**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。

**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、命令：printf(“Hello World!\n”)可归入一个{C标准函数、GNU C 函数库、Linux API}中哪一个或者哪几个？请分别用相应的linux系统调用的C函数形式和汇编代码两种形式来实现上述命令。

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/>

**实验原理及结果：**

1、（1）API接口函数getpid()直接调用，输出为3720。

函数为：（见hello.c）

#include <stdio.h>

#include <unistd.h>

int main()

{

pid\_t pid;

pid = getpid();

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

return 0;

}

getpid()函数对应表头文件为#include <unistd.h>，返回值的类型为pid\_t，返回了当前进程的进程识别码。

（2）汇编终端调用getpid()，输出为3788。

函数为：（见hello2.c）

#include <stdio.h>

#include <unistd.h>

int main()

{

pid\_t pid;

asm volatile(

"mov $0,%%ebx\n\t"

"mov $0x14,%%eax\n\t"

"int $0x80\n\t"

"mov %%eax,%0\n\t"

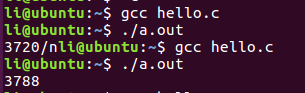
:"=m"(pid)

);

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

return 0;

}



（3）系统调用通过中断的方式在内核态和用户态之间切换。由汇编代码可以看出，eax保存系统调用号，ebx等寄存器保存具体参数，当触发0x80中断时，经过中断处理程序，进入了内核态。将eax寄存器的值存入pid。int软件中断指令会将返回地址入栈，使程序跳转到对应中断入口处。

由代码得知getpid系统调用号为20，linux系统调用的的中断向量号为0x80。

2、该命令归入C标准函数与Linux API。

Linux中C函数形式：(hello3.c)

#include <stdio.h>

int main()

{

printf(“Hello World!\n");

return 0;

}



汇编代码形式：

.section .data

output:

.ascii "Hello world!\n"

len = . - output

.section .text

.globl \_start

\_start:

movl $4, %eax

movl $1, %ebx

movl $output, %ecx

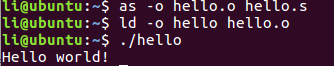
movl $len, %edx

int $0x80

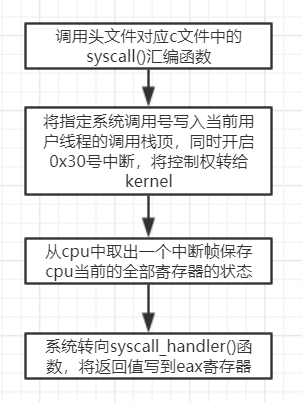
movl $1, %eax

movl $0, %ebx

int $0x80



3、



**Problems and solutions:**

1、Problem：不同网站上写的getpid函数的系统调用号不同，有的写20，有的写39。Getpid的有效的系统调用号不确定。

Solution：32位linux中getpid函数的系统调用号为20，64位Linux中getpid函数的系统调用号为39。

2、Problem:代码编译虽然成功输出”Hello World!”。但是报segmentation fault(core dumped)的错误。

Solution：Segmentation fault (core dumped)多为内存不当操作造成。空指针、野指针的读写操作，数组越界访问，破坏常量等。代码中加上movl $1, %eax和movl $0, %ebx语句，关闭程序。

**Remaining issues:**

1、64位系统可以用64位或32位的系统调用号，但是当系统调用号位39时，输出的getpid()值为负。在哪里选择64系统选择使用64位还是32位系统调用号呢？

二、（并发实验）根据以下代码完成下面的实验。

**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。

#include <stdio.h>

#include <stdlib.h>

#include <sys/time.h>

#include <assert.h>

int main(int argc, char \*argv[])

{

if (argc != 2) {

fprintf(stderr, "usage: cpu <string>\n");

exit(1);

}

char \*str = argv[1];

while (1) {

for(int i = 0;i<10000000;i++){}

printf("%s\n", str);

}

return 0;

}

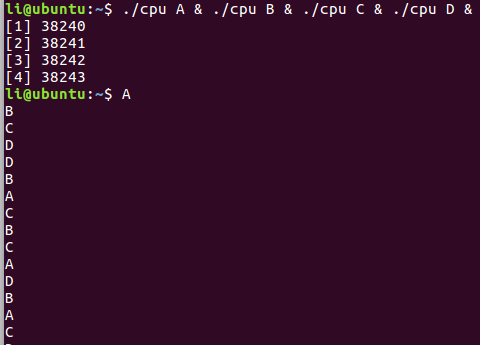
**实验原理及结果：**

1、输出结果如下：



如果命令行参数的个数不为2，则向屏幕输出字符串“usage:cpu <string>“。

2、运行结果如下：



操作系统具有并发性。程序cpu运行了4次，每个程序传入两个命令行参数，四个程序并发运行。一个程序未执行完而另一个程序便已开始执行，因此程序与计算不一一对应。运行次序不能确定，所以输出A，B，C，D的顺序不同。

**Problems and solutions:**

1、Problem：四个程序并发运行后输出全为A。

Solution：死循环中的停顿循环次数过少，导致不断输出A。多设置几次循环次数，保证每个程序的停顿时间足够长。

三、（内存分配实验）根据以下代码完成实验。

**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 &

#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,进程识别码,指针地址

\*p = 0; // a3

while (1) {

for(int i = 0;i<1000000;i++){for(int j = 0;j<3;j++){}}

\*p = \*p + 1;

printf("(%d) p: %d\n", getpid(), \*p); // a4

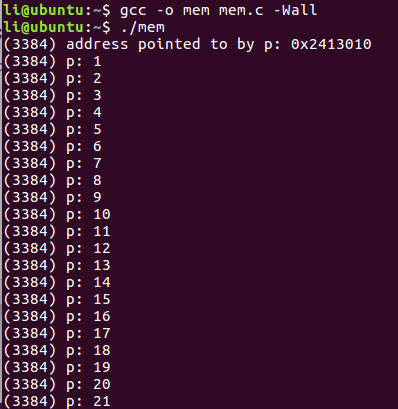
}

return 0;

}

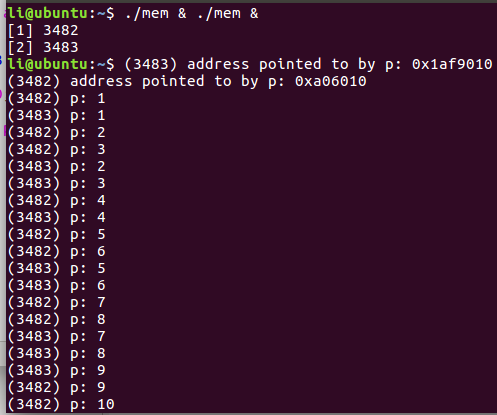
**实验原理及结果：**

1、运行结果如下：



查看所运行程序的进程识别码，及在该进程中分配的指针的地址，可看出程序的运行状态。

2、运行结果如下：



进程号为3483的程序分配的内存为0x1af9010，进程号为3482的程序分配的内存为0xa06010。可看出，并发运行的两个程序分配的内存地址不相同。但是共享同一物理内存。每个进程都有自己的4G地址空间，从 0x00000000-0xFFFFFFFF 。通过每个进程自己的一套页目录和页表来实现。由于每个进程有自己的页目录和页表，所以每个进程的地址空间映射的物理内存是不一样的。所有进程共享同一物理内存，每个进程只把自己目前需要的虚拟内存空间映射并存储到物理内存上。

【注】对并发程序、物理内存及虚拟内存的描述详见博客：

https://blog.csdn.net/zhyfxy/article/details/70157248

四、（共享的问题）根据以下代码完成实验。

**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。）
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。

#include <stdio.h>

#include <stdlib.h>

#include<pthread.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);

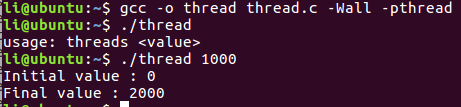
printf("Final value : %d\n", counter);

return 0;

}

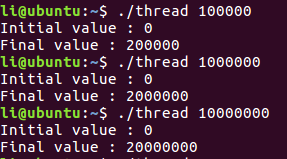
**实验原理及结果：**

1、运行结果如下所示：



传入的参数为执行循环的次数，输出的结果为传入参数的二倍。该程序创建了两个线程，每个线程循环传入参数的值的次数，输出的值为总共循环的次数。

2、运行结果如下所示：



传入参数数字大的时候仍保持二倍。