北京交通大学

《操作系统》实验报告一

学　号：　16281049

姓　名：　王晗炜

专　业：　计算机科学与技术

学　院：　计算机与信息技术学院

提交日期：　2019 年 03 月 11 日

一、（系统调用实验）了解系统调用不同的封装形式

1.1 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？)。

- 调用 API 接口函数：

源代码 **getpid.c**:

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

int main(){
    pid_t pid;
    pid = getpid();
    printf("%d\n",pid);
    return 0;
}
```

在 linux 系统上编译并运行：

```
➜ lab1 gcc -o getpid getpid.c -Wall
➜ lab1 ./getpid
4588
```

- 调用汇编中断：

源代码 **assembly.c**:

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

int main(){
    pid_t pid;
    asm volatile(
        "mov $0,%%ebx\n\t"
        "mov $0*14,%%eax\n\t"
        "int $0*80\n\t"
        "mov %%eax,%0\n\t"
```

```
        :"=m"(pid)
        );
    printf("%d\n",pid);
    return 0;
}
```

在 linux 上编译并运行：

```
→ lab1 gcc -o assembly assembly.c -Wall
→ lab1 ./assembly
4937
```

由以上运行结果可知这两种方式都可以顺利获取进程号，其中第一种方式直接调用了 linux 系统内封装好的 getpid()函数，第二种方式选用汇编语言中的软中断方式，调用 0×80 号中断，在 C 源文件中嵌入汇编语言完成中断调用。

通过查阅 Linux 系统中的 unistd.h 头文件可以找到为 getpid 设置的系统调用号为 172（其查找过程如下）

```
→ include cd /usr/include/asm-generic/
→ asm-generic cat unistd.h | grep getpid
#define __NR_getpid 172
__SYSCALL(__NR_getpid, sys_getpid)
```

而通过阅读给出的汇编代码我们可以发现其中断向量号之后的功能号并不为 172，而是 0×14，即 20。但程序并没有出现问题，仍成功返回了我们需要的进程号。之后自己上网查阅了相关资料，发现 32 位和 64 位的 Linux 系统中的系统调用号并不相同，并分别存储在 unistd_32.h 和 unistd_64.h 头文件中，在 32 位系统中 getpid 的系统调用号为 20，与我们编写的汇编程序一致，64 位系统中的 getpid 的系统调用号为 39。而对于 64 位的操作系统，基本上已经抛弃

了 int 80 这种老旧的系统调用的方式，取而代之的是 syscall 这个函数。但如果仍在 64 位系统中使用 int 80 这种软中断方式进行系统调用，便还是遵循原先 32 位系统中的系统调用号，这算是 64 位系统对之前系统的一种兼容。至此前面的疑问已经能得到解释。

1.2 上机完成习题 1.13

- 首先直接使用 C 语言实现需求：

源代码 **hello_linux.c**:

```c
#include <stdio.h>

int main(){
    printf("Hello Linux\n");
    return 0;
}
```

在 Linux 系统中使用 gcc 编译并执行：

```
→ lab1 gcc -o hello_linux hello_linux.c -Wall
→ lab1 ./hello_linux
Hello Linux
```

可知得到了对应的结果

- 随后使用汇编语言实现需求：

源代码 **hello_linux.asm**:

```
section data
string db "Hello Linux"
len equ $-msg
section .text
    global _start
_start:
    mov eax,4
```

```
    mov ebx,1
    mov ecx,string
    mov edx,len
    int 0x80
    mov eax,1
    mov ebx,0
    int 0x80
```

使用 nasm 编译并运行：

```
→ lab1 nasm -f elf64 hello_linux.asm
→ lab1 ld -s -o hello_linux hello_linux.o
→ lab1 ./hello_linux
Hello Linux
```

可见输出结果正确，汇编调用成功

1.3 阅读 pintos 操作系统源代码，画出系统调用实现的流程图。

首先可以在 github 上的 Pintos 项目中查阅其源代码，在 Pintos/src/lib/syscall-nr.h 中我们可以看到一系列的系统调用号：

```
{
  /* Projects 2 and later. */
  SYS_HALT,                   /* Halt the operating system. */
  SYS_EXIT,                   /* Terminate this process. */
  SYS_EXEC,                   /* Start another process. */
  SYS_WAIT,                   /* Wait for a child process to die. */
  SYS_CREATE,                 /* Create a file. */
  SYS_REMOVE,                 /* Delete a file. */
  SYS_OPEN,                   /* Open a file. */
  SYS_FILESIZE,               /* Obtain a file's size. */
  SYS_READ,                   /* Read from a file. */
  SYS_WRITE,                  /* Write to a file. */
  SYS_SEEK,                   /* Change position in a file. */
  SYS_TELL,                   /* Report current position in a file. */
  SYS_CLOSE,                  /* Close a file. */

  /* Project 3 and optionally project 4. */
```

```
    SYS_MMAP,                   /* Map a file into memory. */
    SYS_MUNMAP,                 /* Remove a memory mapping. */

    /* Project 4 only. */
    SYS_CHDIR,                  /* Change the current directory. */
    SYS_MKDIR,                  /* Create a directory. */
    SYS_READDIR,                /* Reads a directory entry. */
    SYS_ISDIR,                  /* Tests if a fd represents a directory.
*/
    SYS_INUMBER                 /* Returns the inode number for a fd. */
  };
```

这个调用表中给出了 pintos 系统能实现的所有系统调用

随后继续打开 Pintos/src/lib/user/syscall.c，里面先给出了 4 中系统调用函数的调用方式，分别为 syscall0,syscall1,syscall2,syscall3，每种方式需要传入的参数不同，但所有方式都必需传入系统调用号。

其后还给出了与上表对应的封装完成的函数，系统在使用这些函数的时候简洁调用了以上定义的四种系统调用函数，以下为此源文件的部分代码：

```
#define syscall0(NUMBER)                                        \
       ({                                                      \
         int retval;                                           \
         asm volatile                                          \
           ("pushl %[number]; int $0x30; addl $4, %%esp"       \
              : "=a" (retval)                                  \
              : [number] "i" (NUMBER)                          \
              : "memory");                                     \
         retval;                                               \
       })

/* Invokes syscall NUMBER, passing argument ARG0, and returns the
   return value as an `int'. */
#define syscall1(NUMBER, ARG0)                                          \
       ({                                                              \
         int retval;                                                   \
         asm volatile                                                  \
```

```
          ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp"
\
             : "=a" (retval)                                          \
             : [number] "i" (NUMBER),                                  \
               [arg0] "g" (ARG0)                                       \
             : "memory");                                              \
         retval;                                                       \
       })

/* Invokes syscall NUMBER, passing arguments ARG0 and ARG1, and
   returns the return value as an `int'. */
#define syscall2(NUMBER, ARG0, ARG1)                                   \
       ({                                                          \
         int retval;                                               \
         asm volatile                                              \
           ("pushl %[arg1]; pushl %[arg0]; "                         \
            "pushl %[number]; int $0x30; addl $12, %%esp"            \
             : "=a" (retval)                                         \
             : [number] "i" (NUMBER),                                \
               [arg0] "g" (ARG0),                                    \
               [arg1] "g" (ARG1)                                     \
             : "memory");                                            \
         retval;                                                   \
       })

/* Invokes syscall NUMBER, passing arguments ARG0, ARG1, and
   ARG2, and returns the return value as an `int'. */
#define syscall3(NUMBER, ARG0, ARG1, ARG2)                             \
       ({                                                          \
         int retval;                                               \
         asm volatile                                              \
           ("pushl %[arg2]; pushl %[arg1]; pushl %[arg0]; "        \
            "pushl %[number]; int $0x30; addl $16, %%esp"            \
             : "=a" (retval)                                         \
             : [number] "i" (NUMBER),                                \
               [arg0] "g" (ARG0),                                    \
               [arg1] "g" (ARG1),                                    \
               [arg2] "g" (ARG2)                                     \
             : "memory");                                            \
         retval;                                                   \
       })
void
exit (int status)
{
```

```
  syscall1 (SYS_EXIT, status);
  NOT_REACHED ();
}

pid_t
exec (const char *file)
{
  return (pid_t) syscall1 (SYS_EXEC, file);
}

int
wait (pid_t pid)
{
  return syscall1 (SYS_WAIT, pid);
}

bool
create (const char *file, unsigned initial_size)
{
  return syscall2 (SYS_CREATE, file, initial_size);
}

bool
remove (const char *file)
{
  return syscall1 (SYS_REMOVE, file);
}
```

随后我们进一步在该源代码中的中的 Pintos/src/userprog/syscall.c 文件中找到了中断初始函数：

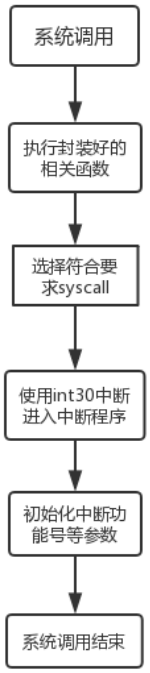
```
void
syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
}
```

在此函数中我们可以发现中断调用的中断号为 0×30，并允许中断嵌套，根据其

文件所 include 的文件，我们找到 threads/interrupt.c 文件，其中 intr_register_int 函数的定义如下：

```
void
intr_register_int (uint8_t vec_no, int dpl, enum intr_level level,
                   intr_handler_func *handler, const char *name)
{
  ASSERT (vec_no < 0x20 || vec_no > 0x2f);
  register_handler (vec_no, dpl, level, handler, name);
}
```

这里传入的参数符合断言要求，因此又进入了 register_handler 函数，此函数中使用的数组则涉及到 intr-stubs.S 汇编文件的数组，其中的信息包括了调用的功能号及其他限制信息。至此系统调用的过程全部完成。以下为流程图：



二、（并发实验）根据以下代码完成下面的实验。

2.1 编译运行该程序（cpu.c），观察输出结果，说明程序功能。

源程序 **cpu.c**：

```c
#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];
    while(1){
        sleep(1);
        printf("%s\n",str);
    }
    return 0;
}
```

使用 gcc 编译运行：

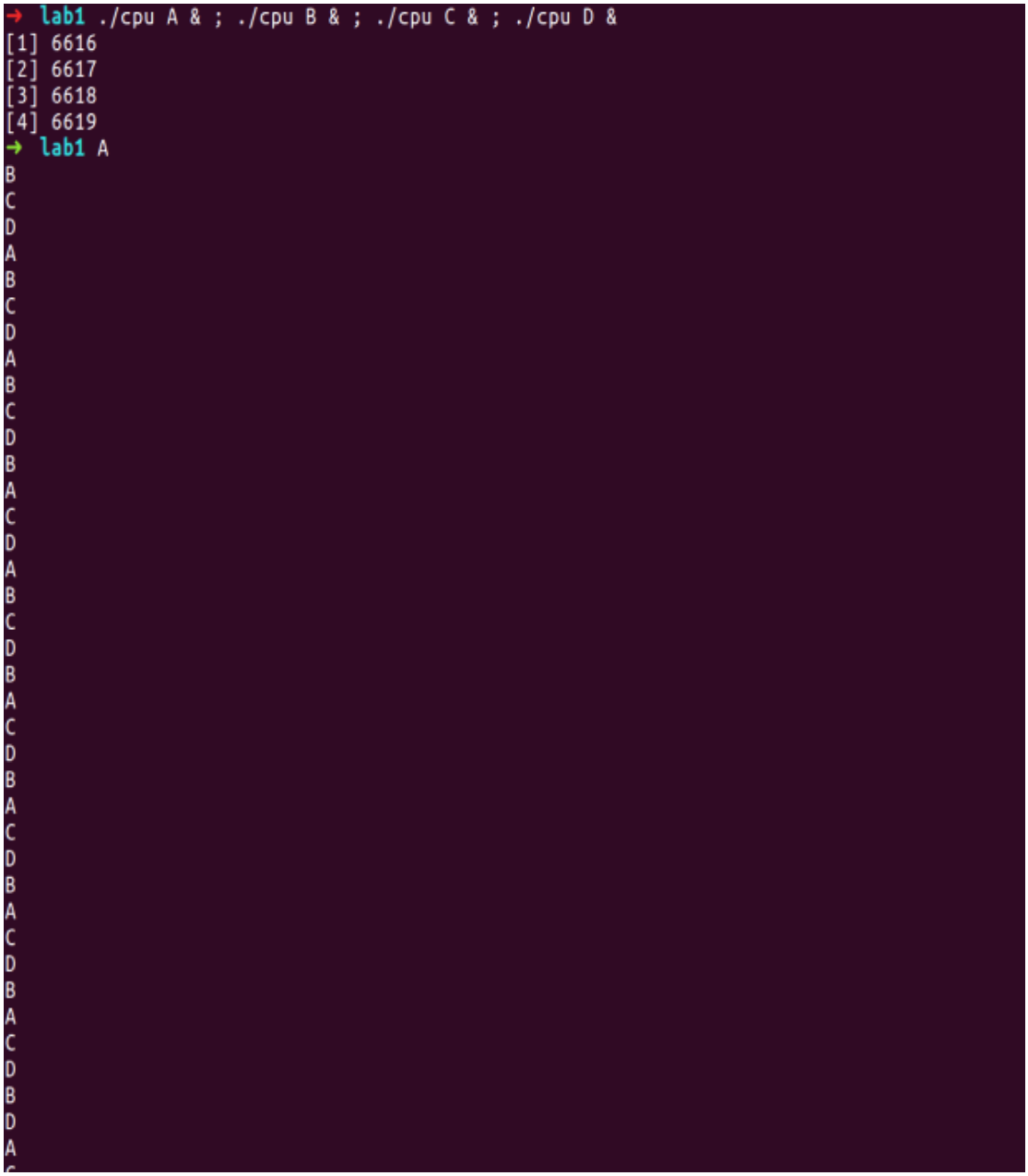
```
➜  lab1 gcc -o cpu cpu.c -Wall
➜  lab1 ./cpu
usage:cpu<string>
```

由上图可见程序运行成功，通过分析 C 语言源代码可知，此程序共有两个功能，当不输入参数或输入多个参数运行时，会直接在屏幕上打印出此程序需要传入的参数和用处，即打印“usage:cpu<string>”，当传入参数字符串时，便会直接在屏幕上循环打印输入的字符串，以一秒为间隔。

2.2 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。

按照要求运行程序：



从上图可以得知系统使用了四个进程运行该程序，每隔一秒在屏幕上打印一次 ABCD，由于程序未设置终止条件，所以程序会一直运行下去，但程序运行的顺序并不确定。这是因为如今的操作系统在同时运行多个程序时会并发执行而不

是顺序执行，也就是具有并发的特征。程序在并发执行时，是多个程序共享系统中的各种资源，因而这些资源的状态将由多个程序来改变，致使程序的运行失去了封闭性。这样，某程序在执行时，必然会受到其它程序的影响。例如，当处理机这一资源已被某个程序占有时，另一程序必须等待。程序在并发执行时，由于失去了封闭性，也将导致其再失去可再现性。因此每一轮程序的运行顺序都会不尽相同。

三、（内存分配实验）根据以下代码完成实验。

3.1 阅读并编译运行该程序(mem.c)，观察输出结果，说明程序功能。

程序源代码 **mem.c**：

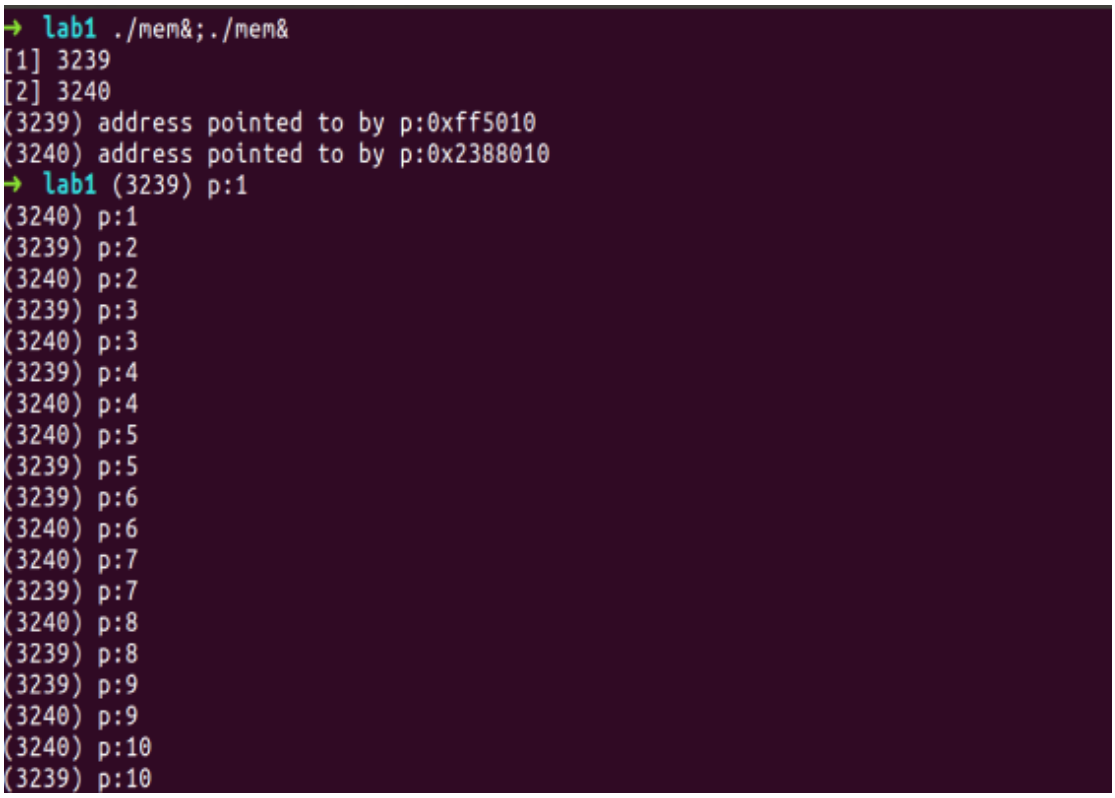
```c
#include <unistd.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

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
    *p = 0;//a3
    while(1){
        sleep(1);
        *p = *p + 1;
        printf("(%d) p:%d\n",getpid(),*p);//a4
    }
    return 0;
}
```

在 Linux 系统下使用 gcc 编译并运行：

由上图的运行结果结合源码进行分析可知道，该程序先为一个指针申请了一个整数字符空间大小的空间，随后打印出进程号和申请的内存空间的首地址并将内存中储存的整数字符置为 0。之后每隔一秒将该整数加 1 并打印其值于屏幕上，循环往复不设终结。

3.2 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？

按照题目要求同时运行两个 mem 程序



由上图的结果我们可以得知两个进程分别给与了不同的内存地址，其操作的数值互不干扰，并发运行。表面上看并没有共享一块内存地址。通过查阅相关资料，Linux 中的进程内存管理采用的是虚拟内存管理技术，Linux 操作系统采用虚拟内存管理技术，使得每个进程都有各自互不干涉的进程地址空间。该空间是块大小为 4G 的线性虚拟空间，用户所看到和接触到的都是该虚拟地址，无法看到实际的物理内存地址。利用这种虚拟地址不但能起到保护操作系统的效果（用户不能直接访问物理内存），而且更重要的是，用户程序可使用比实际物理内存更大的地址空间。

每一个进程都运行在一个属于它自己的内存沙盘中，这个沙盘就是虚拟地址空间，在 32 位模式下它总是一个 4GB 的内存地址块。这些虚拟地址通过页表映射到物理内存，页表由操作系统维护并被处理器引用。因此虽然两个进程的虚拟内存地址并不相同，也不代表它们的实际内存地址不同，但它们一定都应用的

是同一块连续的内存块。

四、（共享的问题）根据以下代码完成实验。

4.1 阅读并编译运行该程序，观察输出结果，说明程序功能。

程序源代码 **thread.c**:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.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;
}
```

在 Linux 下使用 gcc 编译并运行：

```
➜ lab1 gcc -o thread  thread.c -Wall -pthread
➜ lab1 ./thread
usage:threads<value>
➜ lab1 ./thread 1000
Initial value:0
Final value:2000
```

由以上结果结合源代码分析可知，当不输入任何参数或者输入多个参数时程序会进行报错，在屏幕上打印出程序的使用方法。当输入参数（如 1000）时，程序会使用两个进程对共同变量 counter 变量进行循环加 1，循环次数为 loops 的值，即输入参数的大小，最后将 counter 的值打印在屏幕上。

4.2 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）

```
➜ lab1 ./thread 100000
Initial value:0
Final value:100522
➜ lab1 ./thread 1000000
Initial value:0
Final value:1214831
➜ lab1 ./thread 10000000
Initial value:0
Final value:10906392
```

当输入参数 100000 之后，可以发现打印出来的最后结果并不为 200000，而是远小于它，随后自己尝试了更大的参数，结果也是如此。通过查找相关资料可以得知 volatile 关键字可以被解释为“直接存取原始内存地址”，因此每次编译器就会直接从内存地址中读取该关键字修饰的变量的值，当两个线程都要用到某一个变量且该变量的值会被改变时，应该用 volatile 声明，该关键字的作用是防止优化编译器把变量从内存装入 CPU 寄存器中。如果变量被装入寄存器，那么两个线程有可能一个使用内存中的变量，一个使用寄存器中的变量，这会造

成程序的错误执行。

但单纯使用该关键字而没有锁的机制，会是进程读出脏数据，即过时的数据，在本程序中可表现为进程 1 刚从内存中读出数据并加 1，在它保存回其内存之前，进程 2 就已经读出了内存中的数值并进行运算，所以本该有两次加法运算的值最后的结果只进行了一次运算，造成了最终结果远小于 loops 值的两倍。