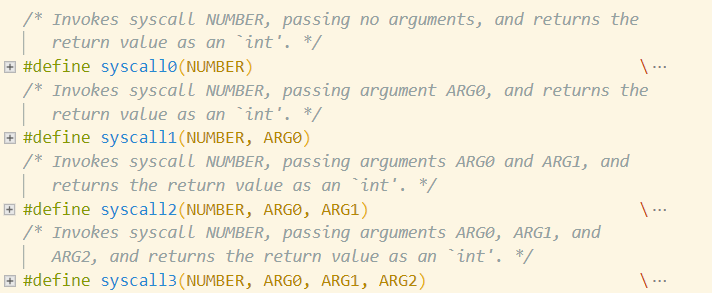


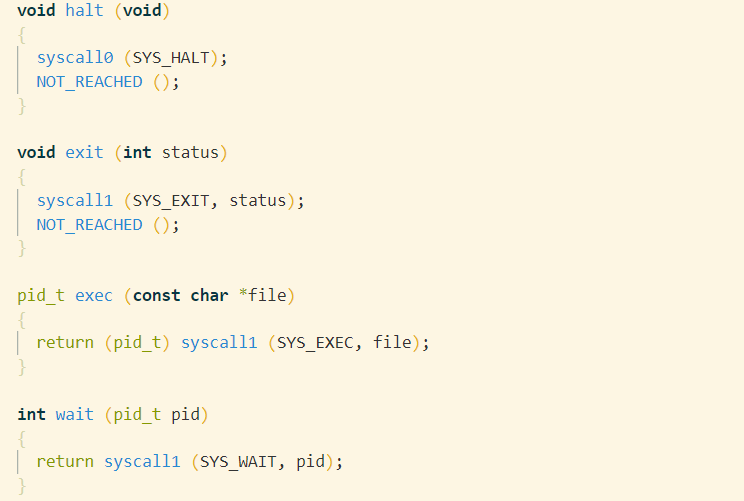
**问题3：**阅读pintos操作系统源代码，画出系统调用实现的流程图。

<http://hgdcg14.blog.163.com/blog/static/23325005920152257504165/>

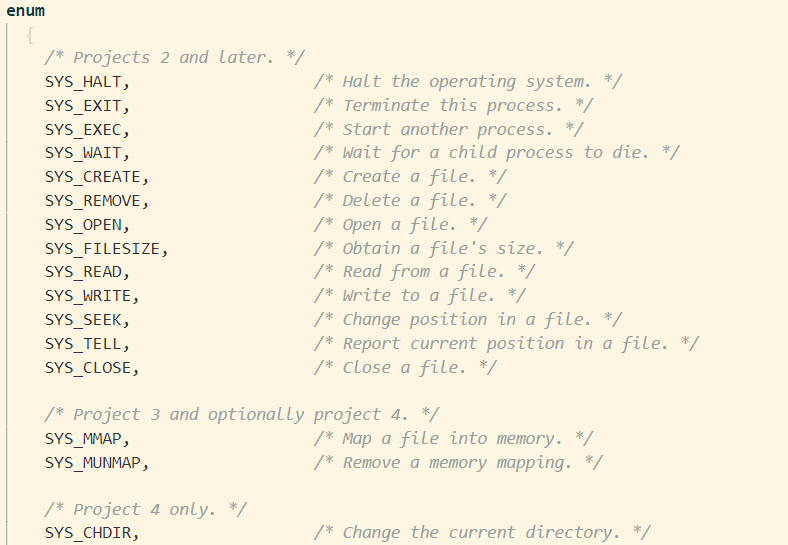
**答：**

* 题目中所给的网址无法打开，在github上可以搜到pintos的源码，链接：<https://github.com/codyjack/OS-pintos/tree/master/pintos/src>
* 经过对源码的阅读和分析发现：/src/lib/user/syscall.c中定义了 系统调用的四种方式以及20种系统调用函数（这里截取部分以作示例），且中断向量号为0x30如下：

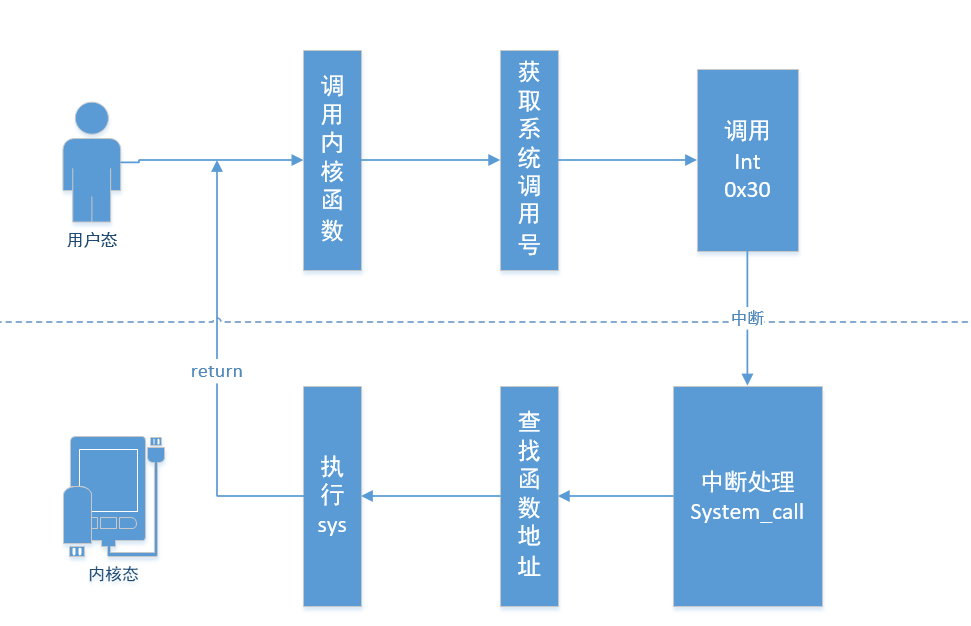




* 在文件/src/lib/syscallnr.h中，通过枚举定义了系统调用号，如下：



* 在文件/src/userprog/syscall.c中，函数syscall\_init 是负责系统调用初始化工作的，syscall\_handler 是负责处理系统调用的。syscall\_init 函数这个函数内部调用了 intr\_register\_int 函数，用于注册软中断从而调用系统调用处理函数。
* 系统调用实现的流程图如下：

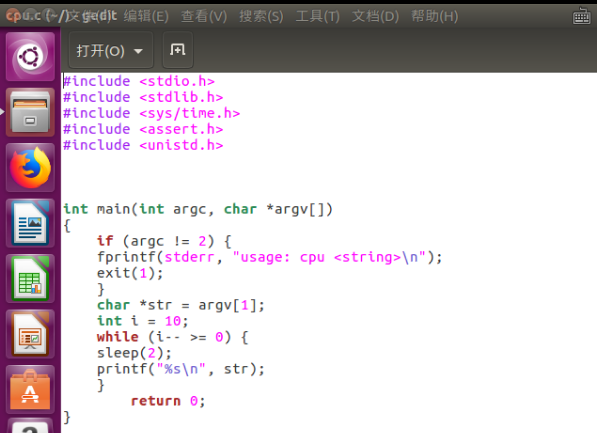


**二、（并发实验）根据以下代码完成下面的实验。**

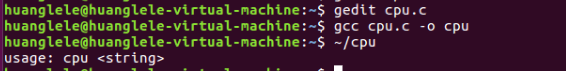
要求：

**问题1：**编译运行该程序（cpu.c），观察输出结果，说明程序功能。

* 在虚拟机中使用gedit命令编写代码cpu.c，该代码的含义是，循环输出传入参数到屏幕，中间通过 sleep（2）让程序执行过程中阻塞两秒。如果无参数则输出usage:cpu<string>，代码如下图：

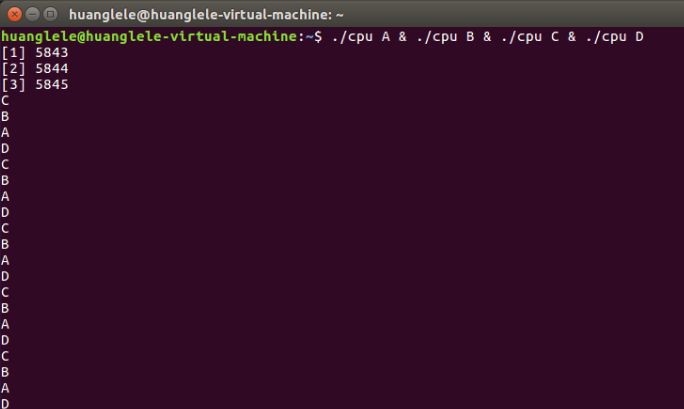


* 运行示例



**问题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。

* 执行命令以及结果如下：



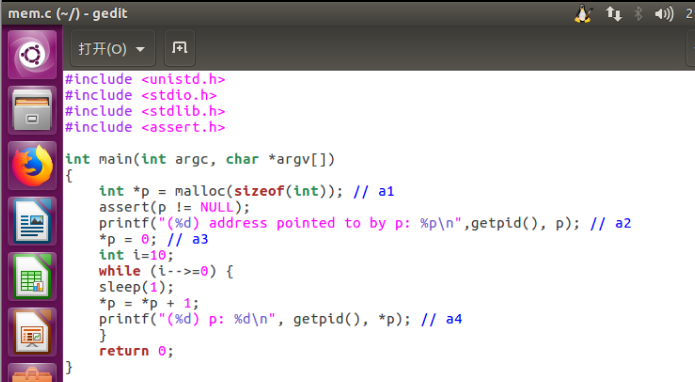
* 结果发现，程序运行了4次，4个程序同步进行，并无规律可循，且只要不手动停止，CPU就一直运行。
* 解释：操作系统的运行是并发实现的，进程不会严格按照先后顺序来执行。当进程执行时，遇到I/O操作（本题中的printf）就会进入等待状态，多进程运行时，等待是异步的，等待结束后，进入就绪状态然后再由系统调度执行，循环直至进程结束。另外系统的调度也会影响到进程的先后顺序，现代CPU一般是多核，故进程也可能是在多个CPU核心并行运行，也可能是两三个进程在一个核心中并发运行，综上，所以进程运行没有规律。

**三、（内存分配实验）根据以下代码完成实验。**

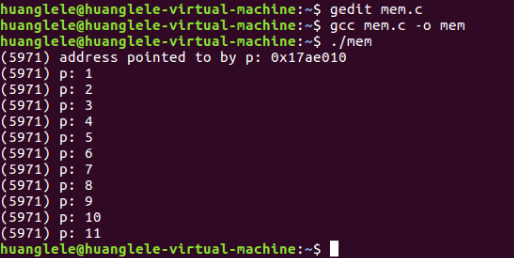
要求：

**问题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。

* 主要功能为: 先申请一个内存空间，打印进程号和内存地址；循环对分配好的内存地址进行累加。每累加一次进行一次 sleep，并输出新建进程的进程识别码。函数代码如下：

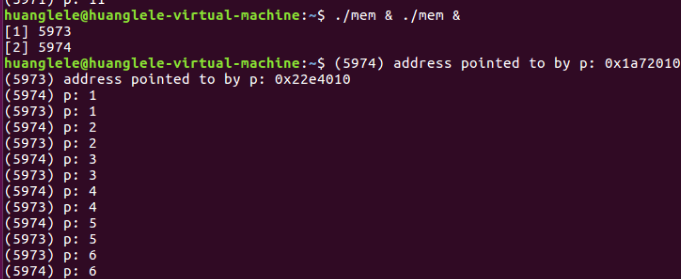


运行示例：

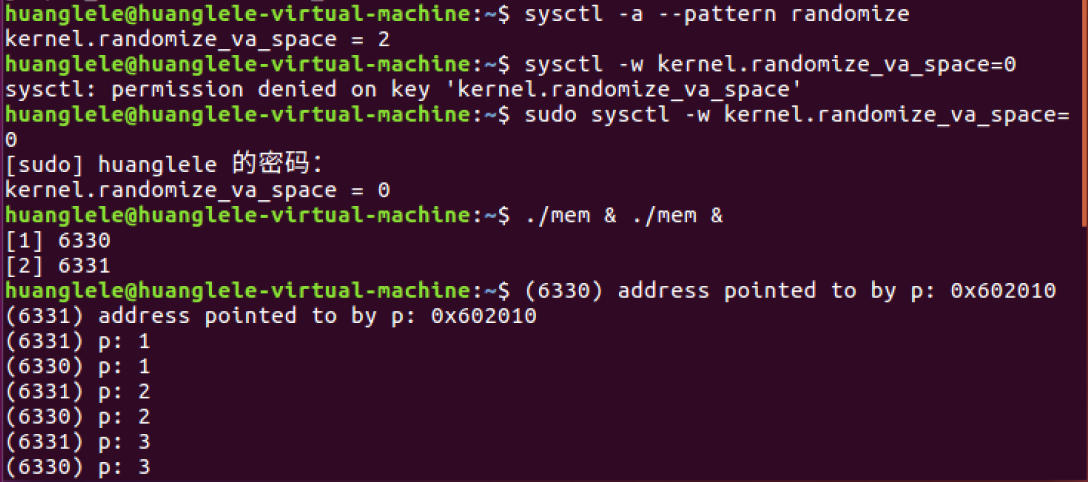


**问题2：**再次按下面的命令运行并观察结果。两个分别运行的程序分配的内存地址是否相同？是否共享同一块物理内存区域？为什么？

* 再次运行命令，结果如下：



* 当关闭了地址空间随机化后运行：



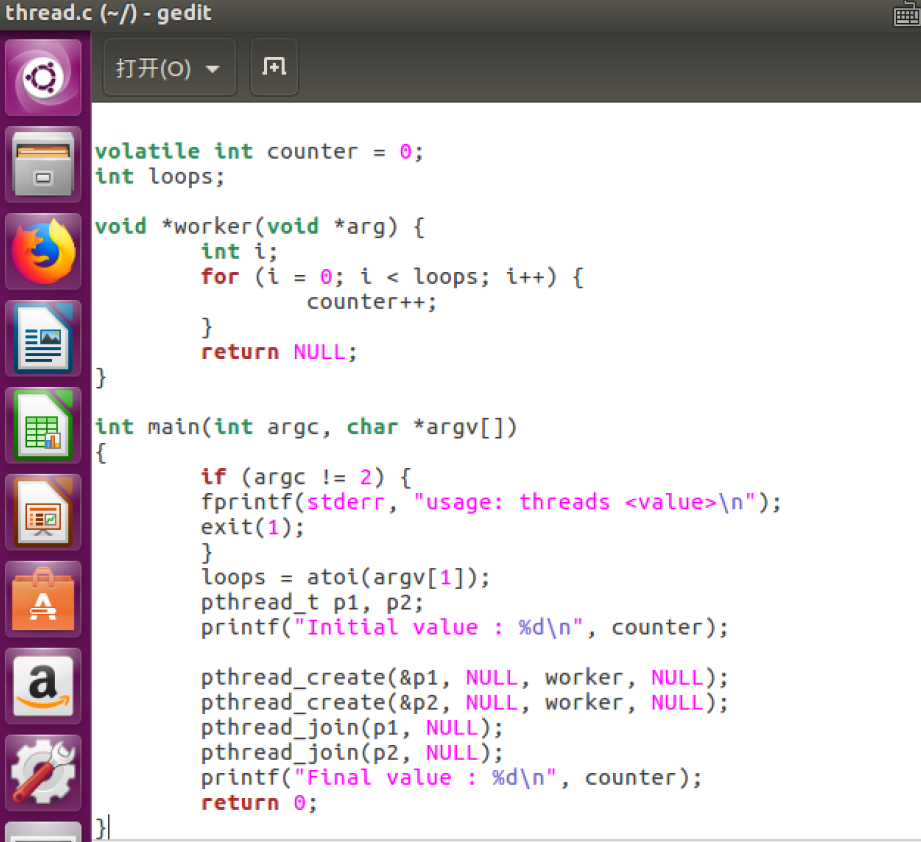
* 可以看到，在当前情况下，两个分别运行的程序的内存地址并不相同；且两个进程之间没有相互影响。
* 但是，每个进程的4G内存空间为虚拟内存空间，需要映射到实际物理内存地址，且所有进程共享同一物理内存区域，进程只需要用页表来记录这些映射即可。我们的函数中输出的是虚拟地址，进程之间享有独立的虚拟内存，故内存地址也可能会相同，但是其映射到的物理地址是不一样的，所以对数值的操作是没有影响的。
* 当关闭了地址空间随机化之后，在每个进程相当于有4G的独立虚拟内存，由于每个进程的程序是一模一样的，没有地址空间随机化的干扰，所以分配的地址也一样。
* **拓展：**ASLR(Address Space Layout Randomization)在2005年被引入到Linux的内核 kernel 2.6.12 中，当然早在2004年就以patch的形式被引入。随着内存地址的随机化，使得响应的应用变得随机。这意味着同一应用多次执行所使用内存空间完全不同，也意味着简单的缓冲区溢出攻击无法达到目的。配置设置中kernel.randomize\_va\_space 的值含义：“0” = 关闭；“1” = 半随机。共享库、栈、mmap() 以及 VDSO 将被随机化。（PIE会影响heap的随机化）；“2” = 全随机。

**四、（共享的问题）根据以下代码完成实验。**

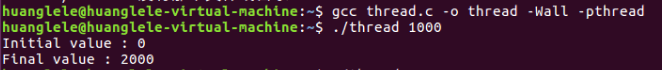
要求：

**问题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）

* 编辑thread.c代码如下图所示，其功能是在主函数创建两个线程，对counter进行累加操作，并输出初始值和累加后的值；

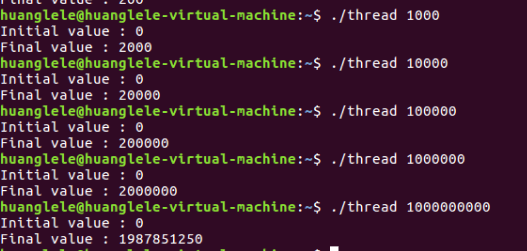


* 运行结果：



**问题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。

* 输入其他参数结果如下：



* 由实验结果，我们可以发现，在大多数情况下，Final value 是参数的2倍，initial value是0，但是，参数比较大的时候，Final value的值接近参数两倍，但不足两倍。
* 解释：counter，loops 这两个全局变量是线程共享的，线程并发执行时访问共享变量时，如果每个线程都拥有读写权限，就有可能读脏数据，例如当其中一个线程在进行累加操作时，另一个线程读入了过时的共享变量counter的数据，便会造成其中一次累加操作被覆盖掉。现代CPU采用了加锁的方法来良好地防止了这种问题的发生，在单核的情况下，是不会发生问题的，但是，由于参数逐渐增大，单核变为多核操作，就会发生这种问题。