

实验一：操作系统初步

16281254 黄春浦 安全 1601

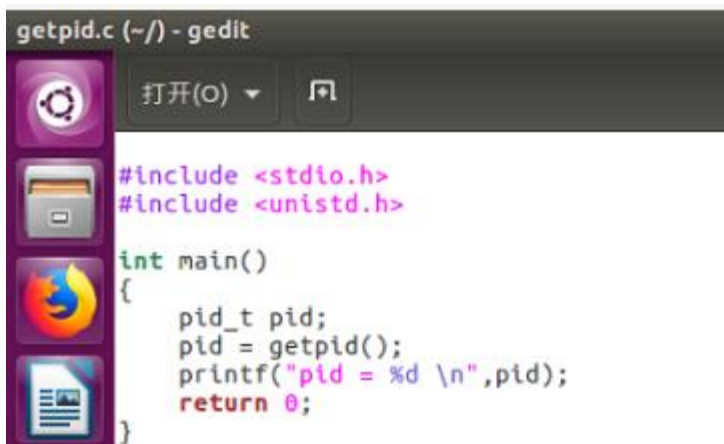
一、（系统调用实验）了解系统调用不同的封装形式。要求如下：

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：



```
getpid.c (-/) - gedit
打开(O)  图标
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    pid = getpid();
    printf("pid = %d \n",pid);
    return 0;
}
```

```
getpid_asm.c (~/) - gedit
打开(O) 保存

#include <unistd.h>
#include <stdio.h>
int main()
{
    pid_t pid;
    pid = getpid();
    asm volatile(
        "mov $0x14,%%eax\n\t" /* 将系统调用号20放入eax中。 */
        "int $0x80\n\t" /* 中断向量号0x80, 即128。int 128 执行系统调用。 */
        "mov %%eax,%0\n\t" /* 返回值保存在eax中, 将它赋值给pid */
        : "=r" (pid)
    );
    printf("pid = %d \n",pid);
    return 0;
}
```

编译, 并运行后可以得到 API 接口函数 getpid()直接调用结果如下:

```
huanglele@huanglele-virtual-machine: ~
huanglele@huanglele-virtual-machine:~$ gedit getpid.c
huanglele@huanglele-virtual-machine:~$ gcc -o getpid getpid.c
huanglele@huanglele-virtual-machine:~$ cd
huanglele@huanglele-virtual-machine:~$ ls
1  examples.desktop  getpid.c  passwd  sudo  test.c  模板  图片  下载  桌面
chsh  getpid  ls  su  sys  公共的  视频  文档  音乐
huanglele@huanglele-virtual-machine:~$ ~/getpid
pid = 4162
```

汇编中断调用结果如下:

```
huanglele@huanglele-virtual-machine:~$ gedit getpid_asm.c
huanglele@huanglele-virtual-machine:~$ gcc -o getpid_asm getpid_asm.c
huanglele@huanglele-virtual-machine:~$ ~/getpid_asm
pid = 4318
huanglele@huanglele-virtual-machine:~$
```

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:

```

huanglele@huanglele-virtual-machine:/usr/include/asm-generic$ cat unistd.h | gre
p -C 2 getpid

/* kernel/timer.c */
#define __NR_getpid 172
__SYSCALL(__NR_getpid, sys_getpid)
#define __NR_getppid 173
__SYSCALL(__NR_getppid, sys_getppid)

```

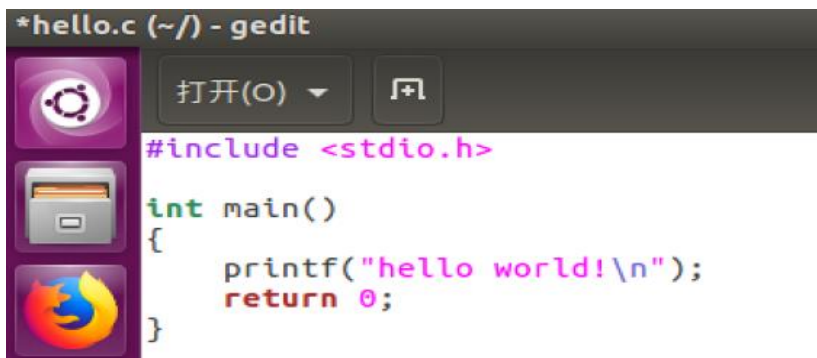
但是，我们的 64 位机中，兼容了 32 位机，并且不使用 int 0x80 进行中断，而是使用 system_call，在汇编代码中，由于汇编语言的编译器是 32 位的，故可以成功获得结果。

- **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：



```
*hello.c (~/) - gedit
#include <stdio.h>

int main()
{
    printf("hello world!\n");
    return 0;
}
```

编译运行结果为在屏幕上打印出“hello world!”, 如下:

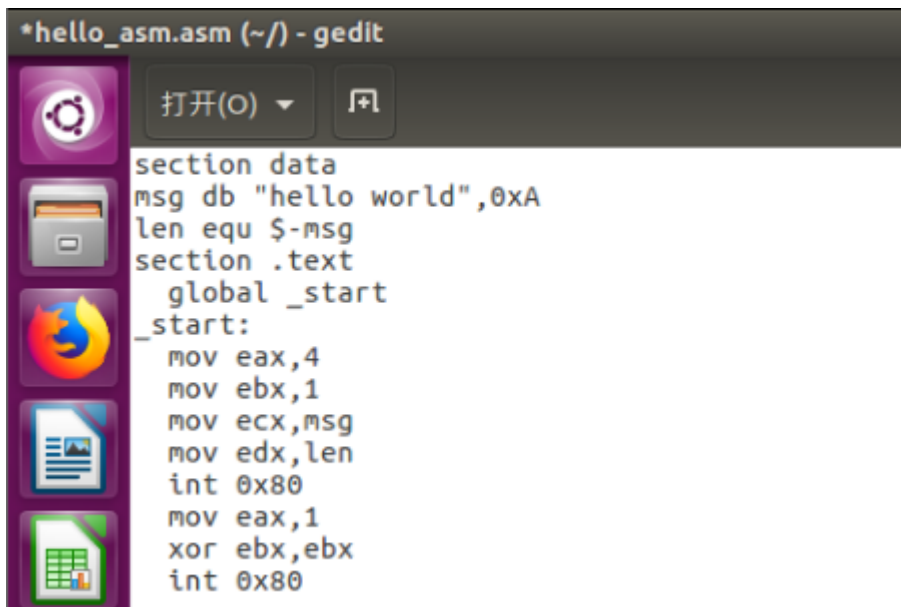
```
huanglele@huanglele-virtual-machine:~$ gedit hello.c
huanglele@huanglele-virtual-machine:~$ gcc hello.c -o hello
huanglele@huanglele-virtual-machine:~$ ./hello
hello world!
```

➤ 使用汇编语言

首先确认系统中安装了 nasm 汇编工具, 如下:

```
huanglele@huanglele-virtual-machine:~/桌面/nasm-2.14.02$ sudo make install
mkdir -p /usr/local/bin
/usr/bin/install -c nasm /usr/local/bin/nasm
/usr/bin/install -c ndisasm /usr/local/bin/ndisasm
mkdir -p /usr/local/share/man/man1
/usr/bin/install -c -m 644 ./nasm.1 /usr/local/share/man/man1/nasm.1
/usr/bin/install -c -m 644 ./ndisasm.1 /usr/local/share/man/man1/ndisasm.1
huanglele@huanglele-virtual-machine:~/桌面/nasm-2.14.02$ nasm -v
NASM version 2.14.02 compiled on Mar 13 2019
```

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



```
*hello_asm.asm (~/) - gedit
section data
msg db "hello world",0xA
len equ $-msg
section .text
    global _start
_start:
    mov eax,4
    mov ebx,1
    mov ecx,msg
    mov edx,len
    int 0x80
    mov eax,1
    xor ebx,ebx
    int 0x80
```

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:

```

huanglele@huanglele-virtual-machine:~$ gedit hello_asm.asm
huanglele@huanglele-virtual-machine:~$ nasm -f elf64 hello_asm.asm
huanglele@huanglele-virtual-machine:~$ ld -o hello_asm hello_asm.o
huanglele@huanglele-virtual-machine:~$ ~/hello_asm
hello world

```

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），且中断向量为 0x30 如下：

```

/* Invokes syscall NUMBER, passing no arguments, and returns the
| return value as an `int'. */
+ #define syscall0(NUMBER)                                     \ ...
/* Invokes syscall NUMBER, passing argument ARG0, and returns the
| return value as an `int'. */
+ #define syscall1(NUMBER, ARG0)                               \ ...
/* Invokes syscall NUMBER, passing arguments ARG0 and ARG1, and
| returns the return value as an `int'. */
+ #define syscall2(NUMBER, ARG0, ARG1)                         \ ...
/* Invokes syscall NUMBER, passing arguments ARG0, ARG1, and
| ARG2, and returns the return value as an `int'. */
+ #define syscall3(NUMBER, ARG0, ARG1, ARG2)                   \ ...

void halt (void)
{
| syscall0 (SYS_HALT);
| NOT_REACHED ();
}

void exit (int status)
{
| syscall1 (SYS_EXIT, status);
| NOT_REACHED ();
}

pid_t exec (const char *file)
{
| return (pid_t) syscall1 (SYS_EXEC, file);
}

int wait (pid_t pid)
{
| return syscall1 (SYS_WAIT, pid);
}

```

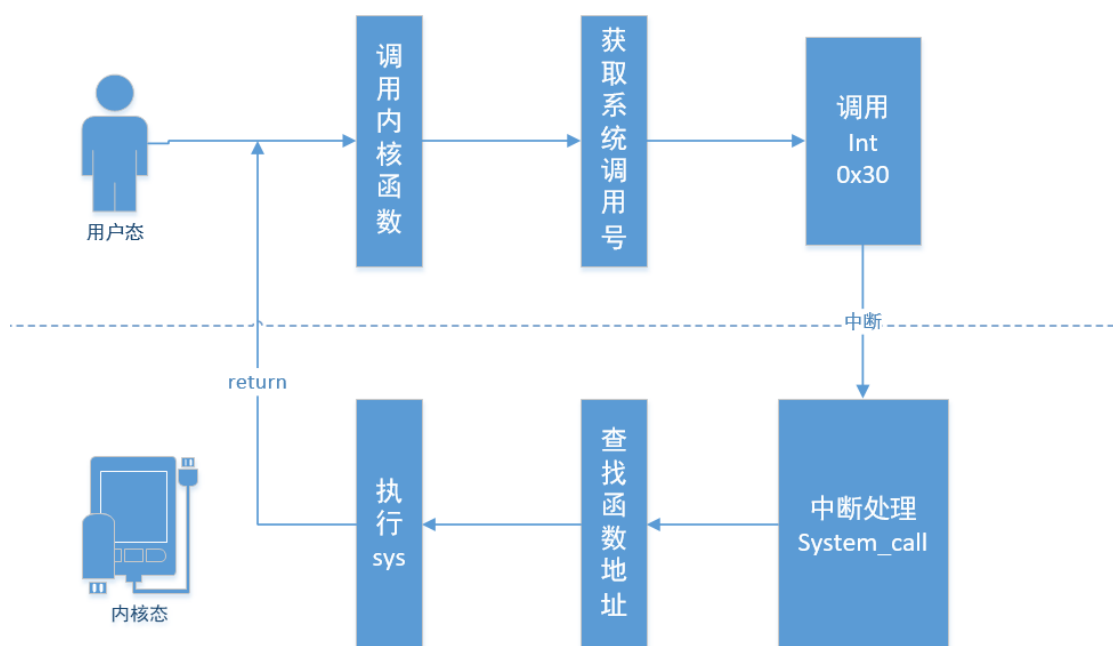
- 在文件/src/lib/syscallnr.h 中，通过枚举定义了系统调用号，如下：

```
enum
{
    /* Projects 2 and later. */
    SYS_HALT,           /* Halt the operating system. */
    SYS_EXIT,           /* Terminate this process. */
    SYS_EXEC,           /* Start another process. */
    SYS_WAIT,           /* Wait for a child process to die. */
    SYS_CREATE,         /* Create a file. */
    SYS_REMOVE,         /* Delete a file. */
    SYS_OPEN,           /* Open a file. */
    SYS_FILESIZE,       /* Obtain a file's size. */
    SYS_READ,           /* Read from a file. */
    SYS_WRITE,          /* Write to a file. */
    SYS_SEEK,           /* Change position in a file. */
    SYS_TELL,           /* Report current position in a file. */
    SYS_CLOSE,          /* Close a file. */

    /* Project 3 and optionally project 4. */
    SYS_MMAP,           /* Map a file into memory. */
    SYS_MUNMAP,         /* Remove a memory mapping. */

    /* Project 4 only. */
    SYS_CHDIR,          /* Change the current directory. */
}
```

- 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：

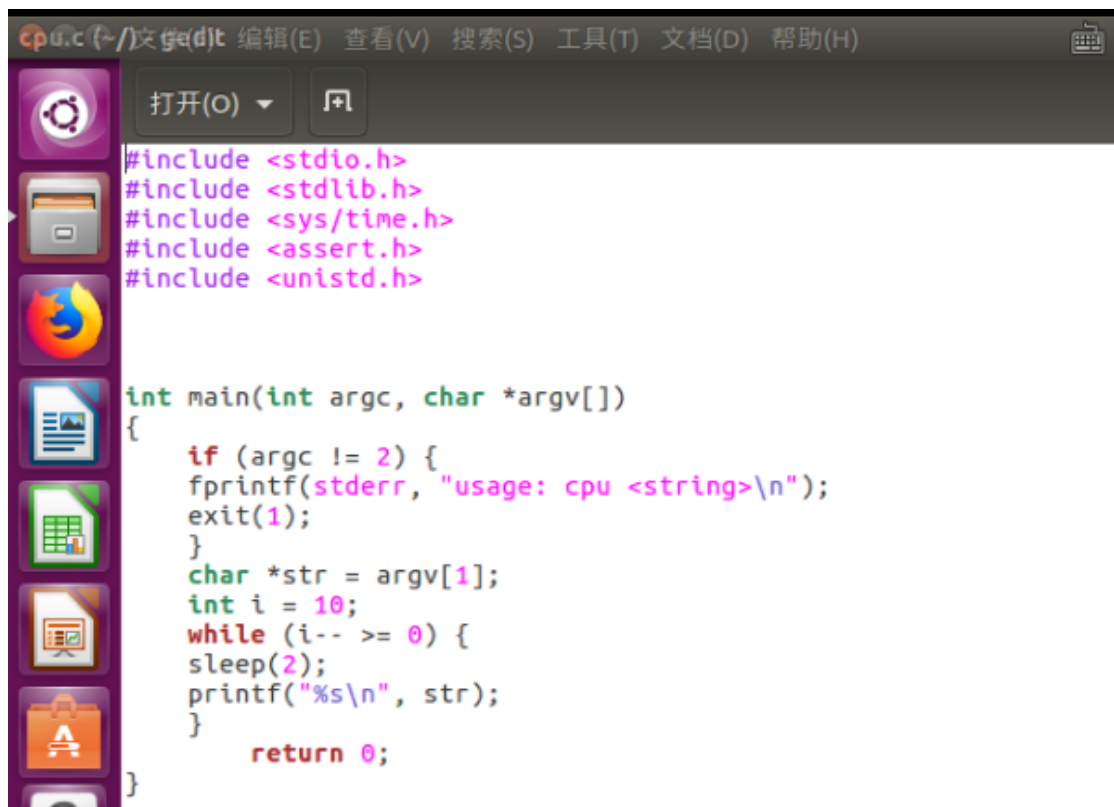


二、（并发实验）根据以下代码完成下面的实验。

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。

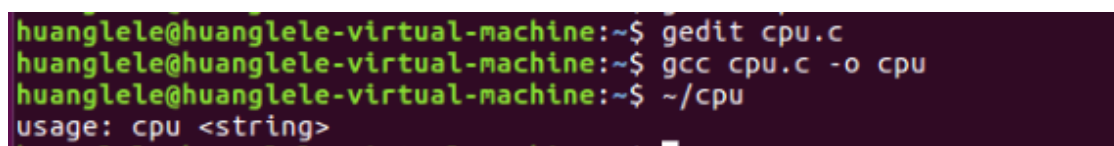
- 在虚拟机中使用 gedit 命令编写代码 cpu.c，该代码的含义是，循环输出传入参数到屏幕，中间通过 sleep（2）让程序执行过程中阻塞两秒。如果无参数则输出 usage:cpu<string>，代码如下图：



```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    int i = 10;
    while (i-- >= 0) {
        sleep(2);
        printf("%s\n", str);
    }
    return 0;
}
```

- 运行示例



```
huanglele@huanglele-virtual-machine:~$ gedit cpu.c
huanglele@huanglele-virtual-machine:~$ gcc cpu.c -o cpu
huanglele@huanglele-virtual-machine:~$ ./cpu
usage: cpu <string>
```

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：

```
huanglele@huanglele-virtual-machine: ~  
huanglele@huanglele-virtual-machine:~$ ./cpu A & ./cpu B & ./cpu C & ./cpu D  
[1] 5843  
[2] 5844  
[3] 5845  
C  
B  
A  
D  
C  
B  
A  
D  
C  
B  
A  
D  
C  
B  
A  
D  
C  
B  
A  
D
```

- 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：


```
mem.c (~/) - gedit
打开(O)  [icon]

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int)); // a1
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n", getpid(), p); // a2
    *p = 0; // a3
    int i=10;
    while (i-->=0) {
        sleep(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p); // a4
    }
    return 0;
}
```

运行示例：

```
huanglele@huanglele-virtual-machine:~$ gedit mem.c
huanglele@huanglele-virtual-machine:~$ gcc mem.c -o mem
huanglele@huanglele-virtual-machine:~$ ./mem
(5971) address pointed to by p: 0x17ae010
(5971) p: 1
(5971) p: 2
(5971) p: 3
(5971) p: 4
(5971) p: 5
(5971) p: 6
(5971) p: 7
(5971) p: 8
(5971) p: 9
(5971) p: 10
(5971) p: 11
huanglele@huanglele-virtual-machine:~$
```

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：

```
huanglele@huanglele-virtual-machine:~$ ./mem & ./mem &
[1] 5973
[2] 5974
huanglele@huanglele-virtual-machine:~$ (5974) address pointed to by p: 0x1a72010
(5973) address pointed to by p: 0x22e4010
(5974) p: 1
(5973) p: 1
(5974) p: 2
(5973) p: 2
(5974) p: 3
(5973) p: 3
(5974) p: 4
(5973) p: 4
(5974) p: 5
(5973) p: 5
(5973) p: 6
(5974) p: 6
```

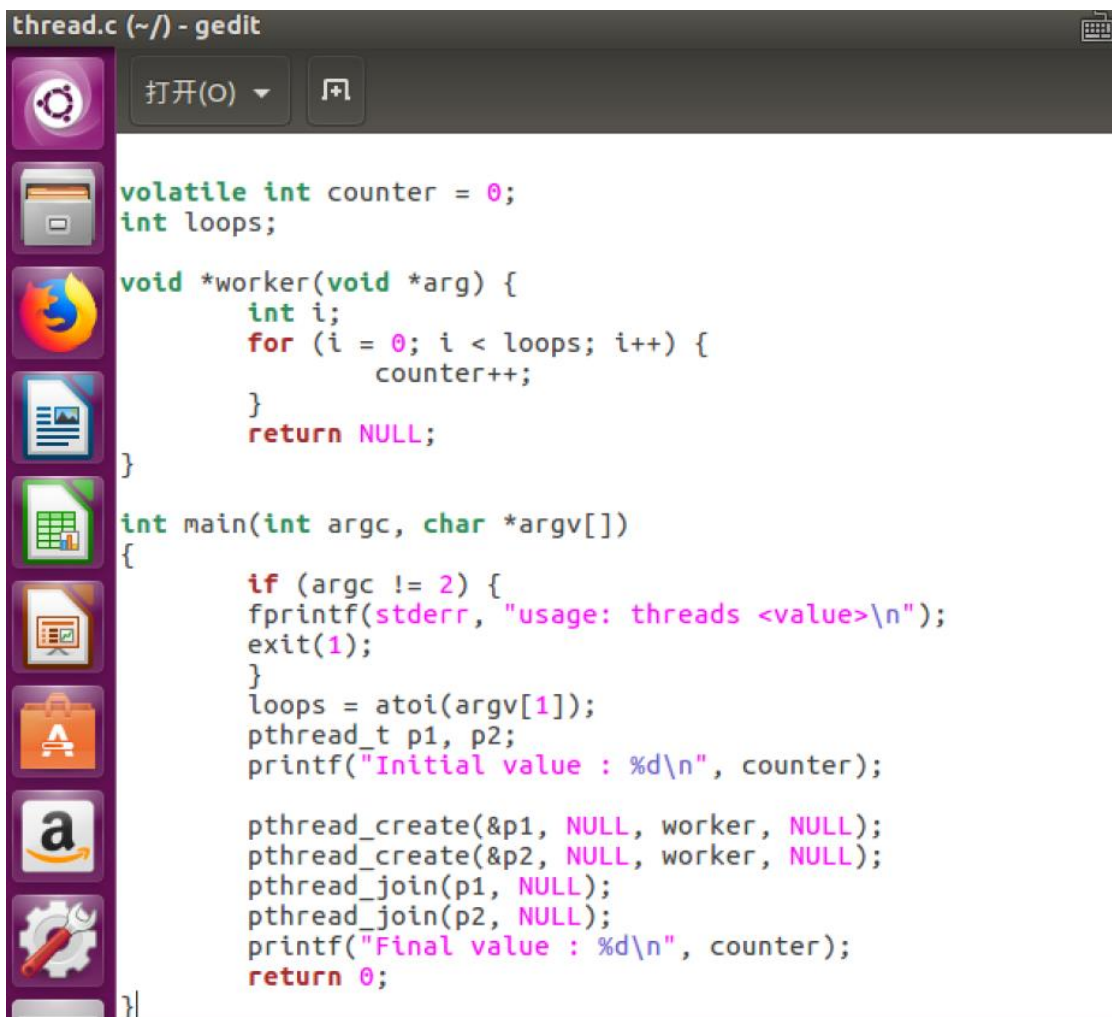
- 可以看到，在当前情况下，两个分别运行的程序的内存地址并不相同；且两个进程之间没有相互影响。
- 但是，每个进程的 4G 内存空间为虚拟内存空间，需要映射到实际物理内存地址，且所有进程共享同一物理内存区域，进程只需要用页表来记录这些映射即可。我们的函数中输出的是虚拟地址，进程之间享有独立的虚拟内存，故内存地址也可能会相同，但是其映射到的物理地址是不一样的，所以对数值的操作是没有影响的。

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；



```
thread.c (~/) - gedit

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value : %d\n", counter);
    return 0;
}
```

- 运行结果：

```
huanglele@huanglele-virtual-machine:~$ gcc thread.c -o thread -Wall -pthread
huanglele@huanglele-virtual-machine:~$ ./thread 1000
Initial value : 0
Final value : 2000
```

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：

```
huanglele@huanglele-virtual-machine:~$ ./thread 10000
Initial value : 0
Final value : 20000
huanglele@huanglele-virtual-machine:~$ ./thread 100000
Initial value : 0
Final value : 200000
huanglele@huanglele-virtual-machine:~$ ./thread 1000000
Initial value : 0
Final value : 2000000
huanglele@huanglele-virtual-machine:~$ ./thread 1000000000
Initial value : 0
Final value : 1987851250
```

- 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。