

实验一: 操作系统初步

(注意: 本次所有实验都在 Linux 中完成)

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。

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

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main()
5 {
6     pid_t pid;
7
8     pid = getpid();
9     printf("%d\n",pid);
10
11     return 0;
12 }
```

1、

`getpid` 的系统调用号:

32 位: 0x20

64 位: 0x39

Linux 使用 0x80 号中断作为系统调用的入口

2、

c 程序:

```
#include <stdio.h>
```

```
int main(){
    printf("Hello World\n");
}
```

汇编程序:

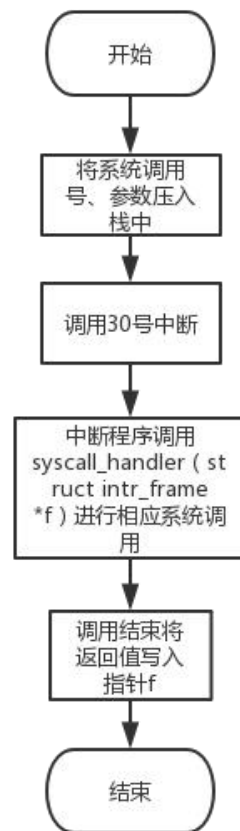
```
.file "test.c"
```

```

        .section  .rodata
.LC0:
        .string   "Hello World"
        .text
        .globl main
        .type main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size main, .-main
        .ident "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.11) 5.4.0 20160609"
        .section .note.GNU-stack,"",@progbits

```

3、



二、（并发实验）根据以下代码完成下面的实验。

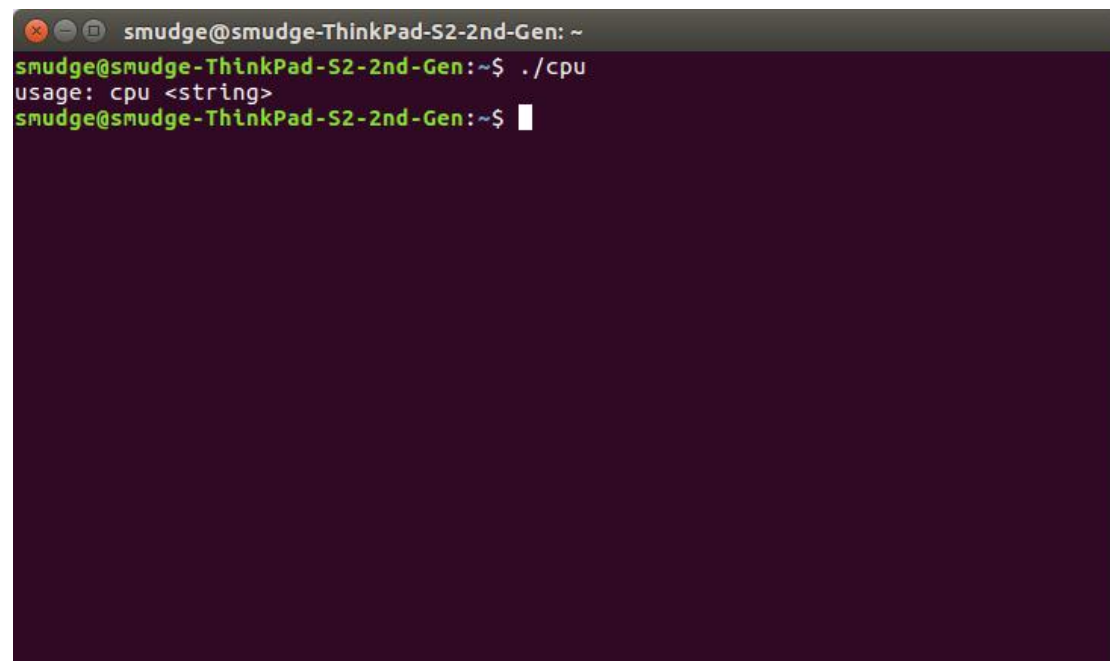
要求：

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 }
19 return 0;
1、
```



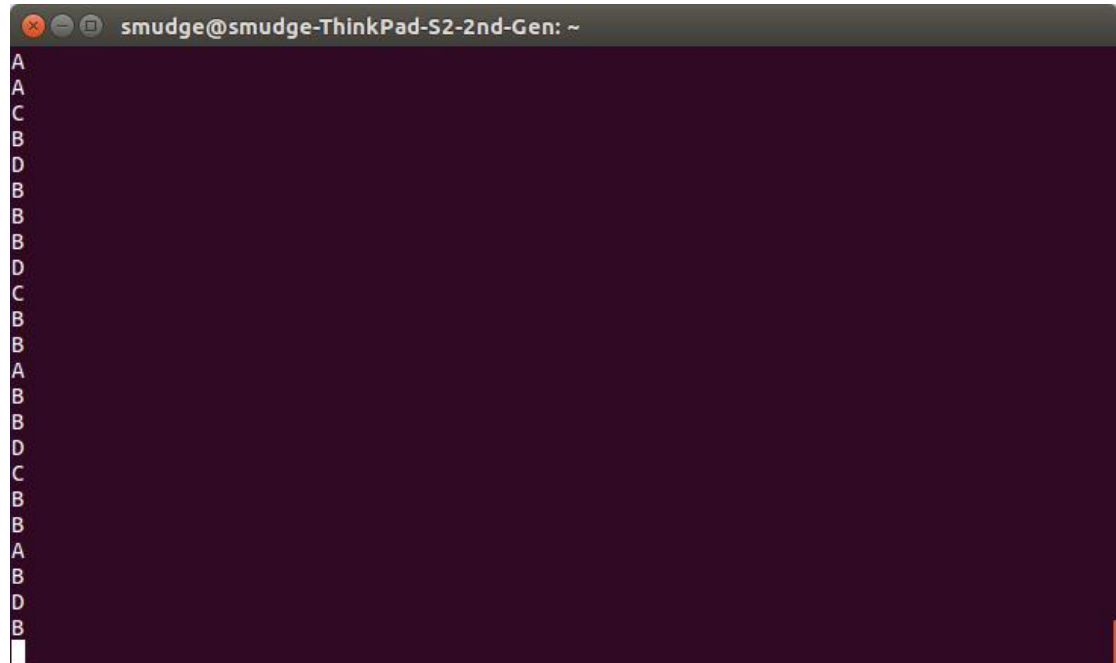
```
smudge@smudge-ThinkPad-S2-2nd-Gen: ~
smudge@smudge-ThinkPad-S2-2nd-Gen:~$ ./cpu
usage: cpu <string>
smudge@smudge-ThinkPad-S2-2nd-Gen:~$
```

argc 是从命令行获得的命令参数数目，argv[] 是从命令行获得的数组，数组元素为具体的命令字符串

我们执行 ./cpu 时，只有一个命令参数，因此输出 usage: cpu <string>

当命令参数为 2 时，输出 argv[1]，argv[0] 为程序名称，argv[1] 为输入的那个参数。

2、



```
smudge@smudge-ThinkPad-S2-2nd-Gen: ~  
A  
A  
C  
B  
D  
B  
B  
B  
B  
D  
C  
B  
B  
A  
B  
B  
B  
D  
C  
B  
B  
B  
A  
B  
D  
B
```

程序 cpu 运行了 4 次，他们的运行不是顺序的，具有间断性的特点，因为这四个进程对资源都有需求，互相制约，某一进程运行时其他的需要等待资源，因此我们看到的输出并不是 ABCD 这样的顺序，而是一种乱序。

三、（内存分配实验）根据以下代码完成实验。

要求：

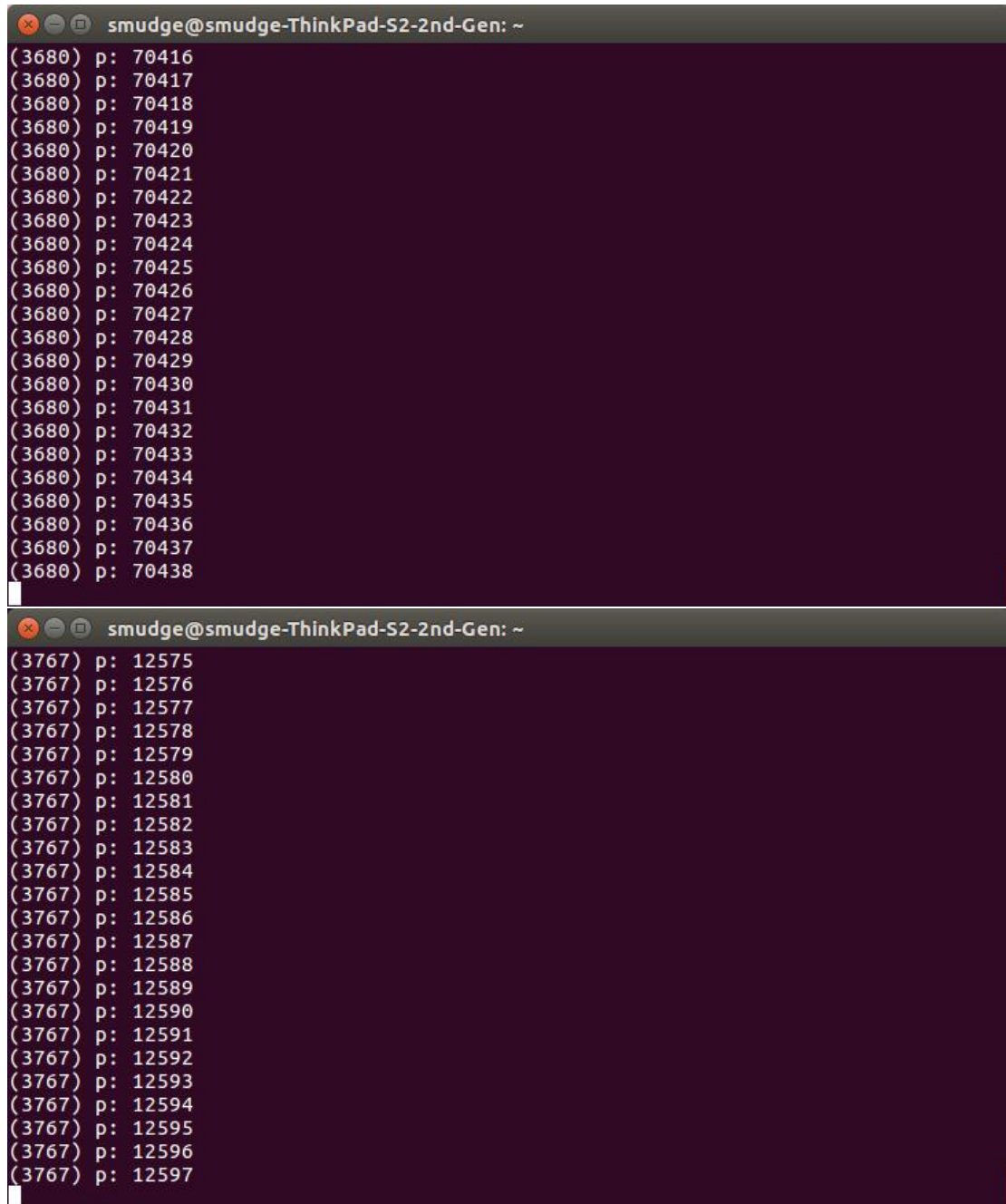
1、阅读并编译运行该程序(mem.c)，观察输出结果，说明程序功能。(命令： gcc -o mem mem.c -Wall)

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;
```

1、



The image shows two terminal windows from a Linux system. The top window shows the output of a program running with PID 3680. It prints 23 lines of memory addresses, starting from 70416 and increasing by 1 until 70438. The bottom window shows the output of a program running with PID 3767. It prints 23 lines of memory addresses, starting from 12575 and increasing by 1 until 12597. Both windows have a dark purple background and a title bar indicating the user is 'smudge' on a 'smudge-ThinkPad-S2-2nd-Gen' machine.

```
smudge@smudge-ThinkPad-S2-2nd-Gen: ~
(3680) p: 70416
(3680) p: 70417
(3680) p: 70418
(3680) p: 70419
(3680) p: 70420
(3680) p: 70421
(3680) p: 70422
(3680) p: 70423
(3680) p: 70424
(3680) p: 70425
(3680) p: 70426
(3680) p: 70427
(3680) p: 70428
(3680) p: 70429
(3680) p: 70430
(3680) p: 70431
(3680) p: 70432
(3680) p: 70433
(3680) p: 70434
(3680) p: 70435
(3680) p: 70436
(3680) p: 70437
(3680) p: 70438

smudge@smudge-ThinkPad-S2-2nd-Gen: ~
(3767) p: 12575
(3767) p: 12576
(3767) p: 12577
(3767) p: 12578
(3767) p: 12579
(3767) p: 12580
(3767) p: 12581
(3767) p: 12582
(3767) p: 12583
(3767) p: 12584
(3767) p: 12585
(3767) p: 12586
(3767) p: 12587
(3767) p: 12588
(3767) p: 12589
(3767) p: 12590
(3767) p: 12591
(3767) p: 12592
(3767) p: 12593
(3767) p: 12594
(3767) p: 12595
(3767) p: 12596
(3767) p: 12597
```

Getpid 获取当前进程号后输出，p 为一个 int 类型的指针，程序的功能为：在当前进程中，利用 p 不断申请 int 类型的地址空间，申请成功后，p 加 1 并输出

2、

```
smudge@smudge-ThinkPad-S2-2nd-Gen: ~  
smudge@smudge-ThinkPad-S2-2nd-Gen:~$ sudo sysctl -w kernel.randomize_va_space=0  
[sudo] password for smudge:  
kernel.randomize_va_space = 0  
smudge@smudge-ThinkPad-S2-2nd-Gen:~$ gcc -o mem mem.c -Wall  
smudge@smudge-ThinkPad-S2-2nd-Gen:~$ gcc -o mem mem.c -Wall  
smudge@smudge-ThinkPad-S2-2nd-Gen:~$ ./mem & ./mem &  
[1] 3701  
[2] 3702  
smudge@smudge-ThinkPad-S2-2nd-Gen:~$ (3701) address pointed to by p: 0x602010  
(3702) address pointed to by p: 0x602010  
(3701) p: 1  
(3702) p: 1  
(3702) p: 2  
(3701) p: 2  
(3701) p: 3  
(3702) p: 3  
(3702) p: 4  
(3701) p: 4  
█
```

两个分别运行的程序分配的内存地址不相同，我认为他们共享同一块物理内存。linux 使用虚拟内存，运行同一程序两次，产生两个进程，但两个进程的内核虚拟内存都映射到同一块物理内存，用户内存的映射则不同。

程序直接使用物理内存，会出现没有连续可用空间的情况，也是不能最大利用内存的缺点。直接使用物理内存，可能会出现不同程序使用相同内存地址的情况，此时会导致两个程序的崩溃。MMU 把虚拟内存里的数据映射到物理内存上，到物理内存处理。虚拟地址对应到物理地址。

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。）

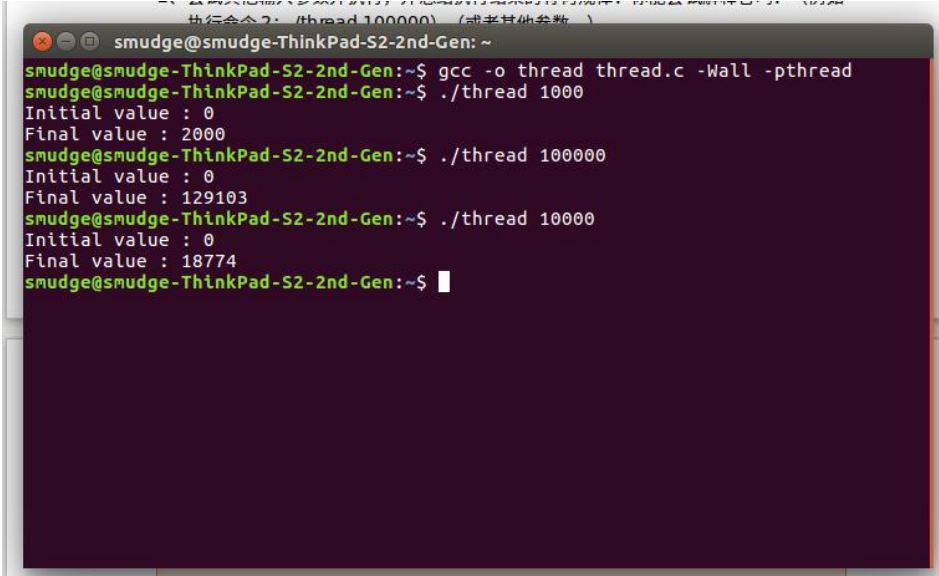
提示：哪些变量是各个线程共享的，线程并发执行时访问共享变量会不会导致意想不到的问题。

```
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;

```



```

smudge@smudge-ThinkPad-S2-2nd-Gen: ~
smudge@smudge-ThinkPad-S2-2nd-Gen:~$ gcc -o thread thread.c -Wall -pthread
smudge@smudge-ThinkPad-S2-2nd-Gen:~$ ./thread 1000
Initial value : 0
Final value : 2000
smudge@smudge-ThinkPad-S2-2nd-Gen:~$ ./thread 100000
Initial value : 0
Final value : 129103
smudge@smudge-ThinkPad-S2-2nd-Gen:~$ ./thread 10000
Initial value : 0
Final value : 18774
smudge@smudge-ThinkPad-S2-2nd-Gen:~$

```

程序的功能：argv[1]为输入的参数作为循环的次数，创建两个线程，每个线程都对被线程共享的变量 counter 做出循环次数的加一，最后输出 counter 的值，理论上 counter 的值应为参数的二倍。

在参数为 1000 时 Final value 为 2000，参数为 10000 时 Final value 为 18774，参数为 100000 时 Final value 为 129103，因为多线程访问共享的全局变量会引起混乱。

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则，排成一个队列，每次调度时，把 CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几 ms 到几百 ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据

此信号来停止该进程的执行，并将它送往就绪队列的末尾;然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程，在一给定的时间内，均能获得一时间片的处理机执行时间。

如果在时间片结束时进程还在运行，则 CPU 将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结束，则 CPU 当即进行切换。调度程序所要做的就是维护一张就绪进程列表，当进程用完它的时间片后，它被移到队列的末尾。