实验报告

课程名称：操作系统

任课教师：何永忠

学生姓名：张云肖

学生学号：16281244

专业年级：安全 1601 班

学院名称：计算机学院

2018 年 3 月 13 日

实验名称:操作系统初步

班级：安全 1601　　姓名：张云肖　　学号：16281244

1. （系统调用实验）了解系统调用不同的封装形式。

1.1. 实验原理

1.1.1. API

API（Application Programming Interface,应用程序编程接口），指的是我们用户程序编程调用的如 read(),write(),malloc(),free()之类的调用的是 glibc 库提供的库函数。API 直接提供给用户编程使用，运行在用户态。这里要另外提一下，POSIX 针对 API 提出标准，即针对 API 的函数名，返回值，参数类型进行规范约束，但是并不管 API 具体如何实现。

1.1.2. 系统调用

系统操作系统为用户态进程与硬件设备进行交互提供了一组接口——系统调用，用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务。通过软中断或系统调用指令向内核发出一个明确的请求，内核将调用内核相关函数来实现（如 sys_read(),sys_write()）。用户程序不能直接调用这些 Sys_read,sys_write 等函数。这些函数运行在内核态。

1.1.3. 两者间区别与联系

区别：API 只是一个函数定义，系统调用通过软终端向内核发出一个明确的请求
联系：

1. Libc 库定义了一些 API 引用了封装例程，目的在于发布系统调用。
2. 一般每个系统调用对应了一个封装例程，库再用这些封装例程定义出给用户的 API。通常 API 函数库中的函数会调用封装例程，封装例程负责发起系统调用，这些都运行在用户态；内核开始接受系统调用后，CPU 从用户态切换到内核态；内核调用相关的内核函数来处理再逐步返回给封装例程，cpu 进行一次内核态到用户态的切换，API 函数从封装例程拿到结果，在处理完毕后返回用户。
3. 当 API 函数不一定都需要进行系统调用

1.1.4. 系统调用号

一个系统调用号，对应一个函数入口地址，glibc 和内核里面的这个系统调用号是一致的，所以 glibc 调用汇编之类把系统调用号传给内核的时候，内核找到这个具体的系统调用服务

例程对应的函数入口地址，如 sys_read。

1.1.5. 系统调用和函数调用的区别

a. 系统调用和库函数的关系

- 系统调用通过软终端 int 0x80 从用户态进入内核态。
- 函数库中的某些函数调用了系统调用
- 函数库中的函数可以没有调用系统调用，也可以调用多个系统调用
- 编程人员可以通过函数库调用系统调用
- 高级编程也可以直接采用 int 0x80 进入系统调用，而不必通过函数库作为中介
- 如果是在核心编程，也可以通过 int 0x80 进入系统调用，此时不能使用函数库。因为函数库中的函数是内核访问不到的，

b. 从用户调用库函数到系统调用执行的流程

- 用户调用 getpid（）库函数
- 库函数执行 int 0x80 中断，由于中断使得进程从用户态进入内核态，参数通过寄存器传送
- 0x80 中断对应的中断例程被称为 system call handler。其工作为
  存储大多数寄存器到内核堆栈中，这是汇编代码写的
  执行真正的系统调用函数-system call service routine，这是 C 代码
  通过 ret_from_sys_call()返回，回到用户态的库函数，这是汇编代码写的

1.1.6. 软中断

a. 软中断与硬中断

硬中断：由与系统相连的外设（比如网卡、硬盘）自动产生的。主要是用来通知操作系统系统外设的变化，比如网卡收到数据包的时候，就会发送一个中断。通常我们说的中断指的是硬中断（hardirq）

软中断：为了满足实时系统的要求，中断处理应该是越快越好。Linux 为了实现这个特点，当中断发生的时候硬中断处理那些短时间就可以完成的工作，而将那些处理事情比较长的工作，放到中断之后来完成，也就是软中断（softirq）

b. 软中断指令

Int 是软中断指令。中断向量表是中断号和中断处理函数地址的对应表，int n-触发软中断 n。相应的中断处理函数的地址为：中断向量表地址+4*n

1.1.7. int 和函数调用 call 的区别

中断 int 过程：取得中断类型码，把标志位压入栈指中，把 CD 压入栈中，把 IP 压入栈中，更改 CS 和 IP，转到中断程序。

而 CALL 是将当前 IP 或者 CS 和 IP 压入栈中，到底是把 IP 还是 IP 和 CS 压入栈中。就要看 CALL 是一个字还是 2 个字。

1.2. Problem a

参考下列网址中的程序。阅读分别运行用 API 接口函数 getpid()直接调用和用汇编中断调用两种方法调用 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 系统调用的终端向量号是多少。

1.2.1. 实验过程

- 编写运用 API 接口函数 getpid()直接调用系统调用，代码如下

root@cloudness-VirtualBox: /home/cloudness/桌面/操作系统
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

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

int main(){
pid_t pid;

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

return 0;
}
```

- 编写汇编中断调用 Linux 操作系统同一个系统调用，代码如下

root@cloudness-VirtualBox: /home/cloudness/桌面/操作系统
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

```
#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;
}
```

- 编译程序，并运行

```
root@cloudness-VirtualBox:/home/cloudness/桌面/操作系统# gcc getpid_call.c -o getpid_call
root@cloudness-VirtualBox:/home/cloudness/桌面/操作系统# gcc getpid_int.c -o getpid_int
root@cloudness-VirtualBox:/home/cloudness/桌面/操作系统# ./getpid_call
4026
root@cloudness-VirtualBox:/home/cloudness/桌面/操作系统# ./getpid_int
4031
```

- 查询 getpid 函数的系统调用号

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

1.2.2. 实验结果

a. getpid()的系统调用号是 172，Linux 系统调用的中断向量号是 0x80。
b. getpid_call 代码解析：pid 用来保存 API 函数 getpid()返回的进程号
c. getpid_int 代码解析：
本程序中嵌入了一段汇编代码

`"mov $0,%%ebx\n\t"`:将 ebx 寄存器清零

`"mov $0,%%ebx\n\t"`:将 20（getpid 系统调用号为 20）进入 eax 寄存器

`"int $0x80\n\t"`:发送系统调用指令

```
"mov %%eax,%0\n\t"
:"=m"(pid)
```
：将 eax 寄存器的值（其中保存的是系统调用返回值）存入变量 pid

1.3. Problem b

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

1.3.1. 实验过程

- 用 linux 系统编写 C 函数形式的”Hello World”程序为 hello_c.c

```
lo_c.c
#include <stdio.h>
int main(){
        printf("Hello World!\n");
        return 0;
}
```

- 编译源码为 hello_c，并执行文件，屏幕打印”Hello World!”

```
→ 操作系统 gcc hello_c.c -o hello_c
→ 操作系统 ./hello_c
Hello World!
```

- 用 linux 系统编写汇编代码形式的”Helloworld”程序为 hello_asm.asm

```
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
```

- 编译源码为 hello_asm，并执行文件，屏幕打印“Hello，World！”

```
→  操作系统 nasm -f elf64 -g -F stabs hello_asm.asm -o hello_asm.o
→  操作系统 ld -o elf_i386 -o hello_asm hello_asm.o
→  操作系统 ./hello_asm
Hello World!
```

1.4. Problem c

阅读 pintos 操作系统源代码，画出系统调用实现的流程图。

1.4.1. Pintos 简介

Pintos 是 80*86 架构的简单操作系统框架。支持内核线程，加载和运行用户程序以及文件系统。我们可以通过 http://cs140.stanford.edu/pintos.tar.gz 获取 pintos

1.4.2. Pintos/src/

Pintos/src 中看到的目录结构有：“threads/”, “userprg/”, “vm/”, “filesys/”, “lib/”, “lib/kernel/”, “lib/user/”, “lib/tests/”, “eamples/”, “utils/”。
其中/Pintos/src/lib 用于标准 C 库的子集实现，此目录中的代码被编译到 Pintos 内核中，并在运行的用户程序中。在内核代码和用户程序中，头文件在这些目录下课可以使用#include<...>标头，你几乎不需要修改此代码。

1.4.3. Pintos/src/bin/lib/user/syscall.c

- 此文件中，syscall.h 里定义了 4 个宏，分别是 syscall0(NUMBER), syscall1(NUMBER,ARG0), syscall2(NUMBER,ARG0,ARG1), syscall3(NUMBER,ARG0,ARG1,ARG2)

```
#include <syscall.h>
#include "../syscall-nr.h"

/* Invokes syscall NUMBER, passing no arguments, and returns the
   return value as an `int'. */
#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;                                                       \
        })

#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] "r" (ARG0),                                     \
                 [arg1] "r" (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] "r" (ARG0),                                     \
                 [arg1] "r" (ARG1),                                     \
                 [arg2] "r" (ARG2)                                      \
               : "memory");                                             \
          retval;                                                       \
        })
```

他们的作用是形成相应的系统调用函数原型，供我们在程序中调用。我们很容易就能发现规律，syscall 后面的数字和 ARGN 的数目一样多。事实上，syscall 后面跟的数字指明了展开后形成函数的参数的个数。同时，我们可以发现四个宏定义函数的系统调用方式的中断向

量号均为 0x30。

- 该文件定义了 20 中系统调用函数

```
void halt (void) NO_RETURN;
void exit (int status) NO_RETURN;
pid_t exec (const char *file);
int wait (pid_t);
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 length);
int write (int fd, const void *buffer, unsigned length);
void seek (int fd, unsigned position);
unsigned tell (int fd);
void close (int fd);

/* Project 3 and optionally project 4. */
mapid_t mmap (int fd, void *addr);
void munmap (mapid_t);

/* Project 4 only. */
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);
```

1.4.4. Pintos/src/userprog/syscall.c

此文件中定义了 syscall_init, syscall_handler 两个函数

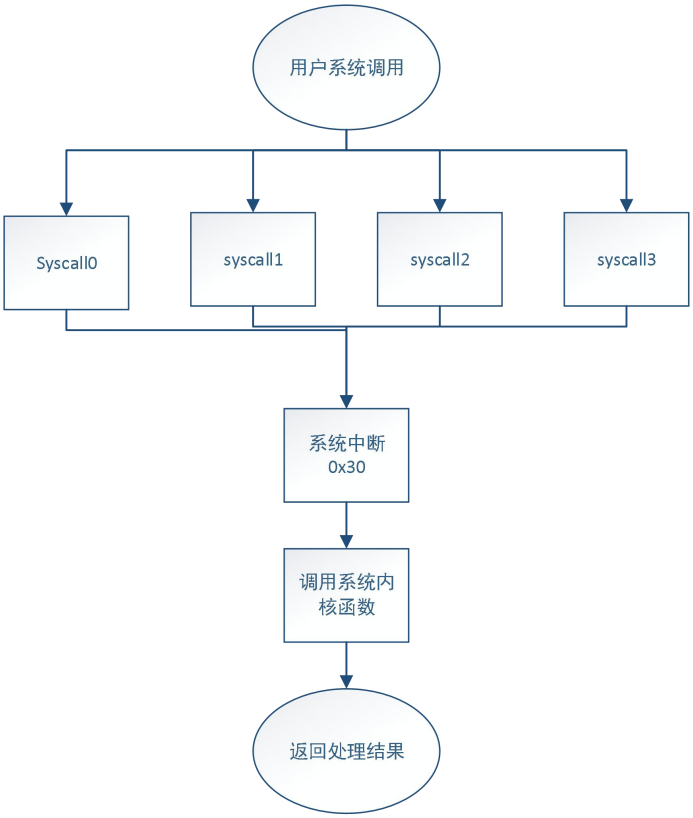
```
void
syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
}

static void
syscall_handler (struct intr_frame *f UNUSED)
{
  printf ("system call!\n");
  thread_exit ();
}
```

Syscall_init:负责系统调用初始化工作，并与函数内部调用了 inter_register_int 函数，注册软中断，从而调用系统调用处理函数。

Syscall_handler:处理系统调用

1.4.5. 系统调用实现流程图



2. （并发实验）根据以下代码完成下面的实验

2.1. Problem a

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）

2.1.1. 实验过程

- 编写 cpu.c 程序，代码如下

```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;
}
```

➢ 编译并执行 cpu.c 程序，查看结果

```
➜  ~ cd /home/cloudness/桌面/操作系统
➜  操作系统 vim cpu.c
➜  操作系统 gcc cpu.c -o cpu
➜  操作系统 ./cpu
usage: cpu <string>
➜  操作系统 ./cpu A
A
A
A
A
```

2.1.2. 代码解释

➢ `int main(int argc, char *argv[])`

argc 为命令行总的参数个数

argv[] 为保存命令行参数的字符串指针，其中第 0 个参数是程序的全名，以后的参数为命令行后面跟的用户输入参数。argv 参数为字符串指针数组，其各元素为命令行中各字符串的首地址。指针数组的长度即为参数个数 argc

➢
```c
if (argc != 2) {
fprintf(stderr, "usage: cpu <string>\n");
exit(1);
}
```

当命令行总参数不为 2 时，即当除却程序名外，有且只有一个参数，否则提示“usage:cpu<string>\n”，退出程序。

➢
```
char *str = argv[1];
while (1) {
sleep(1);
printf("%s\n", str);
}
return 0;
```

当除却程序名后只有一个参数时，将该参数赋值给 str。并进入循环，循环中每打印一次参数，线程睡眠 1 秒。

2.1.3. 实验结果

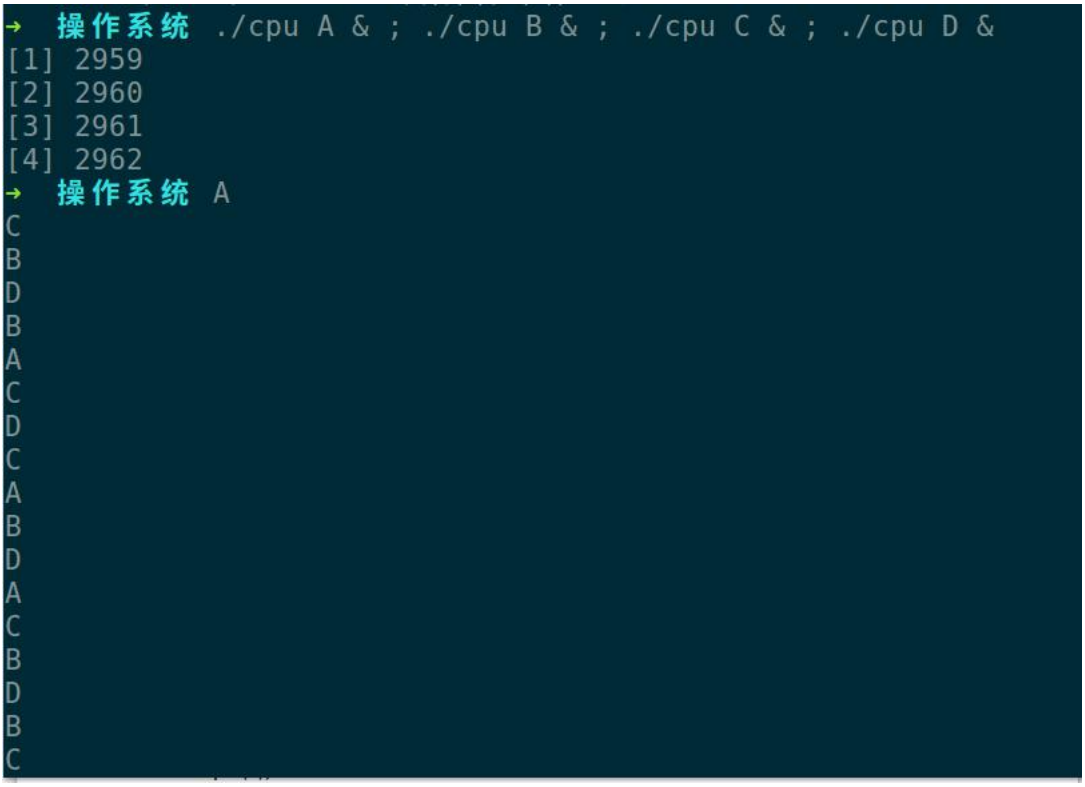
如上所示，当输入./cpu 时出现“usage：cpu<string>\n”的错误提示，表示必须在 cpu 后输入一个参数。当输入./cpu A 后 cpu.c 程序运行，并按照 1 秒钟的时间间隔先屏幕中打印 A。

2.2. Problem b

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。

2.2.1. 实验过程

➢ 输入./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &,并运行



2.2.2. 实验结果

同时并发运行四个程序，发现四个程序的运行顺序没有规律。

2.2.3. 结果分析

在单批道处理系统中，一个作业单独进入内存并独占系统资源，知道运行结束后下一个作业才能够进入系统，当进行 I/O 操作的时候 CPU 处等待状态。它的特征是执行顺序是先进先出的。

用户所提交的作业先放在外存上，并排成一个对列（后备对列），由作业调度程序按照一定的算法，从后备对列中选择若干个作业调入内存，使其共享 CPU 和系统中的各种资源。同时在内存中装入若干程序，这样可以在 A 程序运行时，利用其 IO 操作而暂停的 CPU 空挡时间，再调度另一道程序 B 运行，同样可以利用 B 程序在 IO 操作时调用 CPU 空档调用程序 C 运行，使用多道程序交替运行，始终保持 CPU 忙碌的状态。

多道批处理系统具有多道和成批的特点。多道：在内存中同时存放多个作业，使之同时处于运行状态，这些作业共享 CPU 和外部设备等资源。成批：和他的作业之间没有交互性。用户自己不能干预自己的作业的运行，发现作业错误不能及时改正。

因为多批道处理系统具有成批性，内存中可以同时存在若干道作业，但是用户不能够干预自己作业的运行。并作业通过一定的作业调度算法来使用 CPU 所以作业的执行次序与进入内存次序无严格对应关系。

3. （内存分配实验）根据以下代码完成实验

3.1. Problem a

阅读并编译运行该程序(mem.c)，观察输出结果，说明程序功能。

3.1.1. 实验过程

- 编写 mem.c 程序，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); // a2
*p = 0; // a3
while (1) {
sleep(1);
*p = *p + 1;
printf("(%d) p: %d\n", getpid(), *p); // a4
 }
return 0;
}
```

- 编译并执行 mem.c 程序，查看结果

```
→  操作系统 gcc mem.c -o mem
→  操作系统 ./mem
(3280) address pointed to by p: 0x1db8010
(3280) p: 1
(3280) p: 2
(3280) p: 3
(3280) p: 4
(3280) p: 5
(3280) p: 6
(3280) p: 7
```

3.1.2. 代码解释

-
```c
int *p = malloc(sizeof(int)); // a1
```

为定义一个指针 p 并将一个 int 类型单元的首地址存储到指针变量 p 中

-
```c
assert(p != NULL);
printf("(%d) address pointed to by p: %p\n",getpid(), p); // a2
```

assert 的作用是现计算表达式 p！=NULL，如果其值为假（即为 0），那么它先向 stder 打印出一条出错信息，然后通过调用 abort 来终止程序运行。

getpid 的作用是取得进程码，许多程序利用取得此值来建立临时文件，以避免临时文件带来相同的问题。

%p 的作用是按照十六进制输出地址

这里检查指针 p 不为 NULL，即成功被分配地址后，输出其进程码与指针 p 对应的地址位置

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

将指针 p 执行的 int 空间的值修改为 0，并通过 while 循环每隔 1 分钟对该值进行累加。

3.1.3. 实验结果

此次 mem 程序执行的进程号为 3280，指针 p 所存储的地址是 0x1db8010，对 p 指向的值累加并打印到屏幕成功。

3.2. Problem b

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 &

3.2.1. 实验过程

- 输入./mem &; ./mem &,并运行

3.2.2. 实验结果

同时运行两个程序，发现为两个进程分别分配了两个物理地址，分别为 0xd40010，0xd12010。

3.2.3. 结果分析

操作系统对在进行的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，3-4G 属于内核空间。每个进程得我用户空间不同，但内核空间相同。程序中的 malloc 函数是在用户空间中的堆上通过扩展堆的方式来得到得到虚拟内存地址的返回值，来维持一个页表来进行映射，每次访问空间的某个地址都需要把地址翻译为实际物理内存地址。

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，但是真实的物理地址不会相同，这种虚拟内存也不保证不同进程的相互隔离，错误程序不会干扰别的正确的进程。

4. （共享的问题）根据以下代码完成实验

4.1. 实验原理

4.1.1. Pthread

定义：POSIX threads， 简称 Pthreads，是线程的 POSIX 标准，定义了创建和操纵线程的一套 API。
数据类型：pthread_t:线程 ID， pthread_attr_t:线程属性
操纵函数：
pthread_create():创建一个线程
pthread_exit():终止当前线程
pthread_cancel()：中断当前的线程，知道另一个线程运行结束
pthread_attr_init()：初始化线程的属性
pthread_attr_setdetachstate()：设置脱离状态的属性（决定这个线程在终止时是否可以被结合）
pthread_attr_getdetachstate()：获取脱离状态的属性
pthread_attr_destroy()：删除线程的属性
pthread_kill()：向线程发送一个信号

4.2. Problem a

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）
阅读并编译运行该程序(mem.c)，观察输出结果，说明程序功能。

4.2.1. 实验过程

- 编写 thread.c 程序，代码如下

```c
#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: thread <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;
}
```

- 编译并执行 thread.c 程序，查看结果

```
→  操作系统 gcc -o thread thread.c -Wall -pthread
→  操作系统 ./thread 1000
Initial value: 0
Final value: 2000
```

4.2.2. 代码解释

-
```c
volatile int counter=0;
int loops;
```

定义了两个全局变量 counter、loops

-
```c
void *worker(void *arg)
{
        int i;
        for(i=0;i<loops;i++)
        {
                counter++;
        }
        return NULL;
}
```

定义了 work 函数利用 for 循环对 counter 变量进行 loops 次累加

-
```c
if(argc!=2)
{
        fprintf(stderr,"usage: thread <value>\n");
        exit(1);
}
```

要求输入的参数有且只有一个

-
```
loops=atoi(argv[1]);
pthread_t p1,p2;
printf("Initial value: %d\n",counter);
```

将输入的参数转化为整数型赋值给 loops，并定义两个线程 id 分别为 p1,p2 输出 counter 的初始值。

-
```
pthread_create(&p1,NULL,worker,NULL);
pthread_create(&p2,NULL,worker,NULL);
pthread_join(p1,NULL);
pthread_join(p2,NULL);
```

通过 pthread_create()函数创建两个均进行利用 worker 函数进行 counter 累加工作的线程，然后利用 pthread_join()函数来以阻塞的方式，等待指定线程结束，并回收被等待线程的资源。

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

输出 counter 的累加结果

4.2.3. 实验结果

当输入参数为 1000 时，程序执行得到的累加结果为 2000

4.3. Problem b

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。）

4.3.1. 实验过程

- 分别输入./thread 10000、 ./thread 100000、 ./thread 1000000、./thread 100, ./thread 10 并运行

```
→  操作系统 ./thread 10000
Initial value: 0
Final value: 11852
→  操作系统 ./thread 100000
Initial value: 0
Final value: 112413
→  操作系统 ./thread 1000000
Initial value: 0
Final value: 1184139
→  操作系统 ./thread 100
Initial value: 0
Final value: 200
→  操作系统 ./thread 10
Initial value: 0
Final value: 20
```

4.3.2. 实验结果

发现同时运行两个线程的时候，当输入 loops 参数较小是，counter 的结果基本为 loops 的两倍。当输入的 loops 较大时，会小于两倍。

4.3.3. 结果分析

在该程序中，由 counter、loops 是全局变量，且 worker（）函数是共享的。

但两个线程在同一进程中，并且访问操作的对象是共享的时候，就会发生读脏数据的情况。现代计算机，多采用加锁的机制避免这种问题的发生，保证线程实现的完整性。

当输入参数比较小的手，一个 CPU 的核心足够处理，就是单核 CPU 运行多线程，由于每个核心有内存锁机制，所以计算结果没有问题。但是当输入参数比较大时，就会无法满足，进而导致出现读取脏数据的情况。