沈阳航空航天大学

计算机学院

**实验报告**

|  |  |
| --- | --- |
| 课程名称 | 操作系统 |
| 专 业 | 物联网工程 |
| 班 级 | 2234280102 |
| 学 号 | 223428010210 |
| 学生姓名 | 陈梓欣 |
| 指导教师 | 宗传玉 |
| 实验时间 | 周二7-8节 |
| 实验地点 | 工训楼310 |

# 实验名称

Linux系统内存管理

# 二、实验目的

1. 理解Linux系统段式、页式内存管理方式。
2. 理解Linux系统的二级页表结构及逻辑地址、线性地址、物理地址的映射过程。

# 三、实验内容和要求

1、编写代码实现页式管理分配物理页和释放物理页的过程

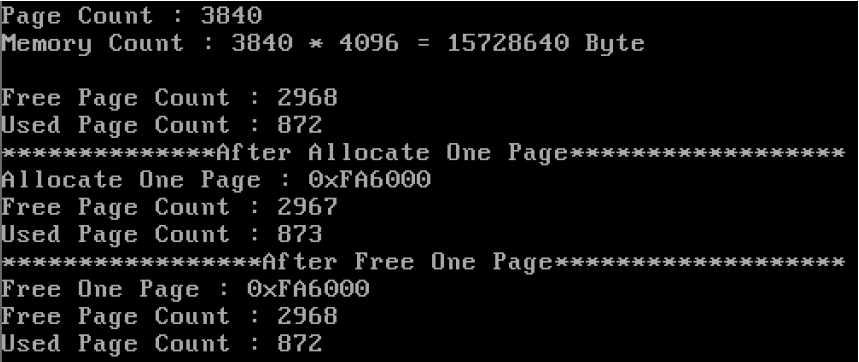
（1）新建一个Linux011 Kernel项目添加一个系统调用内核函数 int dump\_physical\_mem()，该函数功能如下：

①在终端设备上显示当前内存信息，包括物理页总数、空闲页数量和占用页数量；

②调用get\_free\_page函数分配一个空闲的物理页，输出当前空闲页和占用页的数量；

③调用free\_page函数将刚刚分配的物理页进行回收，最后输出当前空闲页和占用页的数量。

（2）启动Linux0.11系统，使用vi编辑器编写程序main.c，该程序调用dump\_physical\_mem()函数，程序执行结果如图28所示。



**图1. 分配物理页和释放物理页的程序执行结果**

实验详细内容请参考《Linux内核实验教程\_2.1》105-106页

2、编写代码输出页目录和页表信息

（1）新建一个Linux011 Kernel项目添加一个系统调用内核函数， 该系统调用的内核函数原型：int sys\_table\_mapping()。该函数首先调用/mm/memory.c中的calc\_mem（）函数将内存中空闲页面的数量以及各个页表（从0号页起）映射的物理页数量输出到文本文件a.txt中，然后依次将下面信息输出到文本文件a.txt中。

①系统当前页目录信息

页目录信息格式为：Page Directory(页目录的物理页框号 | 所在线性地址)：

示例：Page Directory(PFN:0x0 | LA:0x00000000)

②每个页目录项（PDE）信息

页目录项（PDE）信息格式：

PDE: 下标 -> Page Table(页表的物理页框号 | 所在的线性地址)

示例：PDE: 0x0 -> Page Table(PFN:0x1 | LA:0x00001000)

③每个页目录项（PDE）对应的二级页表的页表项（PTE）信息

页表项（PTE）信息格式：

PTE: 下标 -> Physical Page(物理页框号 | 所在的线性地址)

示例：PTE: 0x0 -> Physical Page(PFN:0x0 | LA:0x00000000)

（2）启动Linux0.11系统，使用vi编辑器编写程序meminfo.c，该程序调用sys\_table\_mapping() 函数。程序执行后生成文本文件a.txt示例如图29所示。

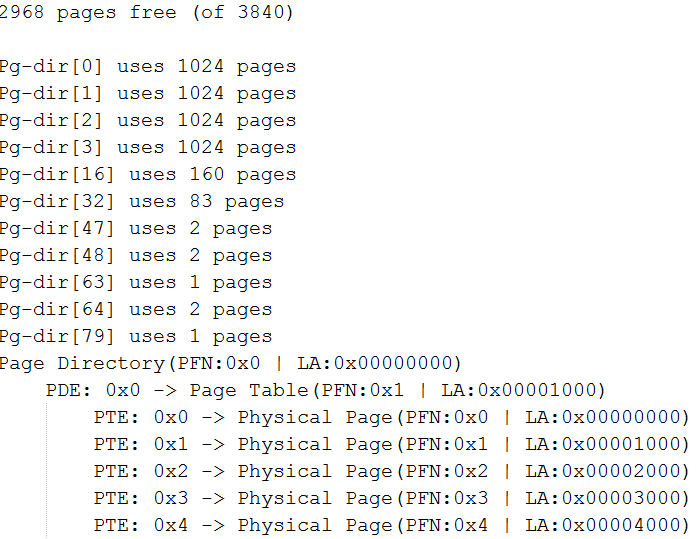


图30. 文本文件a.txt示例

实验详细内容请参考《Linux内核实验教程\_2.1》116-117页

# 四、实验设计

1. 进程与程序的区别

程序是指一个可执行文件，而进程是一个正在执行的程序实例。在 Linux 操作系统中，通过分时技术可以同时运行多个进程。分时技术的核心是将 CPU 的时间划分为短暂的时间片，分配给各个进程轮流运行。虽然某一时刻单核 CPU 上仅有一个进程运行，但由于时间片非常短（通常为几十毫秒），用户感觉多个进程在同时运行。

1. Linux 0.11 内核中的进程

* Linux 0.11 内核最多支持 64 个进程同时存在：
* 手工创建的第一个进程（init 进程）外，其余进程通过调用 fork 系统调用创建。
* 进程通过 进程标识号（PID） 区分。
* 每个进程由以下部分组成：

1. 指令代码
2. 数据区
3. 堆栈区
4. 进程的管理方式
5. Linux 内核通过 进程表 对进程进行管理：
6. 进程表 是一个数组，定义在 kernel/sched.c 文件的第 112 行。
7. 数组大小为 64，每项为一个 task\_struct 结构指针。
8. task\_struct（任务数据结构） 又称为 进程控制块（PCB） 或 进程描述符（PD），定义在 include/linux/sched.h 文件的第 146 行。
9. 主要信息：
10. 进程当前运行状态
11. 信号
12. 进程号及父进程号
13. 运行时间累计值
14. 打开的文件描述符
15. 本任务的局部描述符表（LDT）
16. 任务状态段（TSS）
17. fork 系统调用创建新进程的过程
18. 找到空闲的进程表项：

在进程数组中寻找未被使用的空项，若无空位（已达到 64 个进程），则返回错误。

1. 复制当前进程的 task\_struct：

在主内存中为新进程申请一页，复制当前进程的 task\_struct。

将新进程的 state 设置为 不可中断的睡眠状态，防止调度到未初始化的进程。

1. 初始化新进程的 task\_struct：

将当前进程设置为新进程的父进程。

清除信号位图，重置统计数据。

设置初始运行时间片为 15。

设置 tss 各寄存器值：

tss.eax = 0（子进程 fork 的返回值）

tss.esp0 指向新进程 task\_struct 页面顶部

tss.ss0 设置为内核数据段选择符

若使用协处理器，则保存其状态至 tss.i387。

1. 处理内存与页表：

为新进程设置代码段和数据段描述符。

复制父进程的页表，初始阶段与父进程共享物理页面。

当子进程尝试写入共享页面时，触发 写时复制（Copy on Write），为其分配独立页面。

1. 更新文件描述符：

父进程打开的文件，其引用计数加 1。

1. 更新 GDT 描述符：

在全局描述符表（GDT）中设置新任务的 TSS 和 LDT 描述符。

1. 设置任务状态：

将新任务状态设置为 就绪态，返回新进程的 PID。

1. 写时复制（Copy on Write）技术

在进程创建时，子进程与父进程共享相同的物理内存页面，只有在任一进程尝试写入共享页面时，系统才实际分配独立的物理页面。这种技术有效减少了内存开销。

通过以上设计，实验旨在分析和验证 Linux 0.11 内核中 fork 系统调用的具体实现过程以及相关内核机制。

# 五、实验步骤及实验结果

1. 准备实验

1）启动Engintime Linux Lab。

2）新建一个Linux011 Kernel实验项目。

1. 添加新的系统调用函数

为Linux 0.11添加2个新的系统调用函数，分别是：

1. dump\_physical\_mem()
2. 在终端设备上显示当前内存信息，包括物理页总数、空闲页数量和占用页数量；
3. 调用get\_free\_page函数分配一个空闲的物理页，输出当前空闲页和占用页的数量；
4. 调用free\_page函数将刚刚分配的物理页进行回收，最后输出当前空闲页和占用页的数量。
5. table\_mapping()
6. 该系统调用的内核函数原型：int sys\_table\_mapping()。
7. 该函数首先调用/mm/memory.c中的calc\_mem（）函数将内存中空闲页面的数量以及各个页表（从0号页起）映射的物理页数量输出到文本文件a.txt中，然后依次将下面信息输出到文本文件a.txt中。
8. 步骤如下：

1）首先要为新系统调用分配一个唯一的系统调用号，以原来的最大系统调用号为基础加1作为新的系统调用号。在“项目管理器”窗口中双击打开include/unistd.h文件，在第163-165行添加新的系统调用号，如图2所示：



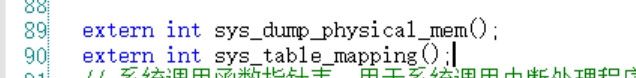
**图3 添加新的系统调用号**

2）添加新系统调用号的同时，也要使系统调用总数在原来的基础上增加1。打开kernel/system\_call.s文件，修改在第73行定义的系统调用总数，如下图：



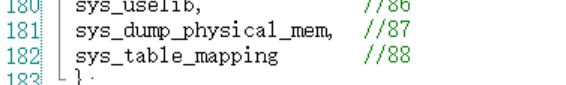
**图4 修改Linux内核的系统调用总数**

3）在include/linux/sys.h文件中的第89-90行使用C语言声明内核函数的原型，如图4所示。注意，这里定义的函数原型并不需要与函数的定义完全一致，只需要定义函数的返回值为int类型，参数为空，能够让标识符代表一个函数名称即可。



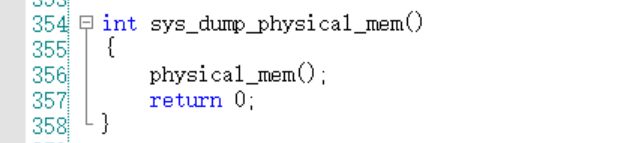
**图5 使用C语言声明内核函数的原型**

在此文件的最后，向系统调用函数指针表sys\_call\_table[]中添加新系统调用函数的指针（注意，系统调用号必须与系统调用内核函数指针在系统调用函数表中的索引一一对应），如图5所示：



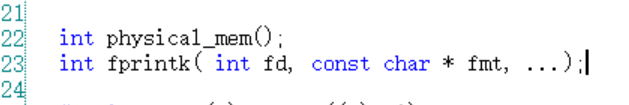
**图6 在系统调用函数指针表中增加内核函数的指针**

4）在kernel/sys.c文件的最后编写代码，实现新系统调用对应的内核函数，如下图所示：



**图8 系统调用内核函数dump\_physical\_mem()**

5）kernel.h文件中22行声明 physical\_mem()函数和fprintk(int fd, const char \* fmt, ...)函数

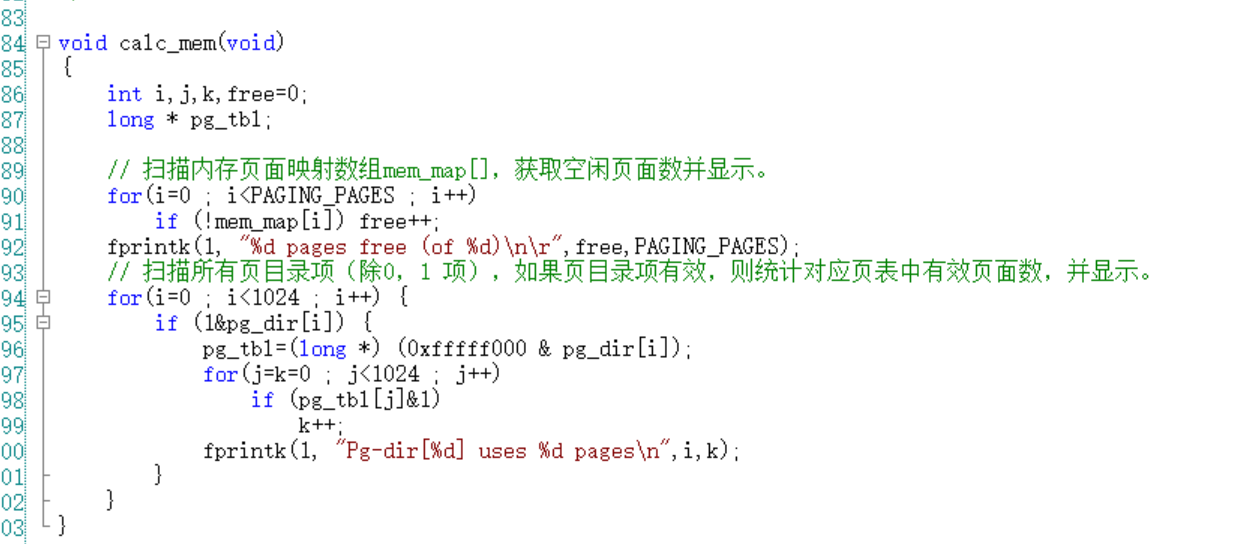


**图9 physical\_mem()函数和fprintk函数声明**

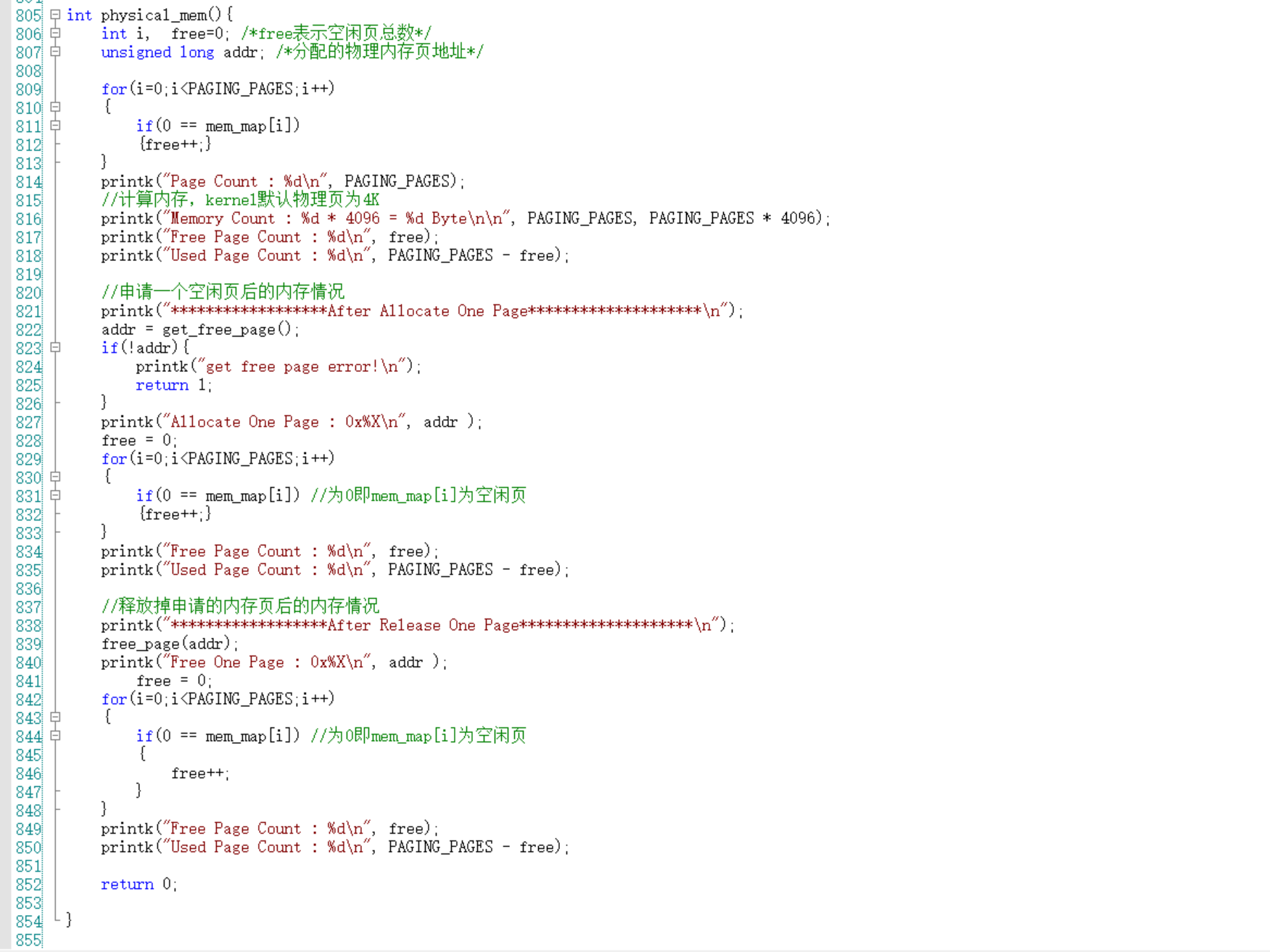


**图9 系统调用内核函数table\_mapping()**

6）在memory.c文件中782行实现 int dump\_physical\_mem()函数、void calc\_mem(void)函数

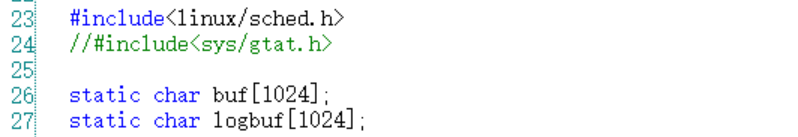


**图10 calc\_mem(void)函数实现**



**图11 dump\_physical\_mem()函数实现**

7）printk.c文件中第23到27行定义fprintk，第55行实现fprintk。



**图12 fprintk的声明**



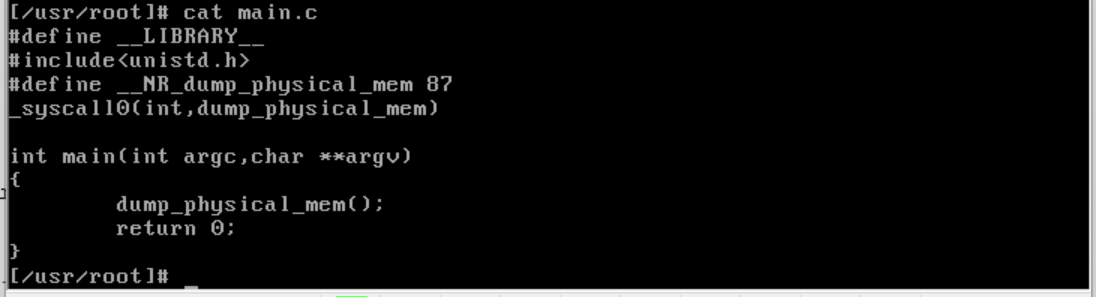
**图13 fprintk的实现**

8）按F7生成Linux 0.11内核，修改语法错误直到生成成功。接下来需要在Linux 0.11中分别编写应用程序来测试新系统调用是否添加成功。

1. 在应用程序中测试新系统调用

1）按F5启动调试。

2）待Linux 0.11启动后，使用vi编辑器新建一个main.c文件。编辑main.c文件中的源代码（如图7所示）。其中，需要定义 \_\_LIBRARY\_\_ 宏及包含unistd.h头文件，还需要再次定义 \_\_NR\_max 宏，并使用 \_syscall0宏对系统调用函数进行定义，当该宏展开时，就会在源代码文件中使用C语言添加dump\_pysical\_mem函数的完整实现。

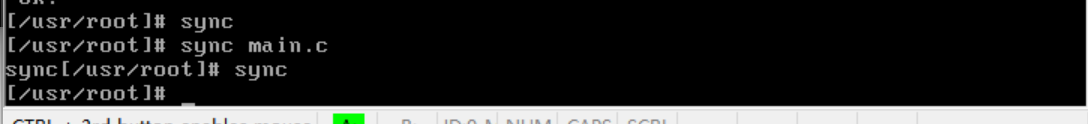


**图14 编辑main.c文件**

3）使用命令gcc main.c -o main生成可执行文件main。

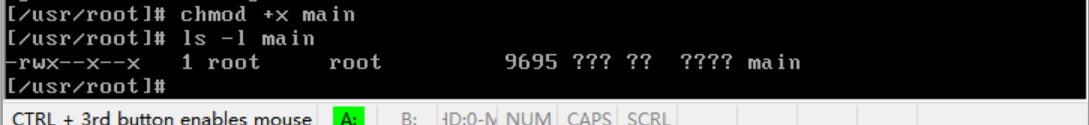
屏幕截图 2024-11-25 215453

4）执行sync命令，将文件保存到磁盘。



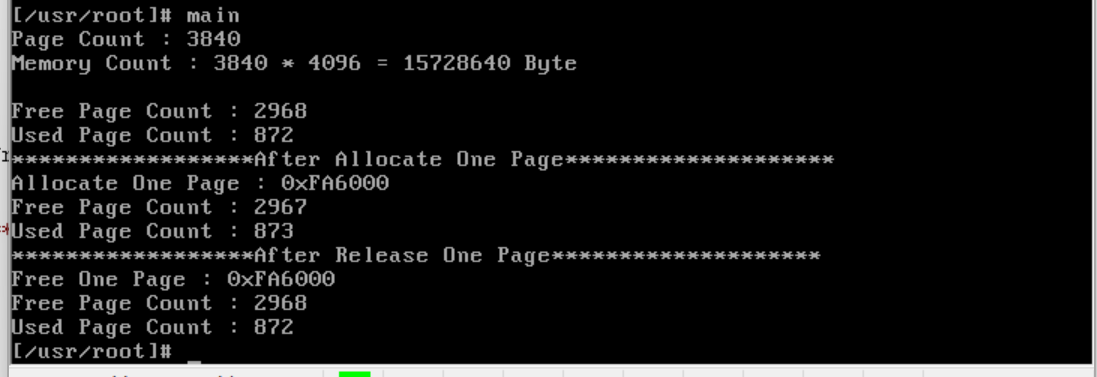
**图15 将文件保存到磁盘**

5）执行 chmod +x main 命令为main文件添加可执行权限。



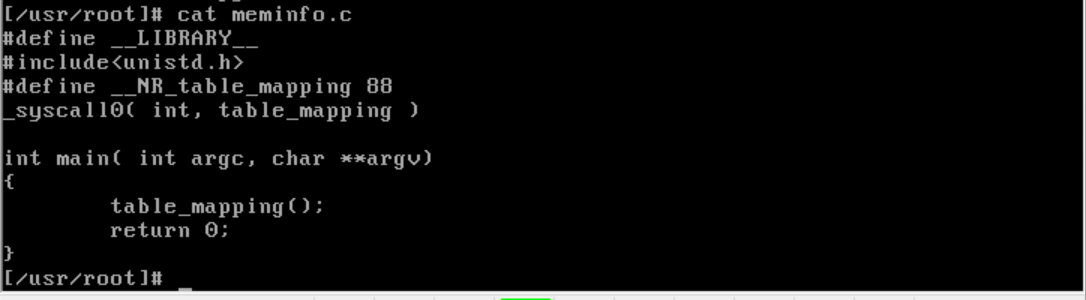
**图16 添加权限**

6）运行main。如果main能输出正确的结果，则说明新系统调用添加成功。



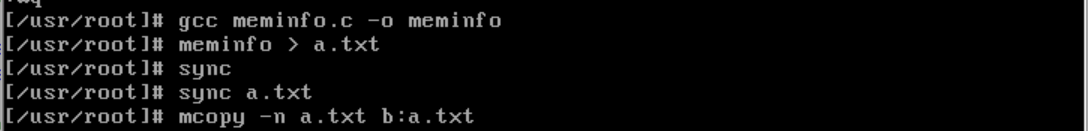
**图17 正确输出分配物理页和释放物理页的程序执行结果**

1. 仿照上述2）-6）步骤，测试新系统调用。



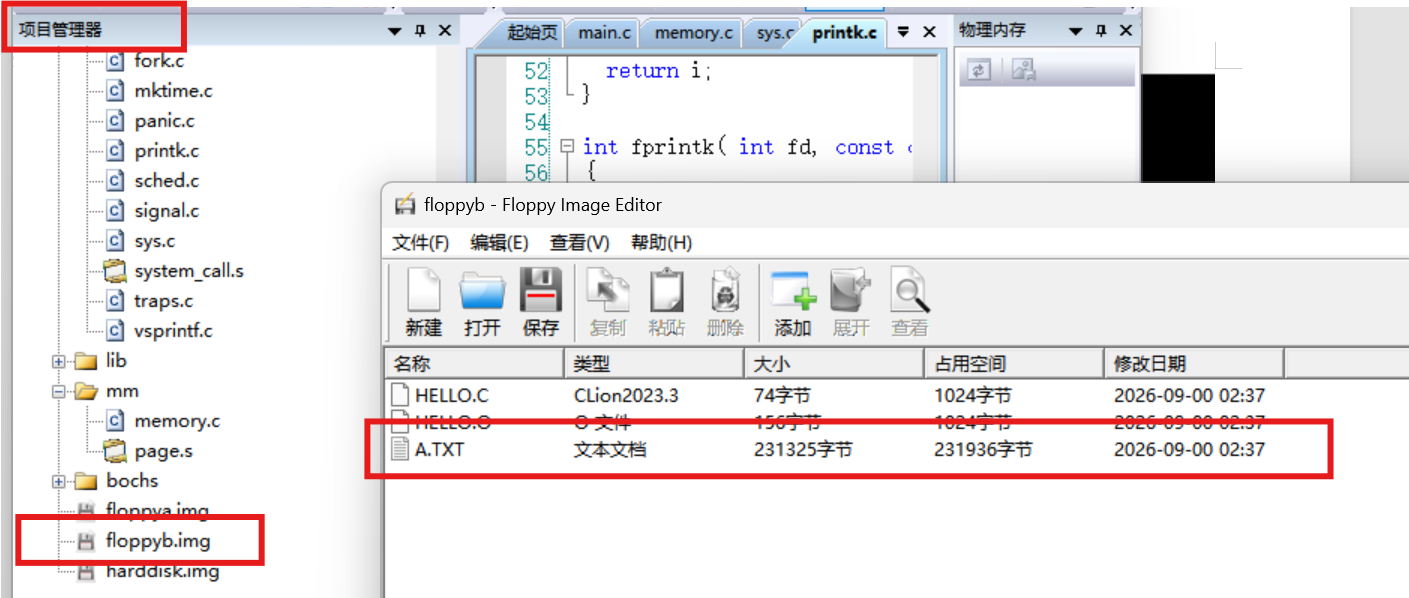
**图18 编辑meminfo.c文件**

1. 依次进行以下操作，将内存中空闲页面的数量以及各个页表（从0号页起）映射的物理页数量输出到文本文件a.txt中，然后依次将下面信息输出到文本文件a.txt中，并将该文件保存到软盘中。



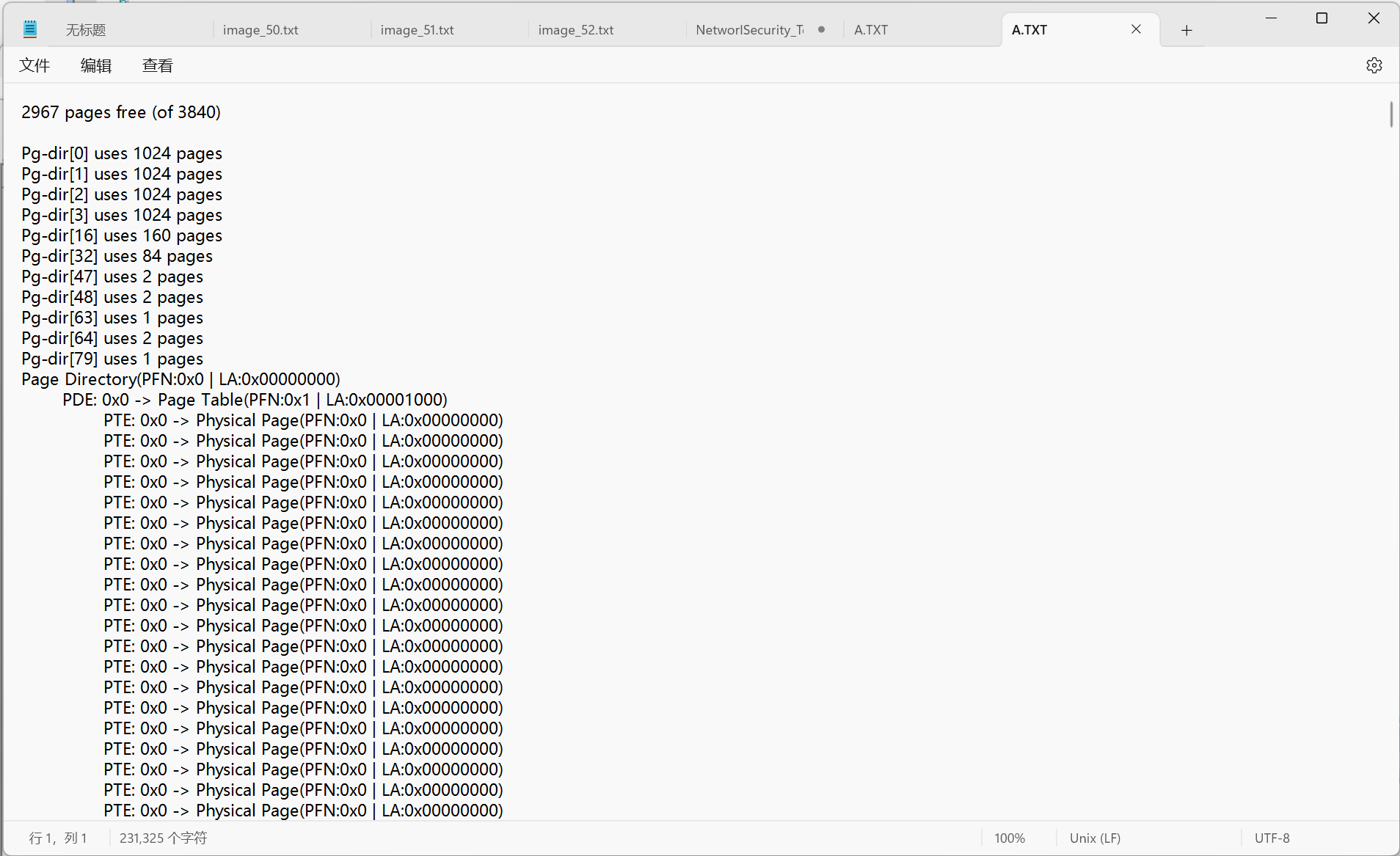
**图19 a.txt文件生成**

1. 在左侧（项目管理器）最下面找到floppyb.img,双击打开



**图19 floppyb.img软盘映像**

1. 查看a.txt文本文件内容。可以选择通过直接双击打开文件开始浏览或者复制到Windows再进行查看。



**图20 文本文件a.txt**

# 六、实验出现的问题及解决方法

问题：floppyb.img文件无法通过双击打开。

解决方法：另外新建一个项目，重新开始。

# 七、结论

通过本次实验，我对 Linux 操作系统核心机制的理解得到了显著提升，尤其是在进程管理与调度方面的实现原理。本实验深入研究了 fork 系统调用的全过程，学习到进程创建不仅涉及当前进程数据的复制，还需要完成对任务描述符、内存空间和调度状态的初始化。特别是写时复制（Copy on Write）技术让我感受到操作系统在资源管理上的高效性，这种技术不仅避免了资源浪费，还提高了系统性能。

同时，通过对 task\_struct 和进程表设计的分析，我认识到结构化数据在内核管理中的重要作用。进程表作为核心数据结构，不仅记录了系统中所有进程的状态，还为操作系统提供了统一的管理接口。任务控制块中的丰富信息，例如进程状态、寄存器值和时间片，为调度算法的实现奠定了坚实的基础。

在实验过程中，我还掌握了内核代码的阅读和分析方法，理解了如何从内核实现中探索操作系统的工作原理。这不仅提升了我的技术能力，也让我体会到操作系统设计的巧妙与复杂。本次实验让我对操作系统底层原理有了更加系统的认识，为后续深入学习提供了重要的理论与实践支持。

# 附录：程序源代码

//printk.c文件

int fprintk( int fd, const char \* fmt, ... )

{

va\_list args;

int i;

struct file \* file;

struct m\_inode \* inode;

va\_start (args, fmt);

i = vsprintf (logbuf, fmt, args);

va\_end (args);

if( fd<3 )

{

\_\_asm\_\_ ("push %%fs\n\t"

"push %%ds\n\t"

"pop %%fs\n\t"

"pushl %0\n\t"

"pushl $\_logbuf\n\t"

"pushl %1\n\t"

"call \_sys\_write\n\t"

"addl $8,%%esp\n\t"

"popl %0\n\t"

"pop %%fs"

::"r" (i),"r" (fd):"ax", "dx");

}

else

{

if( !( file=task[0]->filp[fd] ) )

return 0;

inode=file->f\_inode;

\_\_asm\_\_ ("push %%fs\n\t"

"push %%ds\n\t"

"pop %%fs\n\t"

"pushl %0\n\t"

"pushl $\_logbuf\n\t"

"pushl %1\n\t"

"pushl %2\n\t"

"call \_file\_write\n\t"

"addl $12,%%esp\n\t"

"popl %0\n\t"

"pop %%fs"

::"r" (i), "r" (file), "r" (inode) );

}

return i;// 返回字符串长度

}

//sys.c文件

int sys\_dump\_physical\_mem()

{

physical\_mem();

return 0;

}

int sys\_table\_mapping()

{

unsigned long index\_of\_dir, index\_of\_table;

unsigned long entry;

unsigned long page\_table\_base;

unsigned long page\_dir\_base = 0;

\_\_asm("cli");

calc\_mem(); /\*显示内存空闲页面数\*/

//首先打印输出页目录信息，格式为：PageDirectory(页目录的物理页框号 | 所在线性地址)

fprintk(1, "Page Directory(PFN:0x0 | LA:0x00000000)\n");

/\*第一层循环，遍历页目录中的所有PDE\*/

for(index\_of\_dir = 0; index\_of\_dir < 1024; index\_of\_dir++)

{

entry = ((unsigned long\*)page\_dir\_base)[index\_of\_dir];

if(!(entry & 1)) /\*跳过无效的 PDE\*/

continue;

//输出PDE信息，格式为：

//PDE: 下标 -> Page Table(页表的物理页框号 | 所在的线性地址)

fprintk(1, "\tPDE: 0x0 -> Page Table(PFN:0x1 | LA:0x00001000)\n", index\_of\_dir, entry >> 12, 0xfffff000 & entry);

/\*页目录的高20位即为页表的物理地址，页表的物理地址即为页表的逻辑地址\*/

page\_table\_base = 0xfffff000 & entry;

/\*第二层循环，遍历页表中的所有PTE\*/

for(index\_of\_table = 0; index\_of\_table < 1024; index\_of\_table++)

{

entry = ((unsigned long\*)page\_table\_base)[index\_of\_table];

if(!(entry & 1)) /\*跳过无效的PTE\*/

continue;

//输出PTE信息，格式为：

//PTE: 下标 -> Physical Page(物理页框号 | 所在的线性地址)

fprintk(1, "\t\tPTE: 0x0 -> Physical Page(PFN:0x0 | LA:0x00000000)\n", index\_of\_table, entry >> 12, (index\_of\_dir << 22) | (index\_of\_table << 12));

}

}

\_\_asm("sti");

return 0;

}

//memory.c文件

void calc\_mem(void)

{

int i,j,k,free=0;

long \* pg\_tbl;

// 扫描内存页面映射数组mem\_map[]，获取空闲页面数并显示。

for(i=0 ; i<PAGING\_PAGES ; i++)

if (!mem\_map[i]) free++;

fprintk(1, "%d pages free (of %d)\n\r",free,PAGING\_PAGES);

// 扫描所有页目录项（除0，1 项），如果页目录项有效，则统计对应页表中有效页面数，并显示。

for(i=0 ; i<1024 ; i++) {

if (1&pg\_dir[i]) {

pg\_tbl=(long \*) (0xfffff000 & pg\_dir[i]);

for(j=k=0 ; j<1024 ; j++)

if (pg\_tbl[j]&1)

k++;

fprintk(1, "Pg-dir[%d] uses %d pages\n",i,k);

}

}

}

int physical\_mem(){

int i, free=0; /\*free表示空闲页总数\*/

unsigned long addr; /\*分配的物理内存页地址\*/

for(i=0;i<PAGING\_PAGES;i++)

{

if(0 == mem\_map[i])

{free++;}

}

printk("Page Count : %d\n", PAGING\_PAGES);

//计算内存，kernel默认物理页为4K

printk("Memory Count : %d \* 4096 = %d Byte\n\n", PAGING\_PAGES, PAGING\_PAGES \* 4096);

printk("Free Page Count : %d\n", free);

printk("Used Page Count : %d\n", PAGING\_PAGES - free);

//申请一个空闲页后的内存情况

printk("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*After Allocate One Page\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

addr = get\_free\_page();

if(!addr){

printk("get free page error!\n");

return 1;

}

printk("Allocate One Page : 0x%X\n", addr );

free = 0;

for(i=0;i<PAGING\_PAGES;i++)

{

if(0 == mem\_map[i]) //为0即mem\_map[i]为空闲页

{free++;}

}

printk("Free Page Count : %d\n", free);

printk("Used Page Count : %d\n", PAGING\_PAGES - free);

//释放掉申请的内存页后的内存情况

printk("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*After Release One Page\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

free\_page(addr);

printk("Free One Page : 0x%X\n", addr );

free = 0;

for(i=0;i<PAGING\_PAGES;i++)

{

if(0 == mem\_map[i]) //为0即mem\_map[i]为空闲页

{

free++;

}

}

printk("Free Page Count : %d\n", free);

printk("Used Page Count : %d\n", PAGING\_PAGES - free);

return 0;

}