**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：**

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？)；

2、上机完成习题1.13；

3、阅读pintos操作系统源代码，画出系统调用实现的流程图。

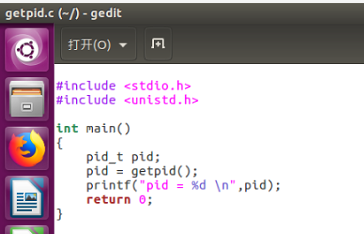
<http://hgdcg14.blog.163.com/blog/static/23325005920152257504165/>

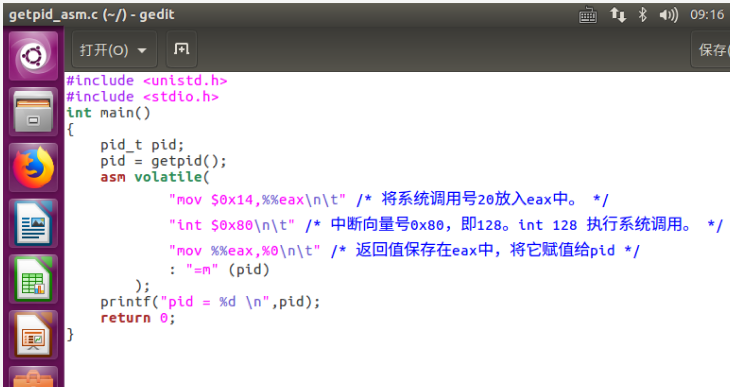
**解答：**

**问题1：**

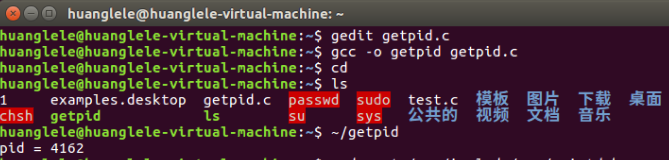
PID（Process Identification）：操作系统里指进程识别号，也就是进程标识符。操作系统里每打开一个程序都会创建一个进程ID，即PID。每个进程有唯一的PID编号，进程终止后PID标识符就会被系统回收，可能会被继续分配给新运行的程序。Getpid()函数可以返回进程识别码。

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：

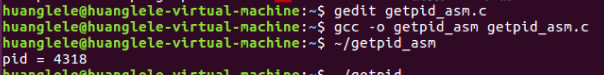




编译，并运行后可以得到API接口函数getpid()直接调用结果如下：

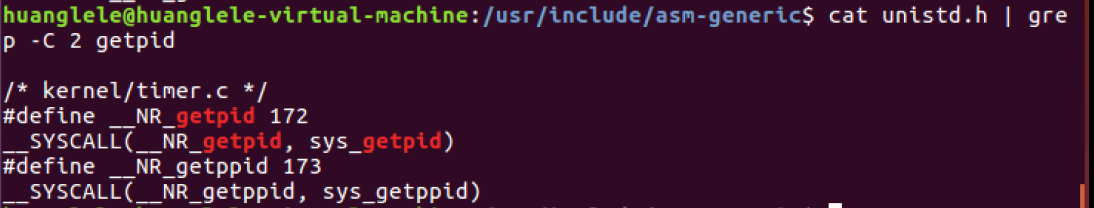


汇编中断调用结果如下：



**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；
* 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：

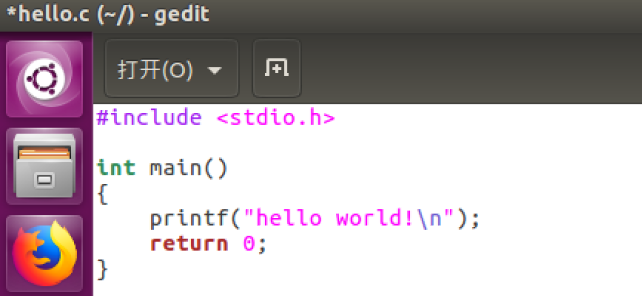


但是，我们的64位机中，兼容了32位机，并且不使用int 0x80进行中断，而是使用system\_call，在汇编代码中，由于汇编语言的编译器是32位的，故可以成功获得结果。

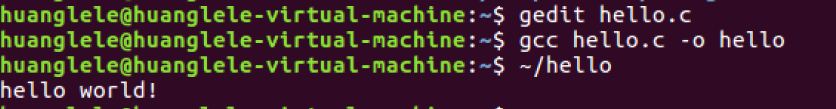
* **软中断int**的用法：系统程序需要核心态特权才能运行，此时用函数调用的办法是无法调用系统API函数的。解决这个问题的方法是使用软中断，当应用程序需要调用API时，就先设置功能号(如AX=0H)，然后触发软中断(如**INT 80H**)，根据系统程序设置好中断向量表。这样，应用程序就可以间接找到系统API了。
* 系统调用的过程：函数库提供的封装函数接口（API）->**system\_call**(是所有系统调用的入口，通过查看系统调用表(sys\_call\_table)找到所调用的内核函数入口地址)，这个入口会根据系统调用号（eax入栈）,调用对应的系统调用例程；
* **Int**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：**

* 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：

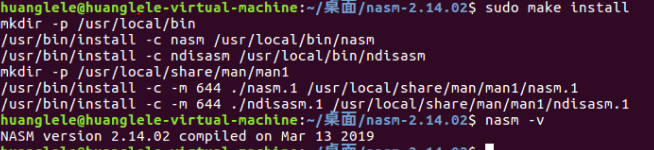


编译运行结果为在屏幕上打印出“hello world!”，如下：

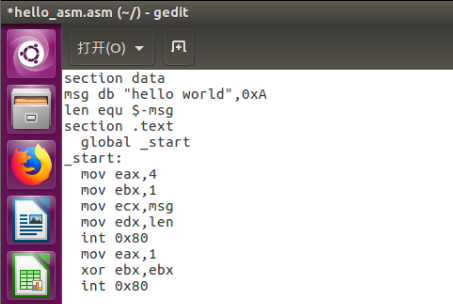


* 使用汇编语言

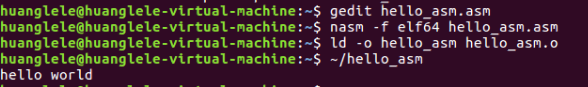
首先确认系统中安装了nasm汇编工具，如下：



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：**

* 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：

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。

1 #include <stdio.h>

2 #include <stdlib.h>

3 #include <sys/time.h>

4 #include <assert.h>

5 #include "common.h"

6

7 int

8 main(int argc, char \*argv[])

9 {

10 if (argc != 2) {

11 fprintf(stderr, "usage: cpu <string>\n");

12 exit(1);

13 }

14 char \*str = argv[1];

15 while (1) {

16 spin(1);

17 printf("%s\n", str);

18 }

1. eturn 0;

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)

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 &

1 #include <unistd.h>

2 #include <stdio.h>

3 #include <stdlib.h>

4 #include "common.h"

5

6 int

7 main(int argc, char \*argv[])

8 {

9 int \*p = malloc(sizeof(int)); // a1

10 assert(p != NULL);

11 printf("(%d) address pointed to by p: %p\n",

12 getpid(), p); // a2

13 \*p = 0; // a3

14 while (1) {

15 Spin(1);

16 \*p = \*p + 1;

17 printf("(%d) p: %d\n", getpid(), \*p); // a4

18 }

19 return 0;

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）
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。）
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。

1 #include <stdio.h>

2 #include <stdlib.h>

3 #include "common.h"

4

5 volatile int counter = 0;

6 int loops;

7

8 void \*worker(void \*arg) {

9 int i;

10 for (i = 0; i < loops; i++) {

11 counter++;

12 }

13 return NULL;

14 }

15

16 int

17 main(int argc, char \*argv[])

18 {

19 if (argc != 2) {

20 fprintf(stderr, "usage: threads <value>\n");

21 exit(1);

22 }

23 loops = atoi(argv[1]);

24 pthread\_t p1, p2;

25 printf("Initial value : %d\n", counter);

26

27 Pthread\_create(&p1, NULL, worker, NULL);

28 Pthread\_create(&p2, NULL, worker, NULL);

29 Pthread\_join(p1, NULL);

30 Pthread\_join(p2, NULL);

31 printf("Final value : %d\n", counter);

32 return 0;