北京交通大学

《操作系统》

实验一

操作系统初步

学号： **16281052**

姓名： **杨涵晨**

班级： **计科 1601**

专业： **计算机科学与技术**

一. 系统调用实验

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。

1.1 题解答：

API 接口函数的直接调用代码如下：可以看到直接利用 getpid()函数进行，然后返回到预先定义好的变量 pid 中。然后将 pid 打印出来。

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

int main()
{
    pid_t pid;

    pid = getpid();
    printf("%d\n",pid);

    return 0;
}
```

利用 gcc 进行编译，生成可执行文件然后运行得到：3064

通过查看文档发现，getpid()函数功能为进行进程的查看，返回值为进程识别码。

```
yhc@yhc-virtual-machine /mnt/hgfs/share/1 gcc getpid.c -o getpid -Wall
yhc@yhc-virtual-machine /mnt/hgfs/share/1 ./getpid
3064
```

通过进行 linux 系统调用号查询，查看 vim 中的文件，可以看到这我的系统中 getpid 为 172.

```
yhc@yhc-virtual-machine /usr/include/asm-generic vim unistd.h
yhc@yhc-virtual-machine /usr/include/asm-generic cd .
yhc@yhc-virtual-machine /usr/include/asm-generic cd ..
yhc@yhc-virtual-machine /usr/include cd asm-generic
yhc@yhc-virtual-machine /usr/include/asm-generic vim unistd.h
```

```
/* kernel/timer.c */
#define __NR_getpid 172
__SYSCALL(__NR_getpid, sys_getpid)
#define __NR_getppid 173
__SYSCALL(__NR_getppid, sys_getppid)
#define __NR_getuid 174
__SYSCALL(__NR_getuid, sys_getuid)
```

在汇编代码中调用中，利用汇编的软中断进行调用，int 0x80;将汇编代码嵌套在 c 中。代码如下：

```
#include <stdio.h>
#include <unistd.h>
int main()
{
        pid_t tt;
        asm volatile (
            "movl $0x14,%%eax\n\t"
            "int $0x80\n\t"
            "movl %%eax,%0\n\t"
            :"=m"(tt)
        );
        printf(" current PID is : %u\n",tt);
        return 0;
}
```

执行结果如下：其中汇编代码的中断向量号为 0x14.

```
yhc@yhc-virtual-machine  /mnt/hgfs/share/1  gcc getpid2.c -Wall -o getpid2
yhc@yhc-virtual-machine  /mnt/hgfs/share/1  ./getpid2
current PID is : 3173
```

1.2 题解答：

C 语言的程序代码如下进行 gcc 编译与执行，直接调用 printf（）函数，进行实现

```
#include <stdio.h>

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

执行结果如下：

```
yhc@yhc-virtual-machine  /mnt/hgfs/share/1  gcc hello.c -o hello -Wall
yhc@yhc-virtual-machine  /mnt/hgfs/share/1  ./hello
hello world!!!%
```

汇编的代码如下，通过 nasm 进行编译和实现。

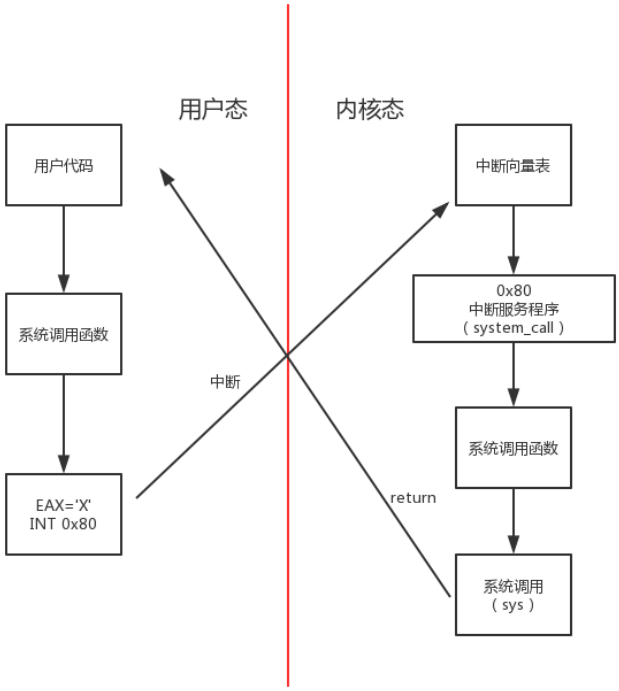
```
section .data
msg     db      "Hello, world!",0xA
len     equ     $ - msg
section .text
   global _start
_start:
        mov     eax,4
        mov     ebx,1
        mov     ecx,msg
        mov     edx,len
        int     0x80
        mov     eax,1
        xor     ebx,ebx
        int     0x80
```

执行结果如下：

```
yhc@yhc-virtual-machine /mnt/hgfs/share/1 nasm -f elf64 hello.asm
yhc@yhc-virtual-machine /mnt/hgfs/share/1 ld -s -o hello hello.o
yhc@yhc-virtual-machine /mnt/hgfs/share/1 ./hello
Hello, world!
yhc@yhc-virtual-machine /mnt/hgfs/share/1
```

1.3 题解答

流程图如下：区分用户态和内核态。



二. 并发实验

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。

2.1 题解答

主要功能为：

1.当输入两个参数时，进行 stderr 标准错误，打印“usage:cpu<string>”

```
 yhc@yhc-virtual-machine  /mnt/hgfs/share/1  ./cpu 2 A
usage: cpu <string>
 ✘ yhc@yhc-virtual-machine  /mnt/hgfs/share/1
```

2.但是当只有一个参数时进行打印，通过循环语句进行限制，循环 10 次后停止。中间通过 sleep（2）让程序执行过程中阻塞两秒。

```
 yhc@yhc-virtual-machine  /mnt/hgfs/share/1  gcc cpu.c -o cpu -Wall
 yhc@yhc-virtual-machine  /mnt/hgfs/share/1  ./cpu A &;./cpu B &./cpu C &;./cp
u D &;
[1] 3679
[2] 3680
[3] 3681
[4] 3682
```

四个程序同步进行时我们可以看见。执行时好像没有规律可循。

```
 ⚙ yhc@yhc-virtual-machine  /mnt/hgfs/share/1  A
C
D
B
A
D
C
B
B
D
A
C
D
A
B
C
A
D
B
C
D
```

主要原因如下：现代操作系统中进程的运行都是并发实现的，并不是像以前的单道批处理的操作系统那样，总是按照进程进入内存的先后顺序来执行，因此进程的运行的顺序并没有规律。现代 CPU 一般都是多核 CUP，因此实验中的四个进程可能也不是简单的在一个 CPU 中并发，而有可能是在多个 CPU 核心中并行运行，也有可能某两个进程在一个 CPU 核心中并发运行，和其他的进程在不同的 CPU 核心中并行运行。所以进程的运行顺序并没有特别的规律。

代码如下：

```
#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;
}
```

三、内存分配实验

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 &

3.1 题解答

主要功能为:通过一个 while 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。

运行程序如下：我们可以看到开辟的两个进程用了两个不同的物理地址，分别为 0x7f9010，0x20c7010.所以物理地址并不相同

```
yhc@yhc-virtual-machine /mnt/hgfs/share/1 gcc mem.c -o mem -Wall
yhc@yhc-virtual-machine /mnt/hgfs/share/1 ./mem & ; ./mem &
[1] 3550
[2] 3551
(3550) address pointed to by p: 0x7f9010
(3551) address pointed to by p: 0x20c7010
```

不能共用一块地址，如果进行共用，这个答案将会是 20，因为两个不同的进程都在给一个物理地址上做加法，这个导致了程序的混乱。如果想要进行共享，那必然要进行保存现场和回复现场。正确结果如下：

```
yhc@yhc-virtual-machine /mnt/hgfs/share/1 (3550) p: 1
(3551) p: 1
(3551) p: 2
(3550) p: 2
(3550) p: 3
(3551) p: 3
(3550) p: 4
(3551) p: 4
(3551) p: 5
(3550) p: 5
(3551) p: 6
(3550) p: 6
```

实验代码如下：

```c
#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);
    *p = 0; // a3
    int i=10;
    while (i-->=0) {
    sleep(1);
    *p = *p + 1;
    printf("(%d) p: %d\n", getpid(), *p); // a4
    }
    return 0;
}
```

四、共享的问题

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。

4.1 题解答

主要功能：通过命令行输入参数到 loop，然后利用 pthread_create()进行线程开辟，其中第三个参数是线程运行函数的地址。

利用 worker 函数进行累加 counter,因为 counter 是一个全局变量，利用两个 pthread_create()进行两个累计，然后利用 pthread_join()函数进行进程回收。

最后利用 printf()打印全局变量 counter 的值。

实验代码如下：

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <ctype.h>

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;
    }
```

对代码进行编译运行。我们可以发现，大部分结果都是输入参数的两倍，但是在线程数足够大的时候就会小于两倍，两个线程共享了一个执行函数 worker，进行 for 循环对 counter 进行累加。

```
 yhc@yhc-virtual-machine  /mnt/hgfs/share/1  gcc thread.c -o thread -Wall -pth
read
 yhc@yhc-virtual-machine  /mnt/hgfs/share/1  ./thread 100
Initial value : 0
Final value : 200
 yhc@yhc-virtual-machine  /mnt/hgfs/share/1  ./thread 1000
Initial value : 0
Final value : 2000
 yhc@yhc-virtual-machine  /mnt/hgfs/share/1  ./thread 10000
Initial value : 0
Final value : 20000
```

```
x@A\+ ⚙ yhc@yhc-virtual-machine  /mnt/hgfs/share/1  ./thread 1000
Initial value : 0
Final value : 2000
 yhc@yhc-virtual-machine  /mnt/hgfs/share/1  ./thread 1000000000
Initial value : 0
Final value : 1992780770
```

其中 counter，loops 这两个全局变量，和 worker()函数是共享的。