

操作系统实验一

16281053 杨瑗彤 计科 1601

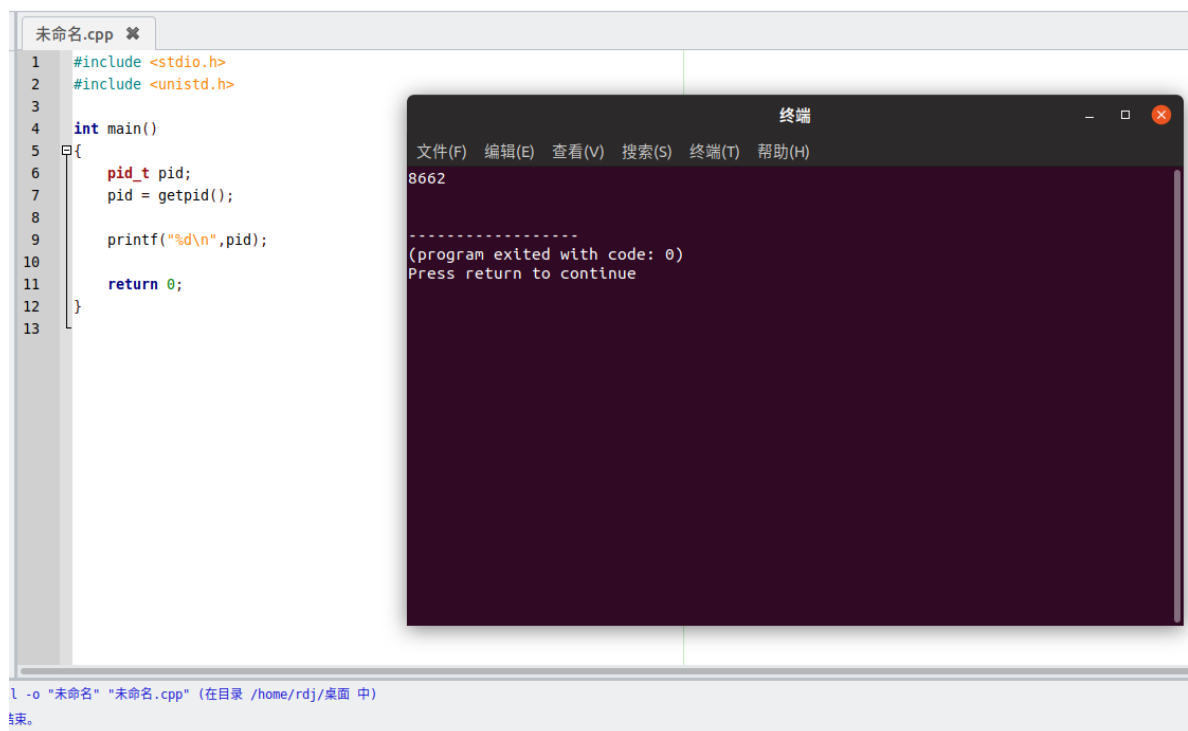
一、（系统调用实验）了解系统调用不同的封装形式。

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。

(1)

直接调用 `getpid()` 函数：

使用直接调用 `getpid()` 函数，取得进程识别码，结果为 8662。代码见如下截图。



The screenshot shows a C++ program in a text editor and its execution in a terminal. The program, named '未命名.cpp', includes `<stdio.h>` and `<unistd.h>`. It defines a `main()` function where a `pid_t` variable `pid` is declared, `getpid()` is called, and the result is printed using `printf("%d\n", pid);`. The program then returns 0. The terminal window, titled '终端', shows the output '8662' followed by a message indicating the program exited with code 0 and a prompt to press return to continue.

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main()
5 {
6     pid_t pid;
7     pid = getpid();
8
9     printf("%d\n", pid);
10
11     return 0;
12 }
13
```

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
8662
-----
(program exited with code: 0)
Press return to continue
```

1 -o "未命名" "未命名.cpp" (在目录 /home/rdj/桌面 中)
结束。

汇编中断方式调用 `getpid()` 函数：

使用汇编中断方式调用 `getpid()` 函数，首先存放一个标志到 `ebx` 中，然后 `eax` 中存放系统调用函数的调用号，根据传递进来的系统中断号，然后找到 `getpid()` 函数调用。结果为 8570，代码见如下截图。

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main()
5 {
6     pid_t pid;
7
8     asm volatile(
9         "mov $0,%%ebx\n\t"
10        "mov $0x14,%%eax\n\t"
11        "int $0x80\n\t"
12        : "=m"(pid)
13        );
14
15    printf("%d\n",pid);
16
17    return 0;
18 }
19
```

终端

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

8570

(program exited with code: 0)
Press return to continue

Getpid 的系统调用号为 39，linux 系统调用的中断向量号是 80H。

(2) 习题 1.13

使用 C 函数形式输出 hello world:

直接使用 printf 输出即可，代码见如下截图。

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello world!\n");
6     return 0;
7 }
8
9
```

终端

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

Hello world!

(program exited with code: 0)
Press return to continue

使用汇编形式编写代码输出 hello world:

代码如下图:

```
cpu.cpp x hwhb.c x hwc.c x
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main(int argc, char **argv)
5 {
6     char* msg = "Hello World";
7     int len = 11;
8     int result = 0;
9
10    asm volatile ("movl %2, %%edx;\n\r"
11                  "movl %1, %%ecx;\n\r"
12                  "movl $1, %%ebx;\n\r"
13                  "movl $4, %%eax;\n\r"
14                  "int $0x80"
15                  : "=m"(result)
16                  : "m"(msg), "r"(len)
17                  : "%eax");
18
19    return 0;
20 }
21
```

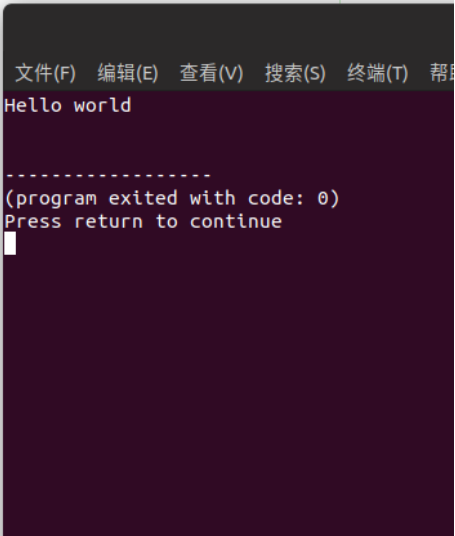
直接编译运行：

```
pu.cpp x hwhb.c x hwc.c x
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    char* msg = "Hello World";
    int len = 11;
    int result = 0;

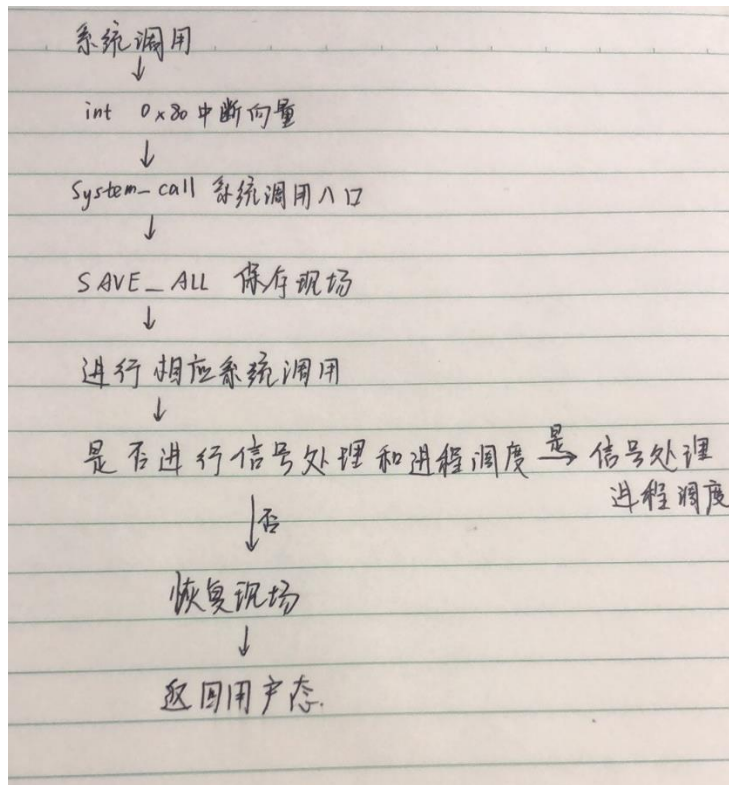
    asm volatile ("movl %2, %%edx;\n\r"
                  "movl %1, %%ecx;\n\r"
                  "movl $1, %%ebx;\n\r"
                  "movl $4, %%eax;\n\r"
                  "int $0x80"
                  : "=m"(result)
                  : "m"(msg), "r"(len)
                  : "%eax");

    return 0;
}
```



```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
Hello world
-----
(program exited with code: 0)
Press return to continue
```

(3) 系统调用实现的流程图



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。


实验代码如下：

```

cpu.cpp ✕
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <sys/time.h>
5  #include <assert.h>
6
7
8  int 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         sleep(1);
17         printf("%s\n",str);
18     }
19     return 0;
20 }
21
22
23

```

编译并在终端运行：



```

rdj@rdj-VirtualBox: ~/桌面
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
rdj@rdj-VirtualBox:~/桌面$ ./cpu
usage:cpu<string>
rdj@rdj-VirtualBox:~/桌面$

```

这个程序的功能是虚拟化 cpu，将单个 cpu 虚拟为多个，使四个程序同时进行。当你输入不为空时，每隔一秒循环输出你所输入的字符。

输入命令 `./cpu A & ./cpu B & ./cpu C & ./cpu D &`，运行结果如下图：

```
rdj@rdj-VirtualBox: ~/桌面
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
rdj@rdj-VirtualBox:~/桌面$ ./cpu
usage:cpu<string>
rdj@rdj-VirtualBox:~/桌面$ ./cpu A &./cpu B &./cpu C &./cpu D &
[1] 2933
[2] 2934
[3] 2935
[4] 2936
rdj@rdj-VirtualBox:~/桌面$ B
A
C
D
B
A
C
D
B
A
C
D
B
A
C
D
B
```

程序 cpu 运行了 4 次, 分别是输入 A、B、C、D 这四个字母所执行的这四次。顺序是 BACD, 执行顺序是随机的、无规律的。程序并发执行是一个程序段的执行尚未结束, 另一个程序段的执行已经开始。这个程序的功能是虚拟化 cpu, 将单个 cpu 虚拟为多个, 使四个程序同时进行。当你输入不为空时, 每隔一秒循环输出你所输入的字符。当输入 ABCD 这四个程序并发执行时, 前一个程序还未执行完, 即时间间隔中下一个程序开始执行, 四个程序以此类推交叉执行, 于是便出现了 BACD 四个字母交替出现的结果。

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 &

实验代码如下:

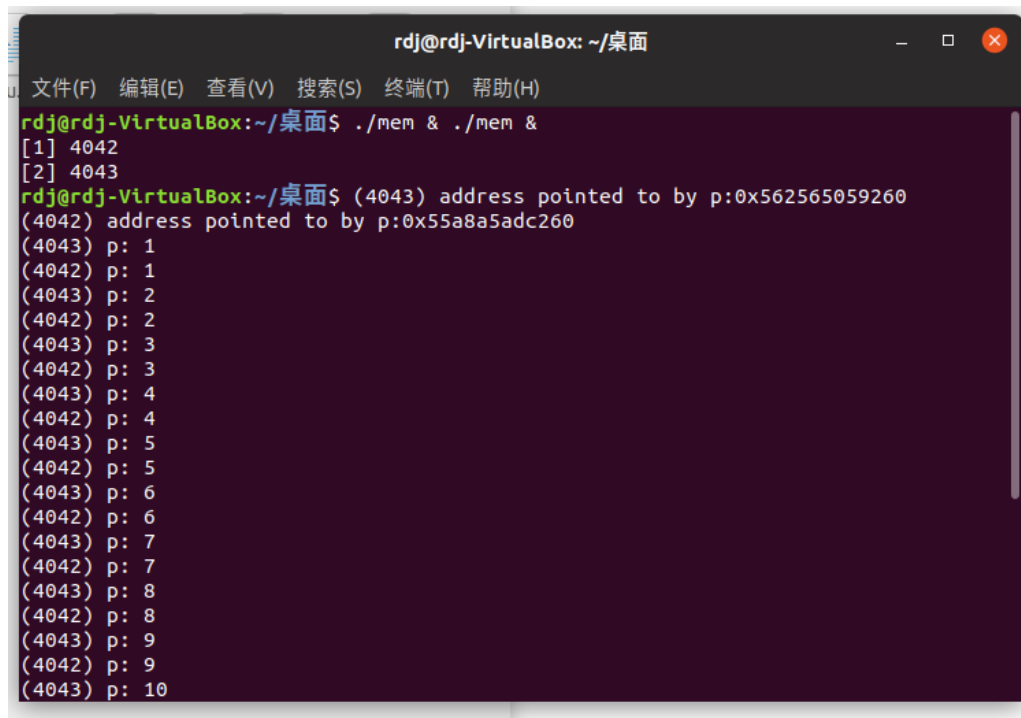
```
mem.c x
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <assert.h>
5
6  int main(int argc, char *argv[])
7  {
8      int *p = malloc(sizeof(int)); //a1
9      assert(p!=NULL);
10     printf("(%) address pointed to by p:%p\n", getpid(), p); //a2
11     *p=0;
12     while(1){
13         sleep(1);
14         *p=*p+1;
15         printf("(%) p: %d\n", getpid(), *p); //a4
16     }
17     return 0;
18 }
19
```

编译并在终端运行：

```
rdj@rdj-VirtualBox: ~/桌面
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
rdj@rdj-VirtualBox:~/桌面$ ./mem
(3978) address pointed to by p:0x55aae606b260
(3978) p: 1
(3978) p: 2
(3978) p: 3
(3978) p: 4
(3978) p: 5
(3978) p: 6
(3978) p: 7
(3978) p: 8
(3978) p: 9
(3978) p: 10
(3978) p: 11
(3978) p: 12
(3978) p: 13
(3978) p: 14
(3978) p: 15
(3978) p: 16
(3978) p: 17
(3978) p: 18
(3978) p: 19
(3978) p: 20
(3978) p: 21
(3978) p: 22
```

首先给 p 分配内存空间，第一行输出为：前面括号内的是进程识别码，先分配内存，然后输出指针 p 的首地址。令 p 指针内容为 0，然后每隔一秒令 p 指针内容加一，前面括号内的仍为进程识别码，冒号后输出 p 指针的内容，即 1、2、3……

输入命令 ./mem & ./mem &，运行结果如下：



```
rdj@rdj-VirtualBox: ~/桌面
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
rdj@rdj-VirtualBox:~/桌面$ ./mem & ./mem &
[1] 4042
[2] 4043
rdj@rdj-VirtualBox:~/桌面$ (4043) address pointed to by p:0x562565059260
(4042) address pointed to by p:0x55a8a5adc260
(4043) p: 1
(4042) p: 1
(4043) p: 2
(4042) p: 2
(4043) p: 3
(4042) p: 3
(4043) p: 4
(4042) p: 4
(4043) p: 5
(4042) p: 5
(4043) p: 6
(4042) p: 6
(4043) p: 7
(4042) p: 7
(4043) p: 8
(4042) p: 8
(4043) p: 9
(4042) p: 9
(4043) p: 10
```

此时为两个 mem 程序并发执行，每个程序的进程识别码分别显示在前面的括号中，每秒钟显示两个程序各自的 p 的内容。

两个程序分配的内存地址不相同，不共享同一块物理内存区域。从第一行显示的首地址可以看出两个指针的首地址不同。

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。）
- 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。

实验代码如下：


```
thread.c ✕
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <unistd.h>
5
6  volatile int counter = 0;
7  int loops;
8
9  void *worker(void *arg){
10     int i;
11     for (i= 0; i<loops;i++){
12         counter ++;
13     }
14     return NULL;
15 }
16
17 int 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;
33 }
```

在终端编译并执行命令 1:

```
rdj@rdj-VirtualBox: ~/桌面
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
rdj@rdj-VirtualBox:~/桌面$ gcc -o thread thread.c -Wall -pthread
rdj@rdj-VirtualBox:~/桌面$ ./thread 1000
Initial value : 0
Final value :2000
rdj@rdj-VirtualBox:~/桌面$
```

执行命令 2:

```
rdj@rdj-VirtualBox: ~/桌面
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
rdj@rdj-VirtualBox:~/桌面$ gcc -o thread thread.c -Wall -pthread
rdj@rdj-VirtualBox:~/桌面$ ./thread 1000
Initial value : 0
Final value :2000
rdj@rdj-VirtualBox:~/桌面$ ./thread 100000
Initial value : 0
Final value :200000
rdj@rdj-VirtualBox:~/桌面$
```

此程序是多线程程序，主程序使用 `pthread.create()` 创建两个线程，每个线程在一个名为 `worker()` 的历程中进行，计数器统计循环次数循环递增。

执行结果与输入的参数成同一数量级

变量 `p1`、`p2` 是各个线程共享的