

实验一 操作系统基础

计科1601 16281052 杨涵晨

一.系统调用实验

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。

```
#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.但是上网查询 (64位linux的编号为39, 32位20)

```
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;
}
```

```
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：

pintos中关于系统调用的主要源码：

1. /src/lib/user/syscall.c

宏定义了四种系统调用的方式，分别是不传递参数、传递一个参数、传递两个参数、传递三个参数，如下所示：

```

define syscall0(NUMBER) ...
define syscall1(NUMBER, ARG0) ...
define syscall2(NUMBER, ARG0, ARG1) ...
define syscall3(NUMBER, ARG0, ARG1, ARG2) ...

```

定义了20种系统调用函数

```

void halt (void);
void exit (int status);
pid_t exec (const char *file);
int wait (pid_t pid);
bool create (const char *file, unsigned initial_size);
bool remove (const char *file);
int open (const char *file);
int filesize (int fd);
int read (int fd, void *buffer, unsigned size);
int write (int fd, const void *buffer, unsigned size);
void seek (int fd, unsigned position);
unsigned tell (int fd);
void close (int fd);
mapid_t mmap (int fd, void *addr);
void munmap (mapid_t mapid);
bool chdir (const char *dir);
bool mkdir (const char *dir);
bool readdir (int fd, char name[READDIR_MAX_LEN + 1]);
bool isdir (int fd);

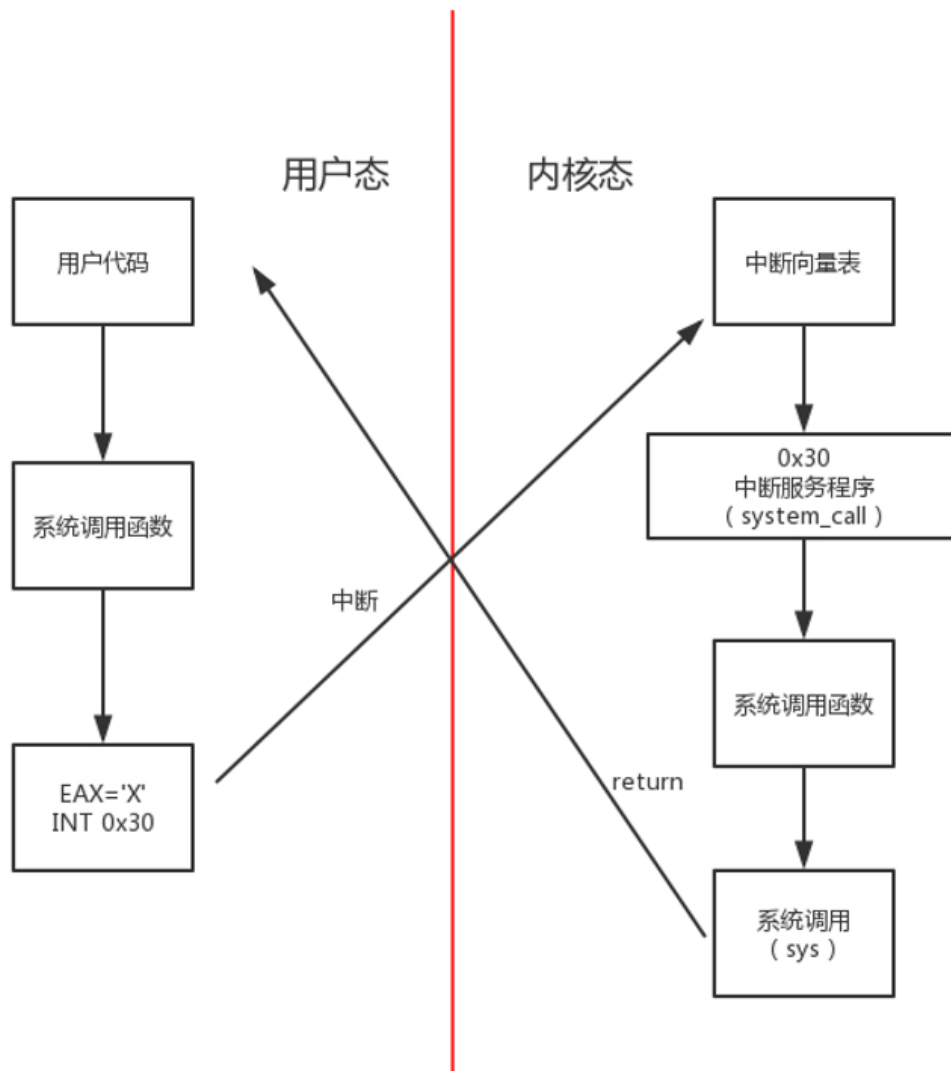
```

```
int inumber (int fd);
```

2. src/userprog/syscall.c

这个文件中只有两个函数syscall_init 和syscall_handler,其中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

流程图如下：区分用户态和内核态。



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)
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:

- 当输入两个参数时, 进行stderr标准错误, 打印“usage: cpu”

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

- 但是当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 & ./cpu 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
```

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CPU，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

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
```

操作系统在进行对每个进程分配空间时不会直接在memory上直接对应存储，而是通过‘虚拟内存技术’，为每个进程分配4G内存空间，0-3G 属于用户空间，3G-4G 属于内核空间。每个进程的用户空间不同，但内核空间相同。

所以从进程的不同性上面来说，有可能对应相同的虚拟地址。但是真实物理地址不会相同，这种虚拟内存也保证不同进程相互隔离，错误的程序不会干扰别的正确的进程。

```

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

```

通过命令进行，尝试关闭ALSR 地址空间随机化

```
sysctl -w kernel.randomize_va_space=0
```

```

yhc@yhc-virtual-machine /mnt/hgfs/share master cd
yhc@yhc-virtual-machine ~ sudo su
[sudo] yhc 的密码:
yhc-virtual-machine# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
yhc-virtual-machine# sudo yhc
sudo: yhc: 找不到命令
yhc-virtual-machine# exit

```

实验结果中，分配的地址进行了变化，都指向0x602010，当我们取消随机分配的时候，操作系统应该会进行连续分配。直接从用户块的首地址往下进行分配，所以两个不同进程对应的地址相同，但是其实都是不同物理地址。

```

yhc@yhc-virtual-machine /mnt/hgfs/share/lab1 master ./mem & ; ./mem &
[1] 3387
[2] 3388
(3387) address pointed to by p: 0x602010
(3388) address pointed to by p: 0x602010
yhc@yhc-virtual-machine /mnt/hgfs/share/lab1 master (3387) p: 1
(3388) p: 1
(3387) p: 2
(3388) p: 2
(3387) p: 3
(3388) p: 3
(3387) p: 4
(3388) p: 4
(3387) p: 5
(3388) p: 5
(3387) p: 6
(3388) p: 6
(3387) p: 7

```

实验代码如下：

```

#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

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 -pthread
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
x0A\+ 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。

由于两个线程在同一个进程中，并且访问操作的是共享的变量。如果每个线程对内存都是可读可写的话，就会发生读取脏数据的问题。现代CPU一般采用了加锁的解决办法，通过加锁使另一个线程不能读取。但是由于现代计算机都是多核心的，对于每个独立的CPU核心来说，都不会发生问题。线程数过大时，就会用不同的cpu核心进行互相读脏数据，依然存在问题。

当输入的参数比较小的时候，一个CPU的核心足够处理，就是单核CPU运行多线程，由于每个核心都有内存锁机制，所以计算结果没有错误当输入的参数比较大的时候，使用多个CPU核心进行运算，就会发生读取脏数据的问题。