**实验一: 操作系统初步**

# 本次实验全部基于Ubuntu 16.04完成

## 一、系统调用实验

了解系统调用不同的封装形式。

1. 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？

**Answer**：

1. 将两个文件分别命名为getpid\_c.c和getpid\_assembly.c，首先用gcc编译API接口调用程序，执行结果如下：

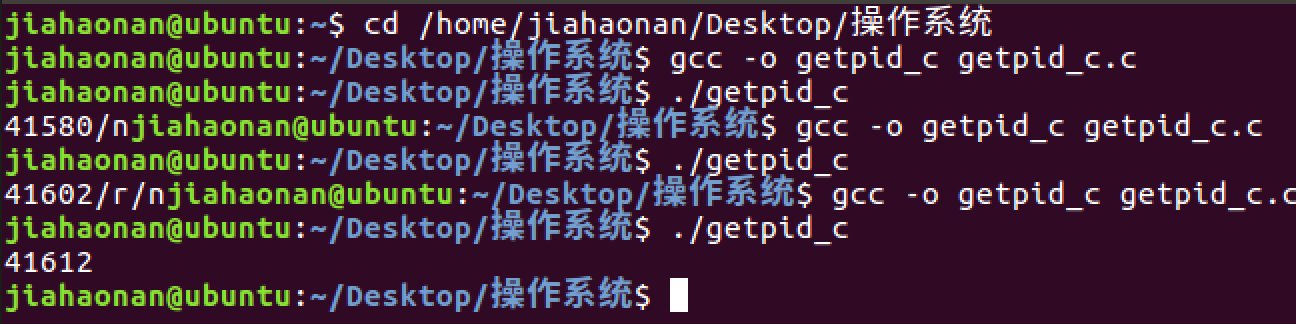


图 1 API接口调用

可以看到程序分别输出了41580、41602、41612，其代表每次执行程序的进程识别码，且进程相互不同。

其次用gcc编译汇编中断调用程序，其执行结果如下：

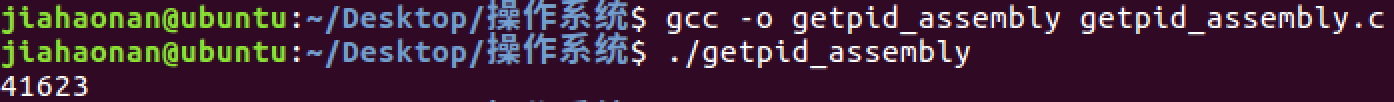


图 2 汇编中断调用

程序的执行结果与API调用实现相同，可以实现出书进程的识别码。

1. 查阅有关博客，getpid的系统调用号分两个：在32位系统下为20，在64位系统下为39.

本题的代码基于32位系统，所以程序中传入的系统调用号位0x14，即20.

1. 根据第二段汇编内嵌代码所提供的信息，其通过代码：**INT 0x80** 进入中断，所以Linux系统调用的中断向量号是80H。
2. 上机完成习题1.13**。**

**Answer**.

1. linux系统调用的C函数形式：

为了实现打印输出“hello world”，这里选用系统调用函数write来实现，通过下面代码即可实现：

#include <stdio.h>

int main()

{

write(1,"hello world!\n",14);

return 0;

}

其中write的三个参数分别制定：输出方式、输出内容和输出长度。

执行结果如下：

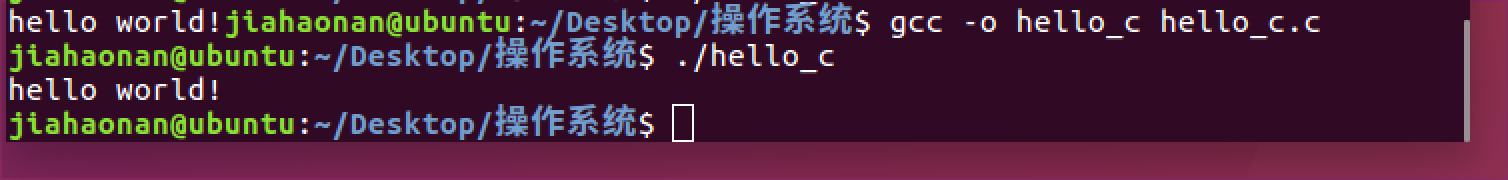


图 3 系统调用C函数形式

1. 汇编代码：

通过C语言内嵌汇编的形式来打印“hello word”的方法如下：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{

char\* msg = "Hello World!\n\r";

int len = 14;

int result = 0;

asm volatile("mov %2, %%edx;\n\r" /\*传入参数：要显示的字符串长度\*/

"mov %1, %%ecx;\n\r" /\*传入参赛：文件描述符（stdout）\*/

"mov $1, %%ebx;\n\r" /\*传入参数：要显示的字符串\*/

"mov $4, %%eax;\n\r" /\*系统调用号：4 sys\_write\*/

"int $0x80" /\*触发系统调用中断\*/

:"=m"(result) /\*输出部分：本例并未使用\*/

:"m"(msg),"r"(len) /\*输入部分：绑定字符串和字符串长度变量\*/

:"%eax");

return 0;

}

程序的执行结果如下：

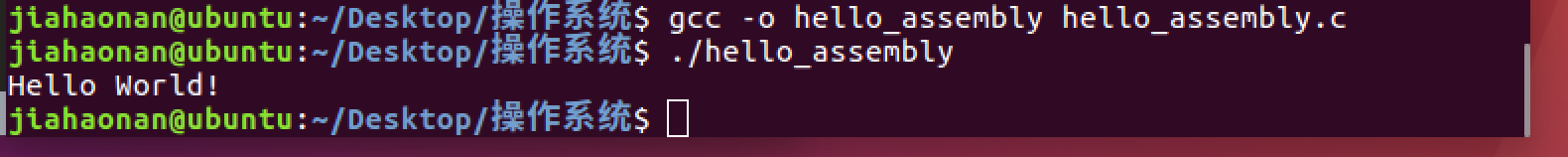


图 4 汇编语言形式

1. 阅读pintos操作系统源代码，画出系统调用实现的流程图。

**Answer**.

由于并未实际操作过pintos系统，所以如何定位系统调用的区域或是执行流程是一个很大的问题。为此，我参考了Linux的系统调用流程，将Linux的系统调用思想转移到pintos系统上，下面是我的学习所得。

先拿Linux的系统调用read函数来举个例子，其实现流程如下：

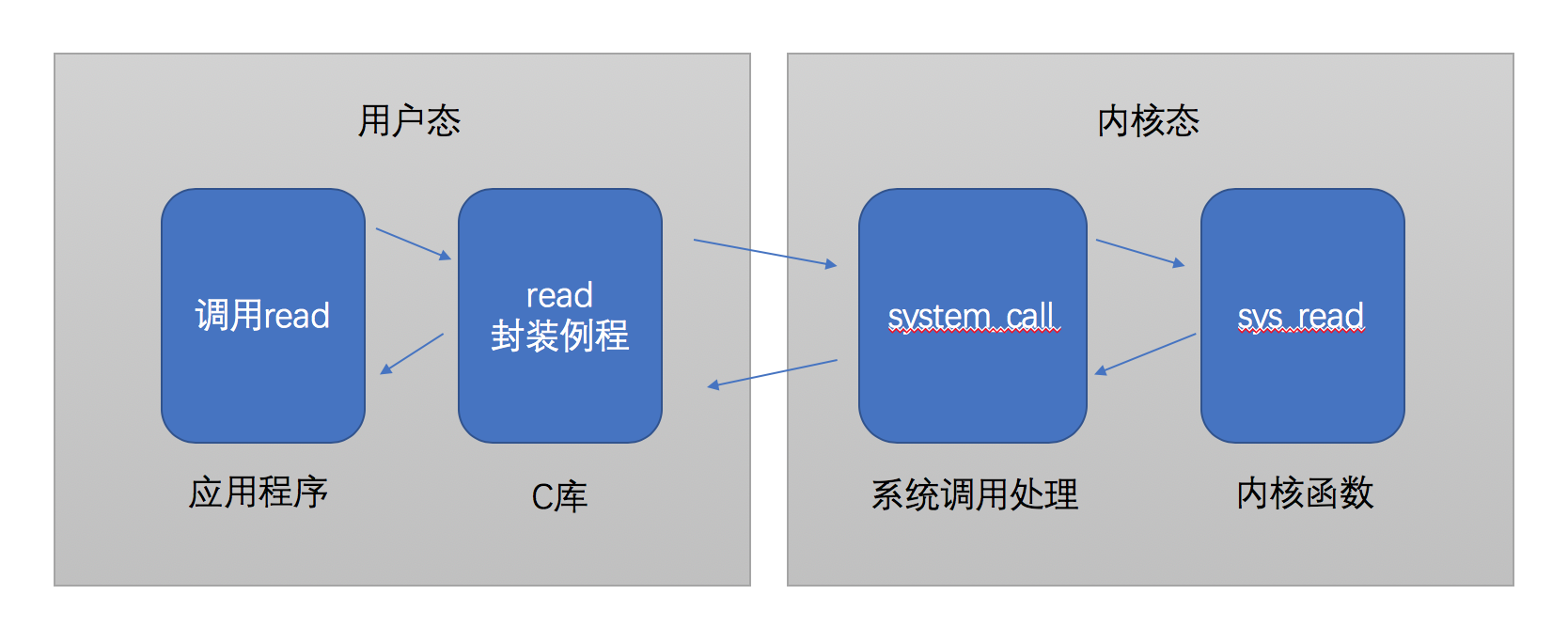


图 5 Linux系统调用

在理解了Linux系统调用后，我尝试对pintos的项目源码进行了研读。

Pintos系统调用的相关代码存储在：pintos\src\lib\user下的syscall.c、syscall.h和syscall-nr.h中。其中：syscall-nr.h用于存储系统调用号的声明，syscall.c和syscall.h则编写了系统调用函数的具体实现。

还是继续拿read()函数来说明pintos的系统调用流程：

* 首先用户在用户程序中调用read函数。
* 接下来进入read函数的封装例程阶段，其对应syscall.c中的代码段：

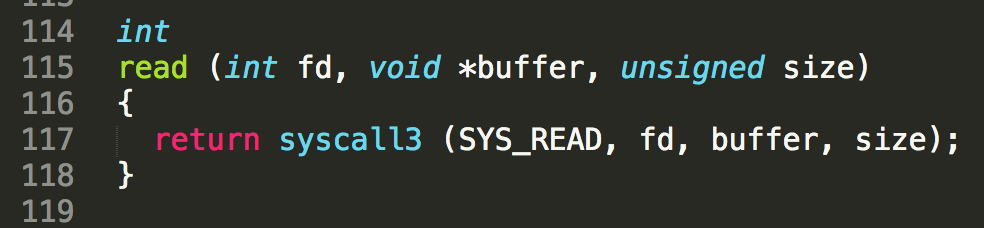


图 6 read函数

* 在C库的read函数中，其返回了一个syscall3 (SYS\_READ, fd, buffer, size);的指向，即系统调用处理程序。其中第一个参数SYS\_READ即为read函数的系统调用号，该声明位于文件syscall-nr.h中：

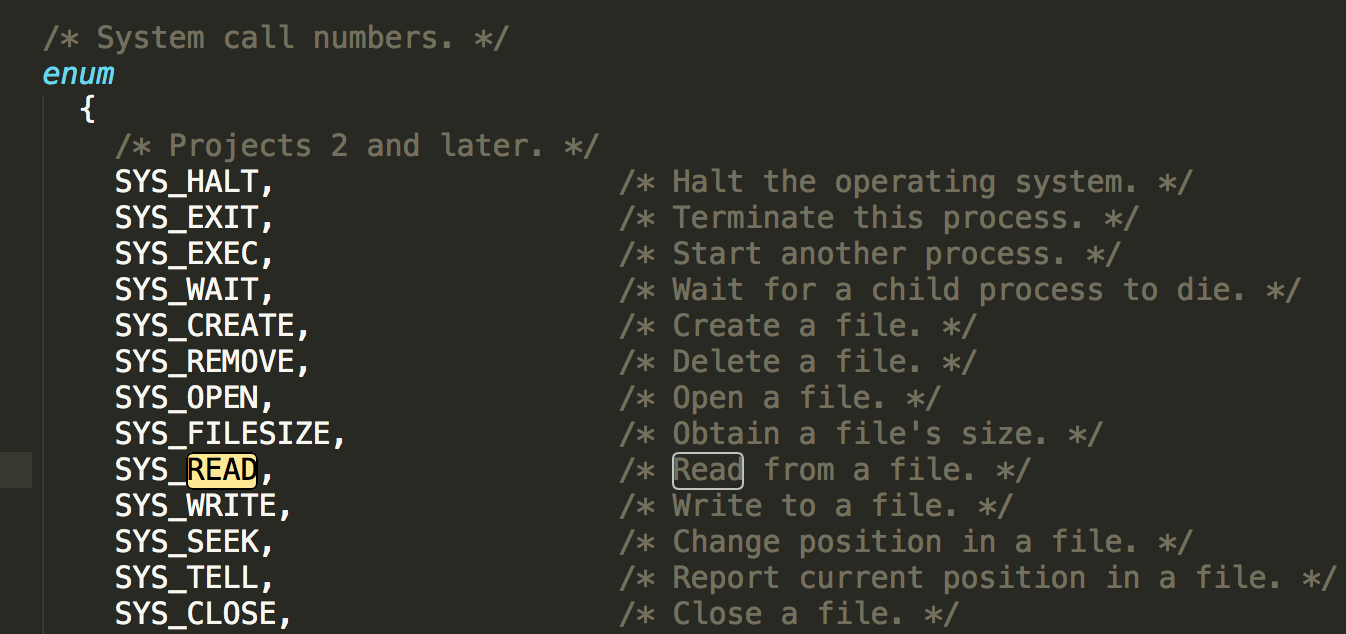


图 7 系统调用号

* Syscall3函数的实现也位于syscall.c文件中，其代码如下：

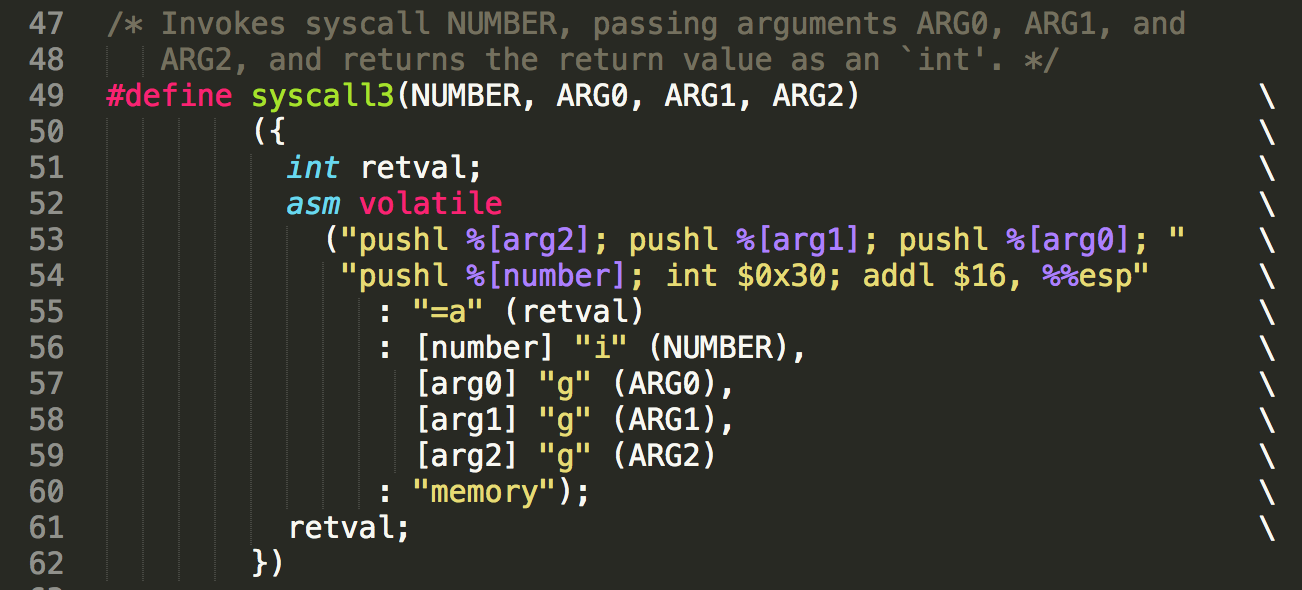


图 8 syscall3

从代码中可以看到，syscall3函数通过**INT $0x30**语句触发了系统的软中断，从而得以调用系统调用服务例程。

所以，pintos的系统调用流程应该为：

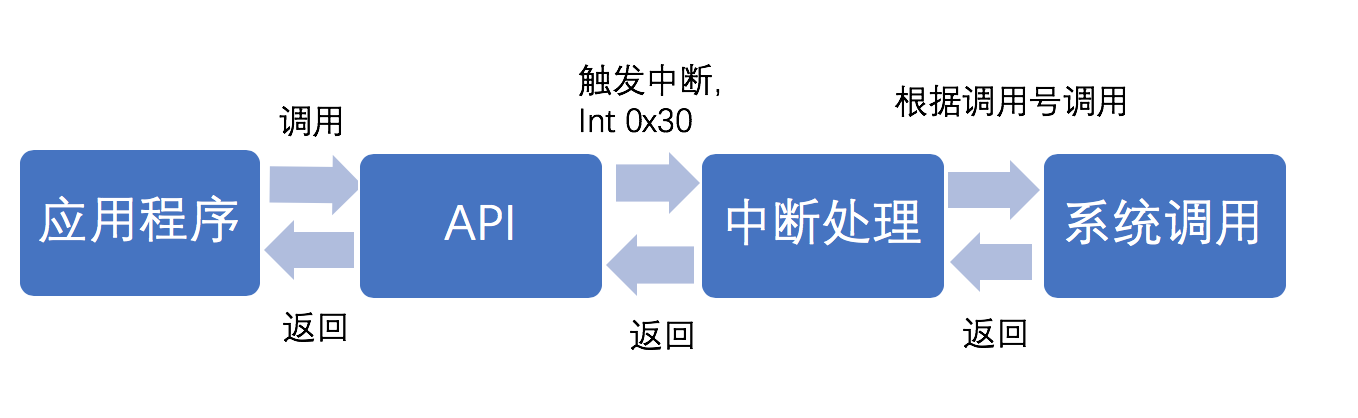


图 9 系统调用流程

## 二、并发实验

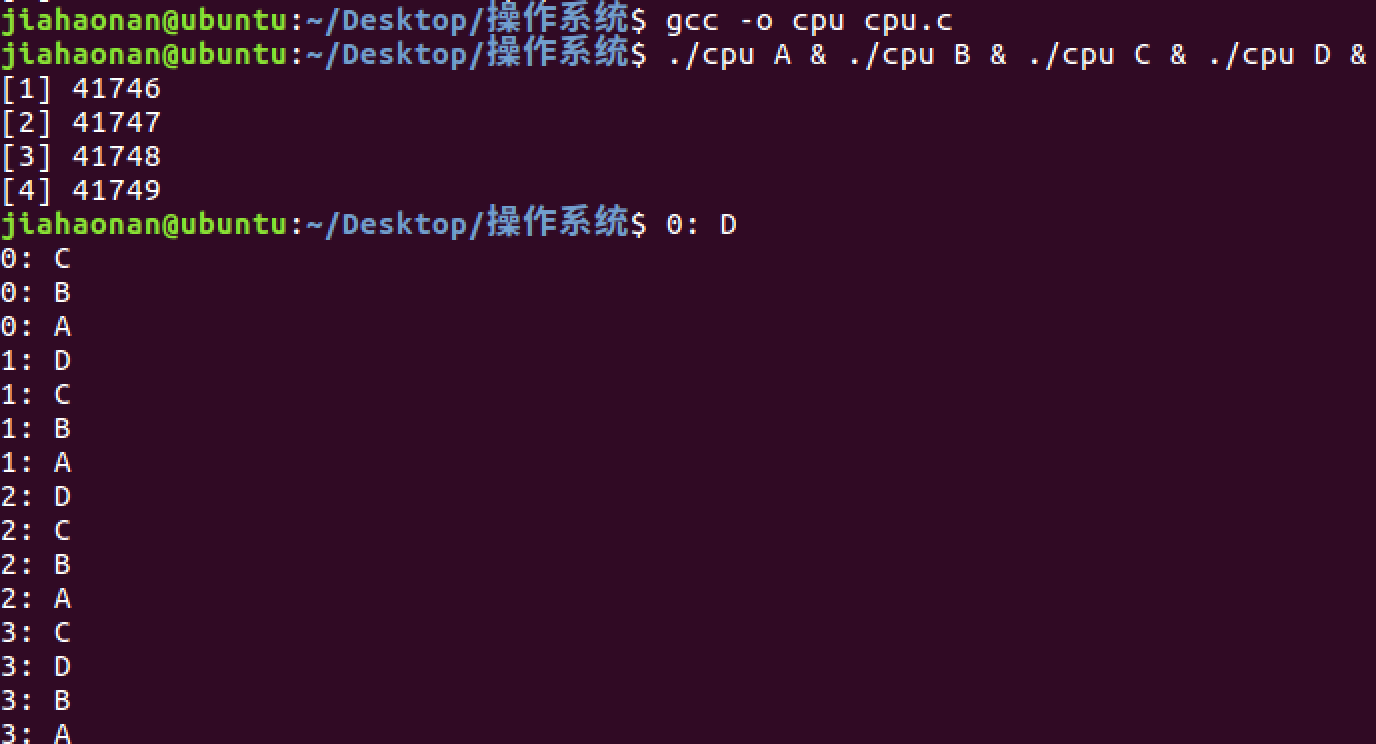
1. 编译运行该程序（cpu.c），观察输出结果，说明程序功能。(编译命令： gcc -o cpu cpu.c –Wall)（执行命令：./cpu）

**Answer**. 通过观察，可以得出程序的功能为：每隔一秒打印输入的参数或提示程序的正确输入格式。

1. 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。

**Answer**.

1. 观察程序可知，CPU的运行次数取决于while循环的执行次数，如果不强制终止程序，CPU将会一直运行下去。
2. 程序的部分执行结果如下：



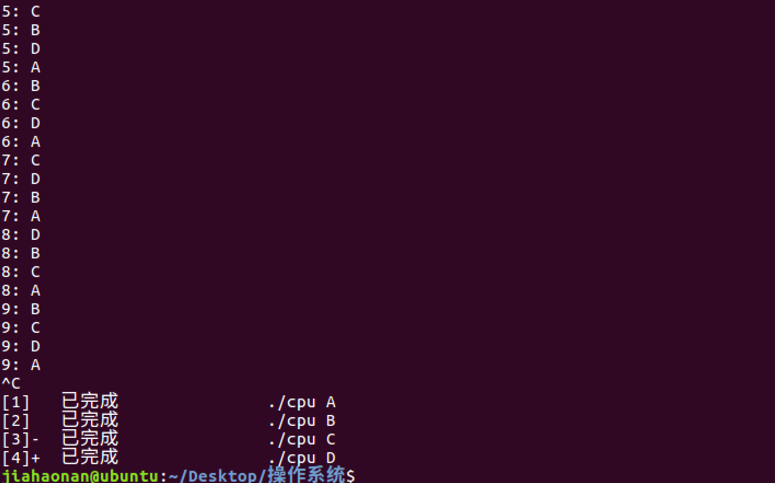


图 10 同时执行多个cpu.c的结果

从上图的执行结果中可以观察到：4个程序在每一轮都会轮流执行，而且执行的打印次序没有固定的顺序。

这是因为：对于4个完全相同的程序而言，CPU的优先级是相同的，所以每一轮的执行顺序是随机的，没有特定的规律。

## 三、内存分配实验

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)

**Answer**. 程序的执行结果如图：

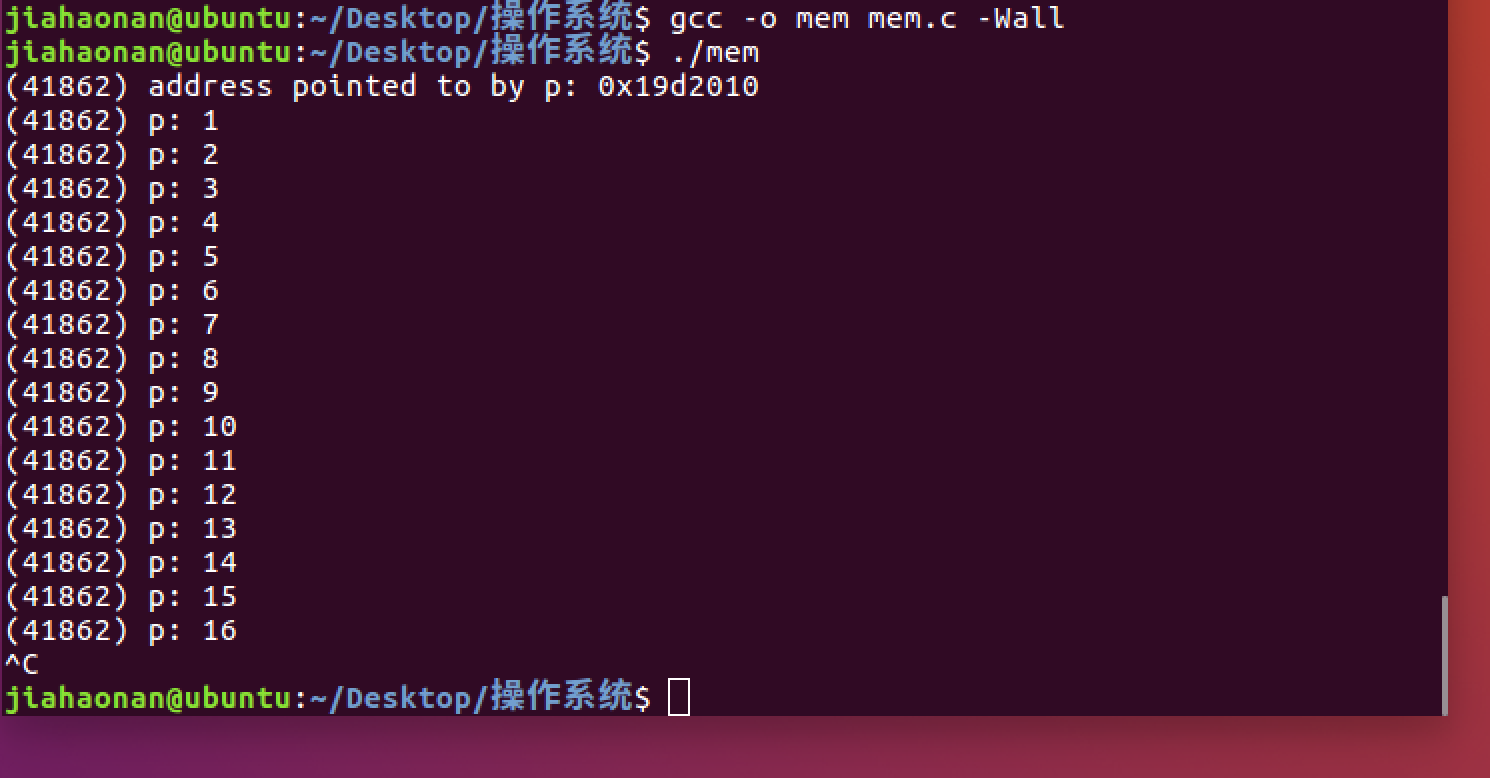


图 11 mem.c执行结果

程序首先打印指针p所指向的内存地址：0x19d2010及其对应的进程：41862 。

其次程序每隔一秒打印指针p+1后的相对地址1、2、3… 及其对应的进程：41862 。

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 &

**Answer**. 调用两个进程同时执行程序mem.c，其结果如下：



图 12 ./mem & ./mem

从上图可以看到，两个分别运行的程序分配的内存地址显然不同，两个指针分别指向内存：0x8a2010和0x2337010.

因为指针指向的内存单元的地址不同，所以不共享同一块物理内存区域。

原因：操作系统为不同的指针分配不同的地址，同时为了避免指针之间的冲突，所以不能把不同的指针指向同一内存区域。

## 四、共享的问题

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）

**Answer**. 程序执行的输出结果如图：



图 13 ./thread 1000执行结果

阅读代码可知，程序的功能为开启两个线程，同时执行子函数worker，实现对counter变量的loops\*2次累加，最后打印counter的初值和累加结果。

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。）

**Answer**. 更改参数，程序执行的输出结果如图：



图 14 更改参数输出结果

从输出结果可以发现，counter的final值为输入参数的2倍，初始值恒为0.

原因：虽然程序同时创建了两个线程执行worker函数，但两个线程均是对同一个counter变量进行相同次数的累加，所以最终结果总为loops的2倍，即输入参数argc的2倍。