

本 科 生 课 程 设 计 报 告

题 目：Linux早期内核中的内存管理

院 系 软件学院

专业班级 数字媒体技术1601班

学 号 U201617167

姓 名 杨逸辰

2018 年 06 月 11 日

# 一、主要任务

分析Linux内存管理的原理，有无采用虚拟内存管理，并对具体的实现源代码、数据结构和流程进行分析。分析Linux物理内存的管理，并对具体的实现源代码、数据结构和流程进行分析。

# 二、研究对象

Linux0.11版本中内存管理对应的源代码，主要位于./linux0.11/mm文件夹中。该目录下有三个文件：

·Makefile

·memory.c

·page.s

page.s由汇编语言编写，包含内存页异常的中断处理过程，主要实现对缺页和页写保护的处理。

Memory.c由C语言编写，是内存页面管理的核心文件，用于内存的初始化操作、页目录和页表的管理、内核其他部分对内存的申请处理过程。

# 三、研究内容

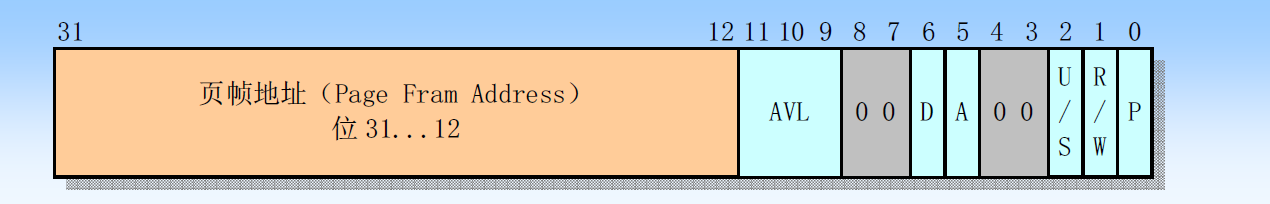
1.Linux内存管理的原理

1.1Linux内存分页管理机制

Linux内存分页管理是通过页目录表和内存页表所组成的二级表进行的。

其中页目录表和页表的结构是一样的，表项结构也相同。页目录表中的每个表项为4字节，即页目录项，用来寻址一个页表。每个页表项，同样是4字节，用来寻址一页物理内存页。因此，当指定了一个页目录项和一个页表项，我们就可以唯一地确定所对应的物理内存页。页目录表占用一页内存，因此最多可以寻址1024个页表。而每个页表也同样占用一页内存，因此一个页表可以寻址最多1024个物理内存页面。Linux0.11中唯一的一个页目录表可以寻址4G空间。但页表不止一个，每个进程都有自己的页表。

它们的表项结构如下图：



其中：

*·页框地址(PAGE FRAME ADDRESS)*指定了一页内存的物理起始地址。因为内存页是位于4K边界上的，所以其低12比特总是0，因此表项的低12比特可作它用。在一个页目录表中，表项的页框地址是一个页表的起始地址；在第二级页表中，页表项的页框地址则包含期望内存操作的物理内存页地址。

·*存在位（P）*确定了一个页表项是否可以用于地址转换过程。P=1 表示该项可用。当目录表项或第二级表项的P=0 时，则该表项时无效的，不能用于地址转换过程。此时该表项的所有其它比特位都可供程序使用；处理器不对这些位进行测试。当 CPU 试图使用一个页表项进行地址转换时，如果此时任意一级页表项的P=0，则处理器就会发出页异常信号。此时缺页中断异常处理程序就可以把所请求的页加入到物理内存中，并且导致异常的指令会被重新执行。

·*已访问（A）和已修改（D）比特位*用于提供有关页使用的信息。除了页目录项中的已修改位，这些比特位将由硬件置位，但不复位。在对一页内存进行读或写操作之前，CPU 将设置相关的目录和二级页表项的已访问位。在向一个二级页表项所涵盖的地址进行写操作之前，处理器将设置该二级页表项的已修改位，而页目录项中的已修改位是不用的。当所需求的内存超出实际物理内存量时，内存管理程序就可以使用这些位来确定那些页可以从内存中取走，以腾出空间。内存管理程序还需负责检测和复位这些比特位。

·*读/写位（R/W）和用户/超级用户位（U/S）*并不用于地址转换，但用于分页级的保护机制，是由CPU 在地址转换过程中同时操作的。

对于内核而言，代码段和数据段长度是16MB，使用了4个页表。这4个页表紧随页目录表之后，见head.s程序相关部分代码：

109 /\*

110 \* I put the kernel page tables right after the page directory,

111 \* using 4 of them to span 16 Mb of physical memory. People with

112 \* more than 16MB will have to expand this.

113 \*/

114 .org 0x1000

115 pg0:

116

117 .org 0x2000

118 pg1:

119

120 .org 0x3000

121 pg2:

122

123 .org 0x4000

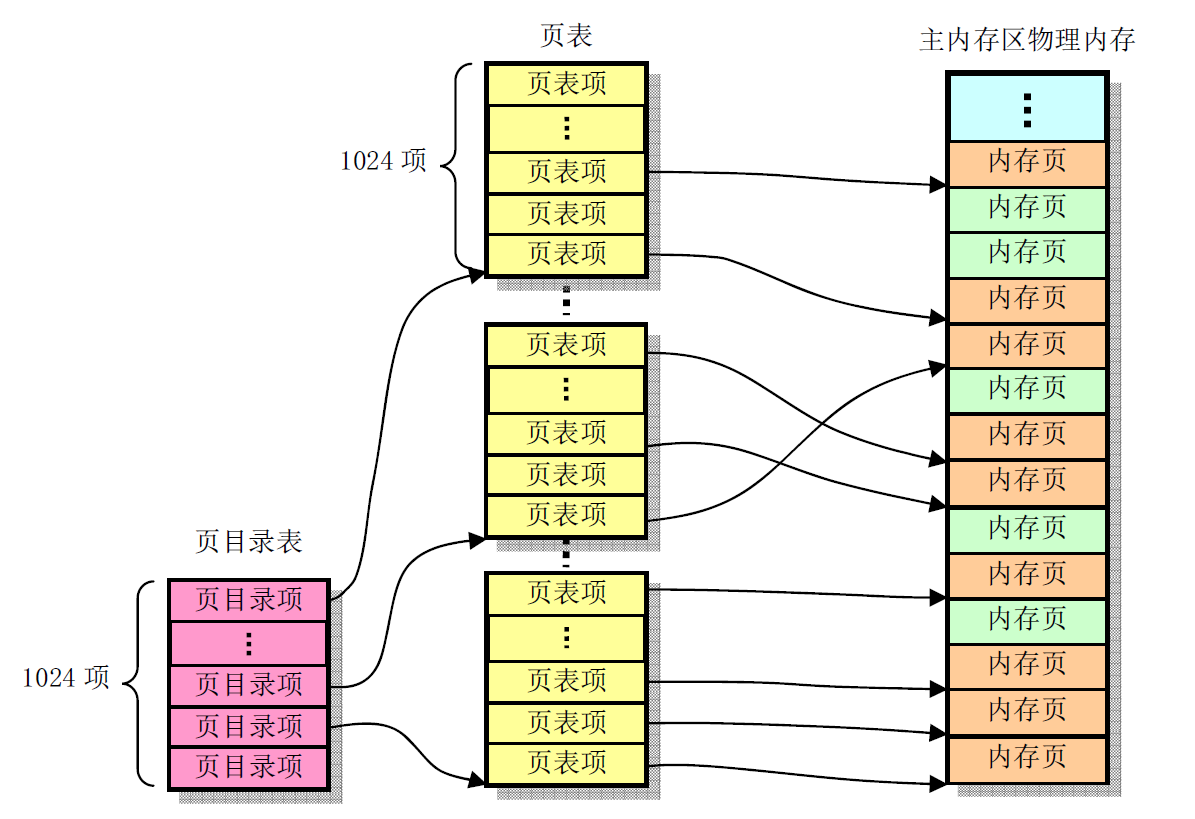
124 pg3:

125

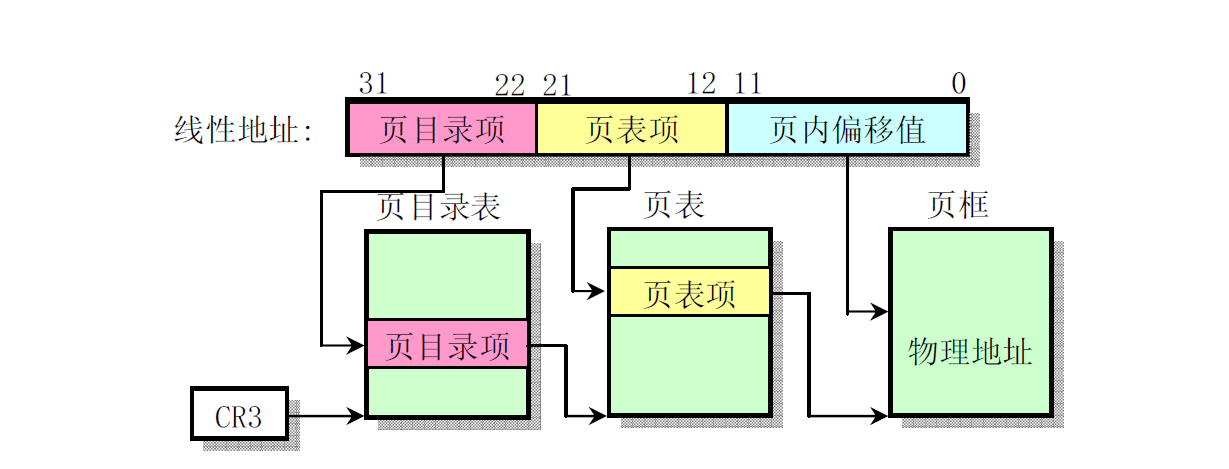
126 .org 0x5000 #

对于应用进程来讲，在申请内存时使用的是32位的线性地址。线性地址分为三个部分，分别用来指定一个页目录项、一个页表项和对应物理内存页上的偏移地址，从而能间接地寻址到线性地址指定的物理内存位置。

页目录表和页表结构如下图：



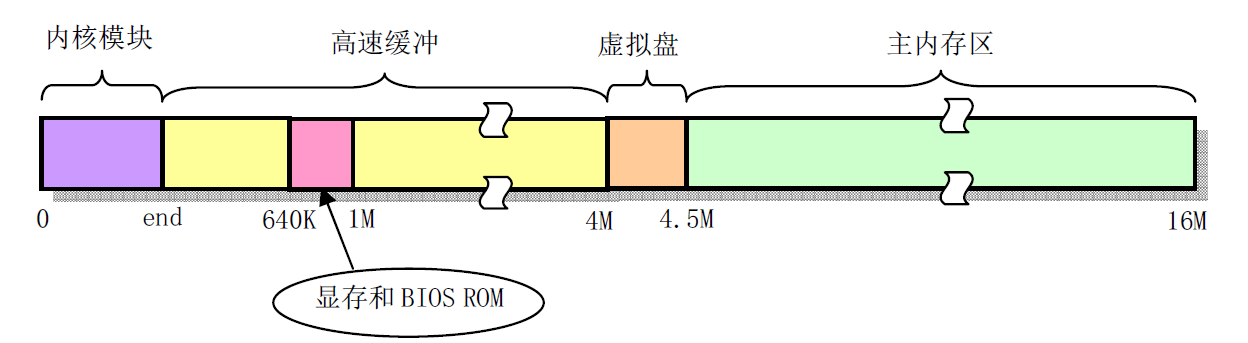
线性地址转化如下图：



在内存管理的函数中，大量使用了从线性地址到实际物理地址的变换计算。对于给定一个进程的线性地址，可以很容易地找到该线性地址对应的页目录项。若该目录项有效，则该目录项中的页框地址指定了一个页表在物理内存中的基址，那么结合线性地址中的页表项指针，若该页表项有效，则根据该页表项中的指定的页框地址，我们就可以最终确定指定线性地址对应的实际物理内存页的地址。再根据页内偏移值可以最终将线性地址转化为物理地址。

1.2 Linux中物理内存的管理和分配

首先说明Linux0.11物理内存空间的情况，如下图所示：



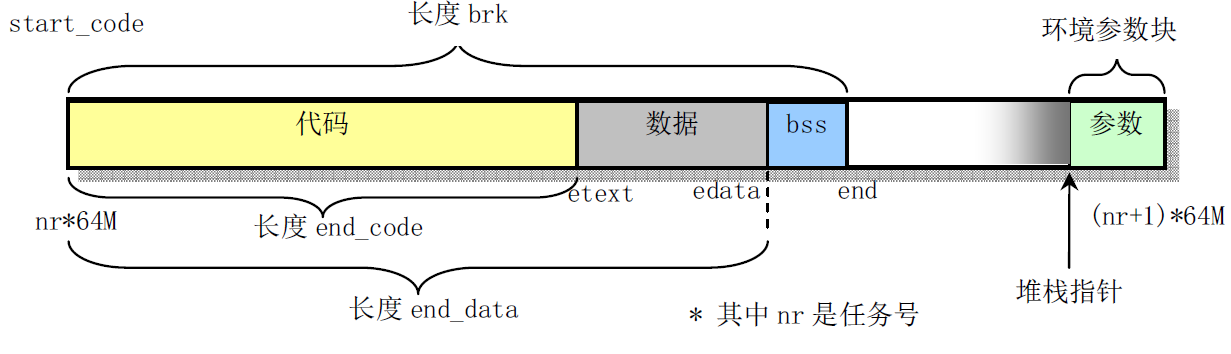
对于linux 0.11内核，它默认最多支持16M 物理内存。在一个具有16MB 内存的80x86计算机系统中，linux 内核占用物理内存最前段的一部分，图中end标示出内核模块结束的位置。随后是高速缓冲区，它的最高内存地址为4M 。高速缓冲区被显示内存和ROM、BIOS 分成两段。剩余的内存部分称为主内存区。主内存区就是由本章的程序进行分配管理的。若系统中还存在RAM 虚拟盘时，则主内存区前段还要扣除虚拟盘所占的内存空间。当需要使用主内存区时就需要向本章的内存管理程序申请，所申请的基本单位是内存页。

为了节约物理内存，在调用fork()生成新进程时，新进程与原进程会共享同一内存区。只有当其中一个进程进行写操作时，系统才会为其另外分配内存页面。这就是*写时复制*的概念。

页异常中断处理过程（int 14）通过page.s程序实现。该中断处理过程对由于缺页和页写保护引起的中断分别调用memory.c 中的do\_no\_page()和do\_wp\_page()函数进行处理。do\_no\_page()会把需要的页面从块设备中取到内存指定位置处。在共享内存页面情况下，do\_wp\_page()会复制被写的页面（copy on write，写时复制），从而也取消了对页面的共享。

1.3 Linux内核对线性空间的使用

进程在线性地址空间中的分布如下图所示：



每个进程在线性地址中都是从（任务号\*64M）的地址位置开始，也就是说，linux0.11给每个进程分配的线性地址空间为64M。

2.结合代码分析

2.1 页目录和内核页表的建立

Linux0.11采用二级页表，对任何一个物理地址的访问都要通过页目录表和页表的映射机制来间接访问。程序提供的任何地址信息都会被当成线性地址进行映射，而操作系统必须要了解物理内存的使用情况才能够进行管理。

要能够在保护模式下感知物理内存，也就是说要能够避开保护模式下线性地址的影响，直接对物理内存进行操作，只有让分页机制对内核失效，让内核使用的线性地址和物理地址重合。

在进入保护模式之前，要初始化页目录表和页表，以供在切换到保护模式之后使用，要实现内核线性地址和物理地址的重合，必须要在这个时候处理好页目录表和页表。

在 linux/boot/head.s 中首先对 5 页（页目录表占一页大小，以及紧随其后的四页页表）内存清零：

198 setup\_paging:

199 movl $1024\*5,%ecx /\* 5 pages - pg\_dir+4 page tables \*/

#设置填充次数ecx=1024\*5

200 xorl %eax,%eax #设置填充到内存单元中的数eax=0

201 xorl %edi,%edi /\* pg\_dir is at 0x000 \*/

#设置填充的起始地址0，也是页目录表的起始位置

202 cld;rep;stosl

清零后进行填写目录项和表项：

203 movl $pg0+7,\_pg\_dir /\* set present bit/user r/w \*/

204 movl $pg1+7,\_pg\_dir+4 /\* --------- " " --------- \*/

205 movl $pg2+7,\_pg\_dir+8 /\* --------- " " --------- \*/

206 movl $pg3+7,\_pg\_dir+12 /\* --------- " " --------- \*/

接着便是对页表的设置：

207 movl $pg3+4092,%edi # edi 指向第四个页表的最后一项4096-4。

208 movl $0xfff007,%eax /\* 16Mb - 4096 + 7 (r/w user,p) \*/

#把第四个页表的最后一项的内容放进eax

209 std # 置方向位，edi 值以4 字节的速度递减。

210 1: stosl /\* fill pages backwards - more efficient :-) \*/

211 subl $0x1000,%eax # 每填写好一项，物理地址值减0x1000。

212 jge 1b # 如果eax 小于0 则说明全填写好了。

# 使页目录表基址寄存器cr3 指向页目录表。

213 xorl %eax,%eax /\* pg\_dir is at 0x0000 \*/

令eax=0x0000 0000(页目录表基址)

214 movl %eax,%cr3 /\* cr3 - page directory start \*/

# 设置cr0 的PG 标志（位31），启动保护模式

215 movl %cr0,%eax

216 orl $0x80000000,%eax # 添上PG 标志位。

217 movl %eax,%cr0 /\* set paging (PG) bit \*/

当这段代码运行完后内存中的映射关系应该如图所示：



接下来将内核代码段描述符gdt 设置为

0x00c09a0000000fff /\* 16Mb \*/ # 代码段最大长度16M。

这样内核中线性地址就和物理地址重合了，操作系统便可对虚拟内存空间进行管理。

2.2初始化

根据第2部分的分析可知，linux对内存分块，在linux0.11 中内存管理主要是对主内存块的管理。

在linux0.11定义了一个字符数组mem\_map [ PAGING\_PAGES ]用于对主内存区的页面分配和共享信息进行记录。

相关宏定义在/mm/memory.c 中：

43 #define LOW\_MEM 0x100000 // 主内存块可能的最低端（1MB）。

44 #define PAGING\_MEMORY (15\*1024\*1024) // 主内存区最多可以占用15M。

45 #define PAGING\_PAGES (PAGING\_MEMORY>>12) // 主内存块最多可以占用的物理页面数

46 #define MAP\_NR(addr) (((addr)-LOW\_MEM)>>12) // 将指定物理内存地址映射为映射数组标号。

47 #define USED 100 // 页面被占用标志

57 static unsigned char mem\_map [ PAGING\_PAGES ] = {0,}; //主内存块映射数组

数组mem\_map[]中每一项内容表示物理内存被多少个对象使用，对应项为0就表示对应物理内存的页面空闲。

对于不是属于实际主内存块的物理内存的对应项清除掉，linux0.11 采用的做法是在初始化时将属于实际属于主内存块的物理内存的对应项的值清零，将不属于的置为一个相对较大的值USED。这样在管理时这些不属于主内存块的页面就不会通过主内存块的管理程序被分配出去使用了。

主内存块初始化的代码位于linux0.11/init/main.c中。

下面是系统初始化函数main() 中的内容：

112 memory\_end = (1<<20) + (EXT\_MEM\_K<<10); // 内存大小=1Mb 字节+ 扩展内存(k)\*1024 字节。

113 memory\_end &= 0xfffff000; // 以页面为单位取整。

114 if (memory\_end > 16\*1024\*1024) // linux0.11最大支持 16M物理内存

115 memory\_end = 16\*1024\*1024;

116 if (memory\_end > 12\*1024\*1024) //根据内存大小设置缓冲区末端的位置

117 buffer\_memory\_end = 4\*1024\*1024;

118 else if (memory\_end > 6\*1024\*1024)

119 buffer\_memory\_end = 2\*1024\*1024;

120 else

121 buffer\_memory\_end = 1\*1024\*1024;

122 main\_memory\_start = buffer\_memory\_end; // 主内存起始位置=缓冲区末端；

123 #ifdef RAMDISK // 如果定义了虚拟盘，重新设置主内存块起始位置

//rs\_init()返回虚拟盘的大小

124 main\_memory\_start += rd\_init(main\_memory\_start, RAMDISK\*1024);

125 #endif

126 mem\_init(main\_memory\_start,memory\_end); //初始化主内存块

下面就是mem\_init（位于linux0.11/mm/memory.c）的代码：

399 void mem\_init(long start\_mem, long end\_mem)

400 {

401 int i;

402

403 HIGH\_MEMORY = end\_mem; // 设置物理内存最高端。

404 for (i=0 ; i<PAGING\_PAGES ; i++) // 将主内存块映射数组所有项置为USED

405 mem\_map[i] = USED;

406 i = MAP\_NR(start\_mem); // 计算实际主内存块物理地址起始位置对应的映射项

407 end\_mem - = start\_mem; //计算实际主内存块大小

408 end\_mem >>= 12; //计算需要初始化的映射项数目

409 while (end\_mem-->0) 将实际 // 主内存块对应的映射项置为0（空闲）

410 mem\_map[i++]=0;

411 }

通过以上的操作之后，操作系统便可以了解主内存块中物理内存页面的使用情况了，从而进行管理。

2.3 内存的分配与回收

当内核或用户进程需要新的物理页面时，通过内核对主内存块中空闲物理内存页面的映射结构mem\_map，找出其中一个空闲项，并将该项映射为对应的物理页面地址即可。实现函数为linux0.11/mm/memory.c的get\_free\_page()。

get\_free\_page()函数用于在主内存区中申请一页空闲内存页，并返回物理内存页的起始地址。它首先扫描内存页面字节图数组mem\_map[]，寻找值是0 的字节项（对应空闲页面）。若无则返回0 结束，表示物理内存已使用完。若找到值为0的字节，则将其置1，并换算出对应空闲页面的起始地址。然后对该内存页面作清零操作。最后返回该空闲页面的物理内存起始地址。

63 unsigned long get\_free\_page(void)

64 {

65 register unsigned long \_\_res asm("ax");

66

67 \_\_asm\_\_("std ; repne ; scasb\n\t" // 方向位置位，将al(0)与对应每个页面的(di)内容比较

68 "jne 1f\n\t" // 如果没有等于0的字节，则跳转结束（返回0）。

69 "movb $1,1(%%edi)\n\t // 将对应页面的内存映像位置1。

70 "sall $12,%%ecx\n\t" // 页面数\*4K = 相对页面起始地址。

71 "addl %2,%%ecx\n\t" // 再加上低端内存地址，即获得页面实际物理起始地址。

72 "movl %%ecx,%%edx\n\t" // 将页面实际起始地址edx寄存器。

73 "movl $1024,%%ecx\n\t" // 寄存器ecx 置计数值1024。

74 "leal 4092(%%edx),%%edi\n\t" // 将4092+edx 的位置􀃎edi(该页面的末端)。

75 "rep ; stosl\n\t" // 将edi 所指内存清零（反方向，也即将该页面清零）。

76 "movl %%edx,%%eax\n" // 将页面起始地址􀃎eax（返回值）。

77 "1:"

78 :"=a" (\_\_res)

79 :"" (0),"i" (LOW\_MEM),"c" (PAGING\_PAGES),

80 "D" (mem\_map+PAGING\_PAGES-1)

81 :"di","cx","dx");

82 return \_\_res; // 返回空闲页面地址（如果无空闲页则返回0）。

83 }

84

当内核使用完一个物理页面或者进程退出时内核归还申请了的物理页面。这时就需要更改相应的信息，以便下一次使用。回收的实现源代码在linux0.11/mm/memeroy.c中，通过free\_page()函数实现。

free\_page()用于释放指定地址处的一页物理内存。它首先判断指定的内存地址是否<1M，若是则返回，因为1M 以内是内核专用的；若指定的物理内存地址大于或等于实际内存最高端地址，则显示出错信息；然后由指定的内存地址换算出页面号: (addr - 1M)/4K；接着判断页面号对应的mem\_map[]字节项是否为0，若不为0，则减1 返回；否则对该字节项清零，并显示“试图释放一空闲页面”的出错信息。

89 void free\_page(unsigned long addr)

90 {

91 if (addr < LOW\_MEM) return; // 如果物理地址addr 小于内存低端（1MB），则返回。

92 if (addr >= HIGH\_MEMORY) // 如果物理地址addr>=内存最高端，则显示出错信息。

93 panic("trying to free nonexistent page");

94 addr -= LOW\_MEM; // 物理地址减去低端内存位置，再除以4KB，得页面号。

95 addr >>= 12;

96 if (mem\_map[addr]--) return; // 如果对应内存页面映射字节不等于0，则减1 返回。

97 mem\_map[addr]=0; // 否则置对应页面映射字节为0，并显示出错信息，死机。

98 panic("trying to free free page");

99 }

2.4 物理地址、线性地址之间的映射

操作系统在控制物理内存的同时又要控制进程的虚拟空间，这就需要在线性地址和物理地址之间做转换工作。将物理地址映射到线性地址上，需要修改页目录表和页表的相关表项，这一转换通过linux0.11/mm/memory中的put\_page()函数实现。

put\_page()用于将一指定的物理内存页面映射到指定的线性地址处。它首先判断指定的内存页面地址的有效性，要在1M 和系统最高端内存地址之外，否则发出警告；然后计算该指定线性地址在页目录表中对应的目录项；此时若该目录项有效（P=1），则取其对应页表的地址；否则申请空闲页给页表使用，并设置该页表中对应页表项的属性。最后仍返回指定的物理内存页面地址。

197 unsigned long put\_page(unsigned long page,unsigned long address)

198 {

199 unsigned long tmp, \*page\_table;

200

201 /\* NOTE !!! This uses the fact that \_pg\_dir=0 \*/

/\* 注意!!!这里使用了页目录基址\_pg\_dir=0 的条件 \*/

202

// 如果申请的页面位置低于LOW\_MEM(1Mb)或超出系统实际含有内存高端HIGH\_MEMORY，则发出警告。

203 if (page < LOW\_MEM || page >= HIGH\_MEMORY)

204 printk("Trying to put page %p at %p\n",page,address);

// 如果申请的页面在内存页面映射字节图中没有置位，则显示警告信息。

205 if (mem\_map[(page-LOW\_MEM)>>12] != 1)

206 printk("mem\_map disagrees with %p at %p\n",page,address);

// 计算指定地址在页目录表中对应的目录项指针。

207 page\_table = (unsigned long \*) ((address>>20) & 0xffc);

// 如果该目录项有效(P=1)(也即指定的页表在内存中)，则从中取得指定页表的地址􀃎page\_table。

208 if ((\*page\_table)&1)

209 page\_table = (unsigned long \*) (0xfffff000 & \*page\_table);

210 else {

// 否则，申请空闲页面给页表使用，并在对应目录项中置相应标志7（User, U/S, R/W）。然后将

// 该页表的地址􀃎page\_table。

211 if (!(tmp=get\_free\_page()))

212 return 0;

213 \*page\_table = tmp|7;

214 page\_table = (unsigned long \*) tmp;

215 }

// 在页表中设置指定地址的物理内存页面的页表项内容。每个页表共可有1024 项(0x3ff)。

216 page\_table[(address>>12) & 0x3ff] = page | 7;

217 /\* no need for invalidate \*/

/\* 不需要刷新页变换高速缓冲 \*/

218 return page; // 返回页面地址。

219 }

220

//// 取消写保护页面函数。用于页异常中断过程中写保护异常的处理（写时复制）。

// 输入参数为页表项指针。

// [ un\_wp\_page 意思是取消页面的写保护：Un-Write Protected。]

221 void un\_wp\_page(unsigned long \* table\_entry)

222 {

223 unsigned long old\_page,new\_page;

224

225 old\_page = 0xfffff000 & \*table\_entry; // 取原页面对应的目录项号。

// 如果原页面地址大于内存低端LOW\_MEM(1Mb)，并且其在页面映射字节图数组中值为1（表示仅

// 被引用1 次，页面没有被共享），则在该页面的页表项中置R/W 标志（可写），并刷新页变换

// 高速缓冲，然后返回。

226 if (old\_page >= LOW\_MEM && mem\_map[MAP\_NR(old\_page)]==1) {

227 \*table\_entry |= 2;

228 invalidate();

229 return;

230 }

// 否则，在主内存区内申请一页空闲页面。

231 if (!(new\_page=get\_free\_page()))

232 oom(); // Out of Memory。内存不够处理。

// 如果原页面大于内存低端（则意味着mem\_map[]>1，页面是共享的），则将原页面的页面映射

// 数组值递减1。然后将指定页表项内容更新为新页面的地址，并置可读写等标志(U/S, R/W, P)。

// 刷新页变换高速缓冲。最后将原页面内容复制到新页面。

233 if (old\_page >= LOW\_MEM)

234 mem\_map[MAP\_NR(old\_page)]--;

235 \*table\_entry = new\_page | 7;

236 invalidate();

237 copy\_page(old\_page,new\_page);

238 }

2.5 对页表的释放

操作系统为每一个进程建立一个页表，当进程退出时，也必然需要回收它们所占用的内存空间。对整个页表的释放在linux0.11/mm/memory中的free\_page\_tables()函数实现。

free\_page\_tables()用于释放指定线性地址和长度（页表个数）对应的物理内存页。它首先判断指定的线性地址是否在4M 的边界上，若不是则显示出错信息，并死机；然后判断指定的地址值是否=0，若是，则显示出错信息“试图释放内核和缓冲区所占用的空间”，并死机；接着计算在页目录表中所占用的目录项数size，也即页表个数，并计算对应的起始目录项号；然后从对应起始目录项开始，释放所占用的所有size个目录项；同时释放对应目录项所指的页表中的所有页表项和相应的物理内存页；最后刷新页变换高速缓冲。

105 int free\_page\_tables(unsigned long from,unsigned long size)

106 {

107 unsigned long \*pg\_table;

108 unsigned long \* dir, nr;

109

110 if (from & 0x3fffff) // 要释放内存块的地址需以4M 为边界。

111 panic("free\_page\_tables called with wrong alignment");

112 if (!from) // 出错，试图释放内核和缓冲所占空间。

113 panic("Trying to free up swapper memory space");

// 计算所占页目录项数(4M 的进位整数倍)，也即所占页表数。

114 size = (size + 0x3fffff) >> 22;

// 下面一句计算起始目录项。对应的目录项号=from>>22，因每项占4 字节，并且由于页目录是从物理地址0 开始，因此实际的目录项指针=目录项号<<2，也即(from>>20)。与上0xffc 确保目录项指针范围有效。

115 dir = (unsigned long \*) ((from>>20) & 0xffc); /\* \_pg\_dir = 0 \*/

116 for ( ; size-->0 ; dir++) { // size 现在是需要被释放内存的目录项数。

117 if (!(1 & \*dir)) // 如果该目录项无效(P 位=0)，则继续。

118 continue; // 目录项的位0(P 位)表示对应页表是否存在。

119 pg\_table = (unsigned long \*) (0xfffff000 & \*dir); // 取目录项中页表地址。

120 for (nr=0 ; nr<1024 ; nr++) { // 每个页表有1024 个页项。

121 if (1 & \*pg\_table) // 若该页表项有效(P 位=1)，则释放对应内存页。

122 free\_page(0xfffff000 & \*pg\_table);

123 \*pg\_table = 0; // 该页表项内容清零。

124 pg\_table++; // 指向页表中下一项。

125 }

126 free\_page(0xfffff000 & \*dir); // 释放该页表所占内存页面。但由于页表在

// 物理地址1M 以内，所以这句什么都不做。

127 \*dir = 0; // 对相应页表的目录项清零。

128 }

129 invalidate(); // 刷新页变换高速缓冲。

130 return 0;

131 }

2.6 内存共享

由第2部分的原理分析可知，Linux0.11采用写时复制机制，因此在写操作之前，需要进行内存共享。

首先通过函数write\_verify()检查页面是否可写，如果不可写复制页面。根据给定的线性地址得到页表项，根据页表项判断对应的物理页是否可写，如果不可写，则调用un\_wp\_page（），完成新页面的复制。这在linux0.11/mm/memory中的copy\_page\_tables()函数实现。

copy\_page\_tables()用于复制指定线性地址和长度（页表个数）内存对应的页目录项和页表，从而被复制的页目录和页表对应的原物理内存区被共享使用。该函数首先验证指定的源线性地址和目的线性地址是否都在4Mb 的内存边界地址上，否则就显示出错信息，并死机；然后由指定线性地址换算出对应的起始页目录项（from\_dir, to\_dir）；并计算需复制的内存区占用的页表数（即页目录项数）；接着开始分别将原目录项和页表项复制到新的空闲目录项和页表项中。页目录表只有一个，而新进程的页表需要申请空闲内存页面来存放；此后再将原始和新的页目录和页表项都设置成只读的页面。当有写操作时就利用页异常中断调用，执行写时复制操作。最后对共享物理内存页对应的字节图数组mem\_map[]的标志进行增1操作。

2.7 页面异常

在内存管理过程中，由于Linux0.11采用需求加载机制，操作系统并非一次性将所有内存分配给进程，而是在进程要访问该地址时分配，因此很有可能出现缺页中断。当进程试图对一个只读页面进行写操作时也会产生页面异常。

内核通过linux0.11/mm/page.s对异常产生的原因进行判断。首先当产生异常中断时，保护现场，根据出错码判断出错的原因，如果是缺页中断，则调用linux0.11/mm/memory.c的do\_no\_page(error\_code,address)函数作缺页处理。如果是写保护出错，则调用linux0.11/mm/memory.c的do\_wp\_page(error\_code,address)作写保护出错处理。

2.8 内存使用情况

Linux0.11/mm/memory中的calc\_mem（）函数实现。

// 计算内存空闲页面数并显示。

413 void calc\_mem(void)

414 {

415 int i,j,k,free=0;

416 long \* pg\_tbl;

417

// 扫描内存页面映射数组mem\_map[]，获取空闲页面数并显示。

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

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

420 printk("%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。

421 for(i=2 ; i<1024 ; i++) {

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

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

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

425 if (pg\_tbl[j]&1)

426 k++;

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

428 }

429 }

430 }

431

# 三、参考资料

[1] 赵炯. Linux0.11内核完全注释. 北京：机械工业出版社，2007.

[2] 赵炯. Linux内核完全注释. 北京：机械工业出版社，2004.

[3] 袁镱. Linux0.11下的内存管理. 百度文库.

[4] Linus. Linux-0.11内核源代码. <https://www.kernel.org>

[5] 操作系统原理上课PPT第七章《内存管理》