**实验一: 操作系统初步**

## 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？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。

答：

1.运行结果如下图所示，





现在来对比分析两种调用方式，现在来分析代码，第一个程序pid 用来保存使用API 函数getpid（）返回的进程号，而第二个程序pid保存嵌入式汇编代码使用系统调用号调用系统调用得到的进程号。分析汇编语言，“mov $0x14,%%eax\n\t”表示将20（getpid系统调用号为20）传进eax寄存器，“int $0x80\n\t”发出系统调用（软中断）指令。“mov %%eax,%0\n\t”

将eax寄存器的值（其中保存的是系统调用返回值）存入变量pid。通过对比两种方式，我们可以看出API 和系统调用是不同的，API只是一个函数定义但系统调用通过软中断向内核发出一个明确的请求，经过查询资料得到以下信息：

* API指的是我们用户程序编程调用的如read(),write(),malloc(),free()之类的调用的是glibc库提供的库函数。API直接提供给用户编程使用，运行在用户态。这里要另外提一下，POSIX针对API提出标准，即针对API的函数名，返回值，参数类型进行规范约束，但是并不管API具体如何实现。
* 操作系统为用户态进程与硬件设备进行交互提供了一组接口——系统调用。通过软中断或系统调用指令向内核发出一个明确的请求，内核将调用内核相关函数来实现（如sys\_read(),sys\_write()）。用户程序不能直接调用这些Sys\_read,sys\_write等函数。这些函数运行在内核态。

2.C标准函数库是所有符合标准的头文件的集合，以及常用的函数库实现程序，例如I/O输入输出和字符串控制。glibc是Linux下面C标准库的实现，即GNU C Library。Glibc本身是GNU旗下的C标准库，后来逐渐成为了Linux的标准C库。而Linux API用于保证应用程序可以在源程序一级上在多种操作系统上移植运行，这些系统调用编程接口主要是通过C库（LIBC）来实现的。我认为Linux API通过调用glibc函数库来实现所需的功能，而其本身并不包括这些函数，因此printf(“Hello World!\n”)可归入C标准函数和GNU C函数库。

C函数：

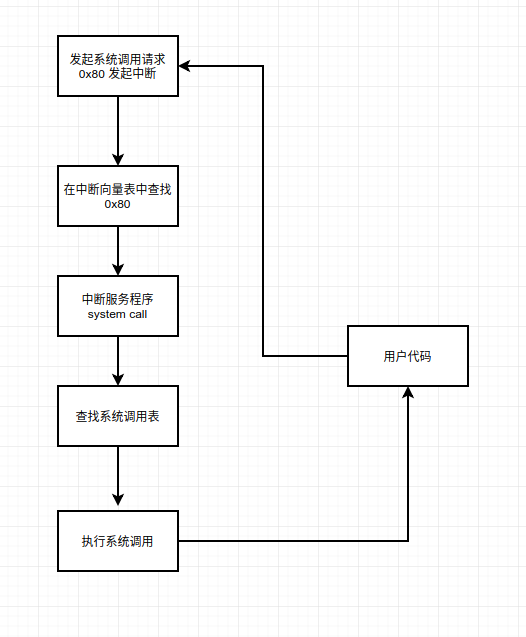




汇编：



3.阅读pintos操作系统源代码，系统调用实现如下图：



## 二、（并发实验）根据以下代码完成下面的实验。

要求：

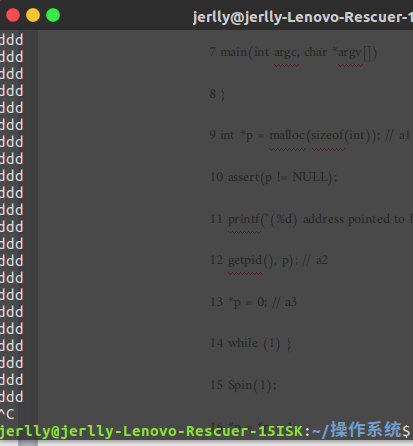
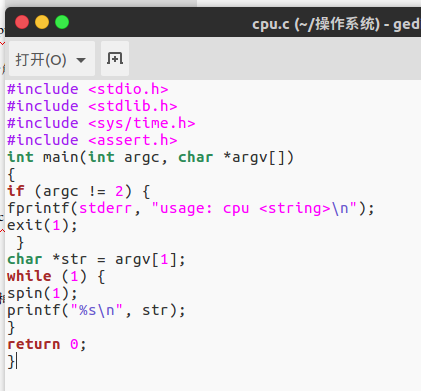
1. 编译运行该程序（cpu.c），观察输出结果，说明程序功能。

(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。

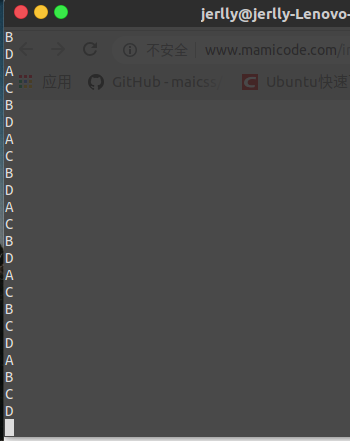
答：

1.



可以看到，它打印出用户在命令行上输入的字符串，并一直重复。代码将打印用户传入的输入字符串，然后继续，程序将永远运行; 只有按“Control-c”我们才能暂停程序。

2.

可以看到，即使我们只有一个处理器，但是四个cpu程序在同时进行，运行并没有明显的先后顺序与规律，这可以用操作系统中的并发行来解释，在操作系统中，并发是指一个时间段中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个处理机上运行，但任一个时刻点上只有一个程序在处理机上运行。所以虽然看起来是同时在运行四个cpu程序，但每一个时刻只有一条指令在执行，并且执行结束与执行开始之间并无先后关系。

## 三、（内存分配实验）根据以下代码完成实验。

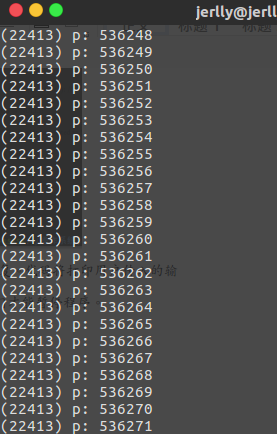
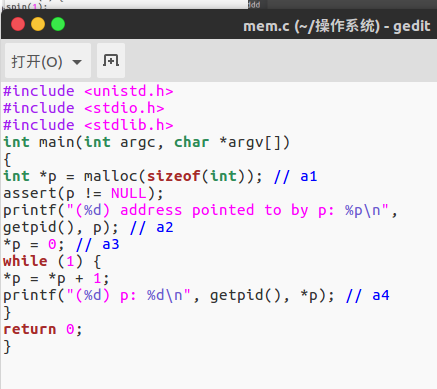
要求：

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 &

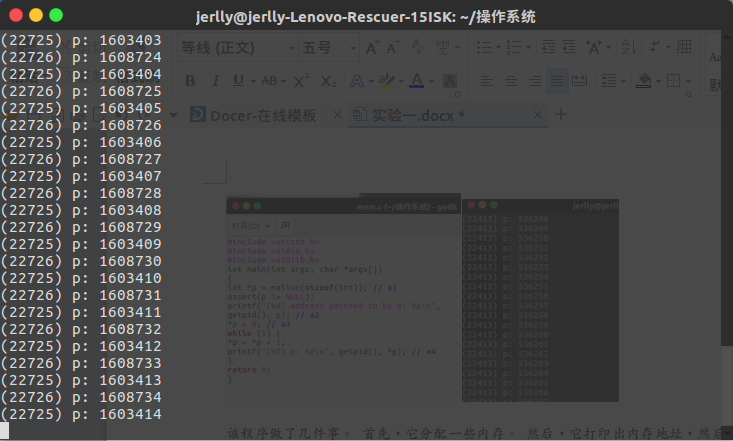
答：

1.



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，然后将数字0放入新分配的内存的第一个位置。 最后，它循环并递增存储在p中保存的地址的值。 对于每个printf语句，它还会打印出正在运行的程序的进程标识符（PID）。 该PID在每个运行过程中都是唯一的。 程序运行时，会慢慢更新值并打印出结果。

2.



我们可以看到，每个正在运行的程序都在同一地址分配了内存，但每个程序似乎都在独立更新这个地址的值。通过查阅资料可以知道，这是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。

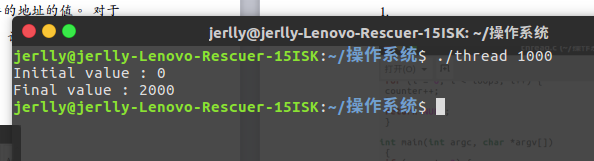
要求：

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）
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。）

答：

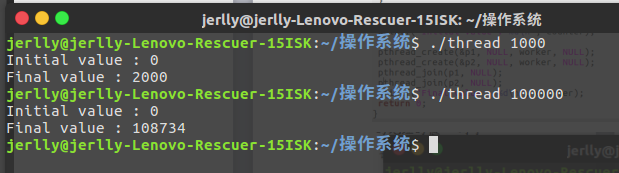
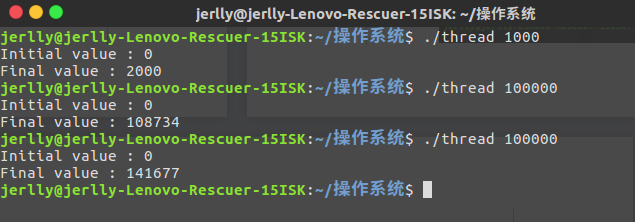
1.





通过观察和阅读代码，主程序使用pthread .create()创建两个线程。在这个例子中，每个线程开始在一个名为worker()的例程中运行，在该例程中，它只是循环递增一个统计循环次数的计数器。当两个线程完成时，计数器的最终值为2000，因为每个线程将计数器递增1000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。

2.

在这次运行中，当我们给出100,000的输入值，而不是最终值为200,000时，我们首先获得108734。 然后，当我们第二次运行程序时，不仅会再次获得错误的值，而且还会获得与上次不同的值。通过查询资料，这些奇怪和不寻常的结果的原因与指令的执行方式有关。上面程序的一个关键部分共享计数器递增，这部分需要三个指令：一个用于将计数器的值从存储器加载到寄存器中，一个用于递增它，一个用于将其存储回内存。 因为这三个指令不是一次全部执行完毕，所以才会发生这样的现象。