




第13章 内存管理(mm)

在 Intel 80x86 体系结构中，Linux 内核的内存管理程序采用了分页管理方式。利用页目录和页表结构处理内核中其他部分代码对内存的申请和释放操作。内存的管理是以内存页面为单位进行的，一个内存页面是指地址连续的 4K 字节物理内存。通过页目录项和页表项，可以寻址和管理指定页面的使用情况。在 Linux 0.11 的内存管理目录中共有三个文件，如列表 13-1 中所示：

列表 13-1 内存管理子目录文件列表

	名称	大小	最后修改时间 (GMT)	说明
	Makefile	813 bytes	1991-12-02 03:21:45	
	memory.c	11223 bytes	1991-12-03 00:48:01	
	page.s	508 bytes	1991-10-02 14:16:30	

其中，page.s 文件比较短，仅包含内存页异常的中断处理过程（int 14）。主要实现了对缺页和页写保护的处理。memory.c 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和内核其他部分对内存的申请处理过程。

13.1 总体功能

在 Intel 80X86 CPU 中，程序在寻址过程中使用的是由段和偏移值构成的地址。该地址并不能直接用来寻址物理内存地址，因此被称为虚拟地址。为了能寻址物理内存，就需要一种地址变换机制将虚拟地址映射或变换到物理内存中，这种地址变换机制就是内存管理的主要功能之一（内存管理的另外一个主要功能是内存的寻址保护机制。由于篇幅所限，本章不对其进行讨论）。虚拟地址通过段管理机制首先转换成一种中间地址形式—CPU 32 位的线性地址，然后使用分页管理机制将此线性地址映射到物理地址。

为了弄清 Linux 内核对内存的管理操作方式，我们需要了解内存分页管理的工作原理，了解其寻址的机制。分页管理的目的是将物理内存页面映射到某一线性地址处。在分析本章的内存管理程序时，需明确区分清楚给定的地址是指线性地址还是实际物理内存的地址。

13.1.1 内存分页管理机制

在 Intel 80x86 的系统中，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。见图 13-1 所示。其中页目录表和页表的结构是一样的，表项结构也相同，见下面图 13-4 所示。页目录表中的每个表项（简称页目录项）（4 字节）用来寻址一个页表，而每个页表项（4 字节）用来指定一页物理内存页。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。这样在 80386 中，一个页目录表所寻址的所有页表共可以寻址 $1024 \times 1024 \times 4096 = 4\text{G}$ 的内存空间。在 Linux 0.11 内核中，所有进程都使用一个页目录表，而每个进程都有自己的页表。内核代码和数据段长度是 16MB，使用了 4 个页表（即 4 个页目录项）。这 4 个页表直接位于页目录表后面，参见 head.s 程序第 109--125 行。经过分段机制变换，内核代码和数据段位于线性地址空间的头 16MB 范围内，再经过分页机制变换，它被直接一一对应地映射到 16MB 的物理内存上。因此对于内核

段来讲其线性地址就是物理地址。

对于应用进程或内核其他部分来讲，在申请内存时使用的是线性地址。接下来我们就要问了：“那么，一个线性地址如何使用这两个表来映射到一个物理地址上呢？”。为了使用分页机制，一个 32 位的线性地址被分成了三个部分，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。见图 13-2 所示。

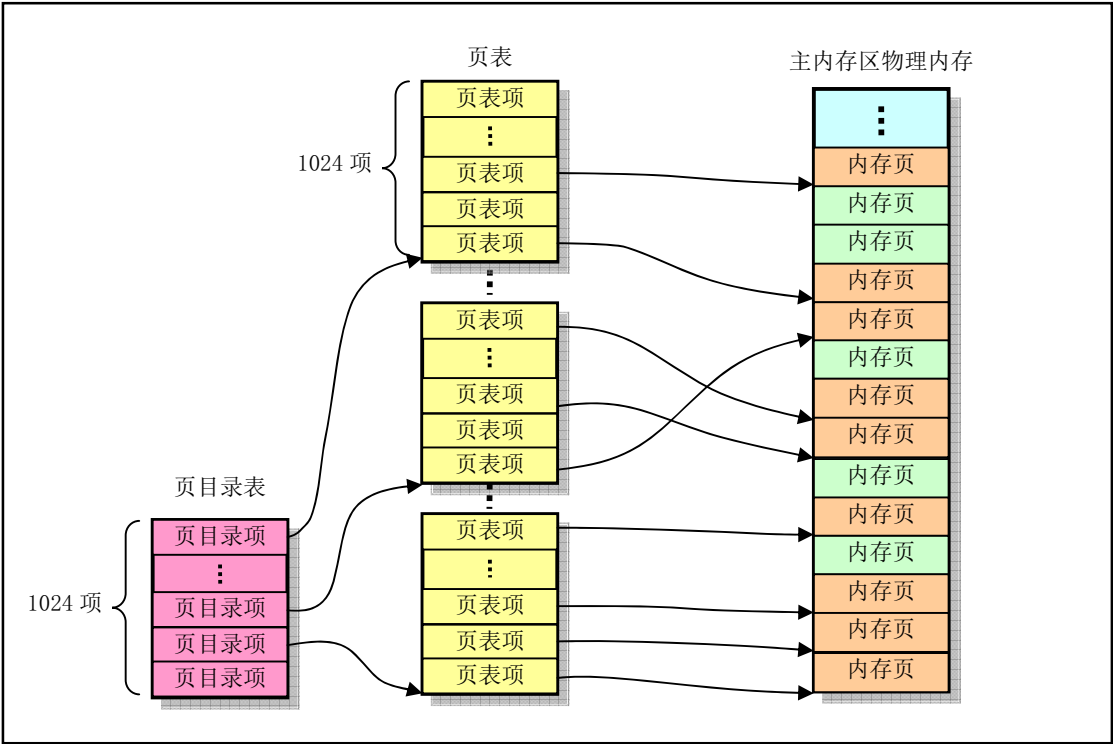


图 13-1 页目录表和页表结构示意图

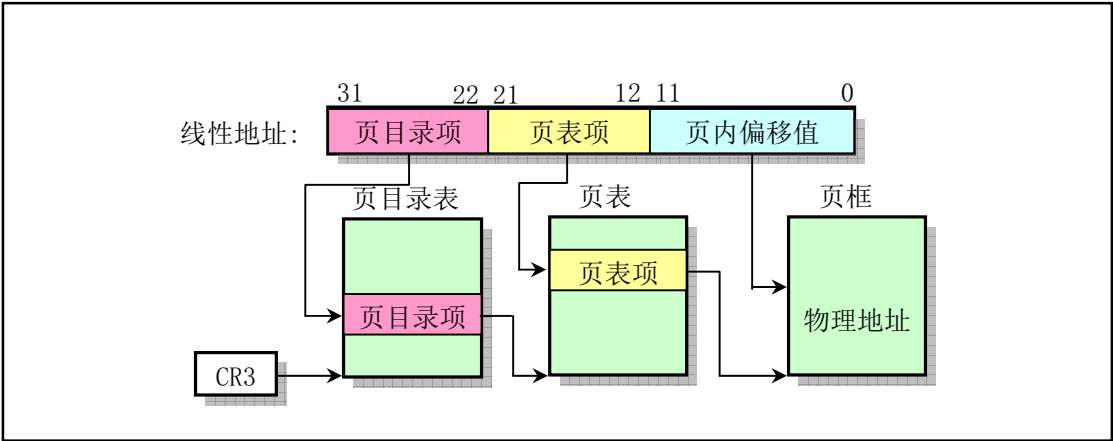


图 13-2 线性地址变换示意图

线性地址的位 31-22 共 10 个比特用来确定页目录中的目录项，位 21-12 用来寻址页目录项指定的页表中的页表项，最后的 12 个比特正好用作页表项指定的一页物理内存中的偏移地址。

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，通过图 13-2 中所示的地址变换关系，我们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（被使用），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。反之，如果需要一个已知被使用的物理内存页地址，寻找对应的线性地址，则需要对整个页目录表 and 所有页表进行搜索。若该物理内存页被共享，我们就可能会找到多个对应的线性地址来。图 13-3 用形象的方法示出了一个给定的线性地址是如何映射到物理内存页上的。对于第一个进程（任务 0），其页表是在页目录表之后，共 4 页。对于应用程序的进程，其页表所使用的内存是在进程创建时向内存管理程序申请的，因此是在主内存区中。

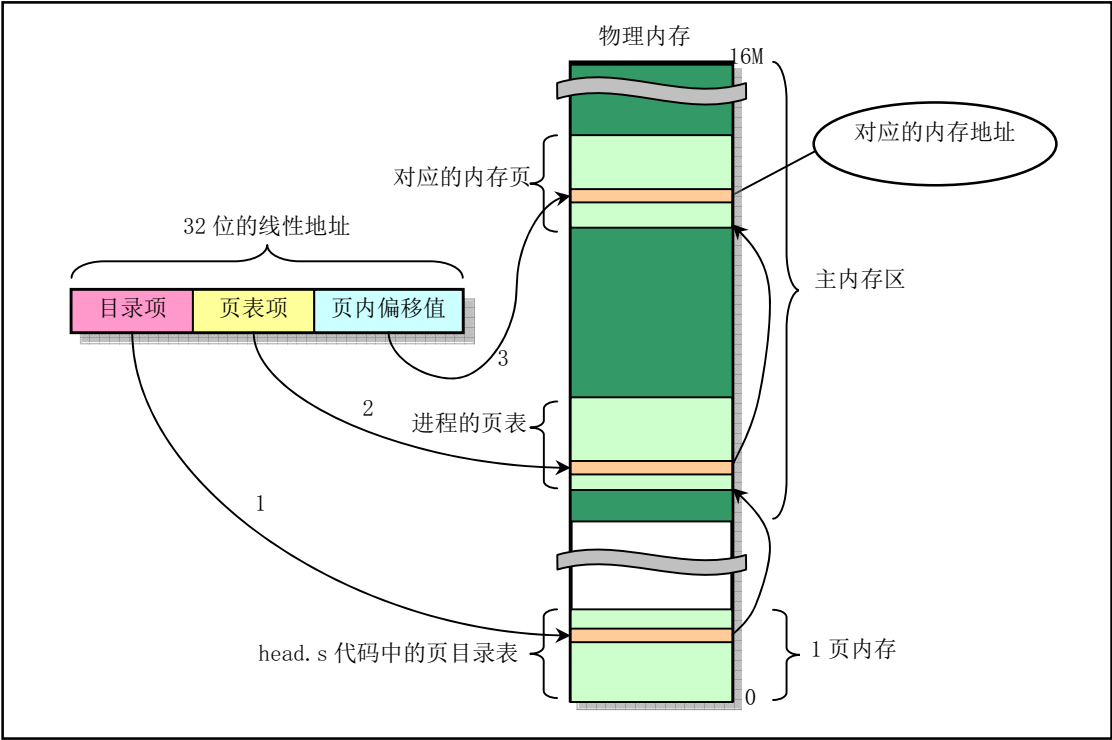


图 13-3 线性地址对应的物理地址

一个系统中可以同时存在多个页目录表，而在某个时刻只有一个页目录表可用。当前的页目录表是用 CPU 的寄存器 CR3 来确定的，它存储着当前页目录表的物理内存地址。但在本书所讨论的 Linux 内核中只使用了一个页目录表。

在图 13-1 中我们看到，每个页表项对应的物理内存页在 4G 的地址范围内是随机的，是由页表项中页框地址内容确定的，也即是由内存管理程序通过设置页表项确定的。每个表项由页框地址、访问标志位、脏（已改写）标志位和存在标志位等构成。表项的结构可参见图 13-4 所示。

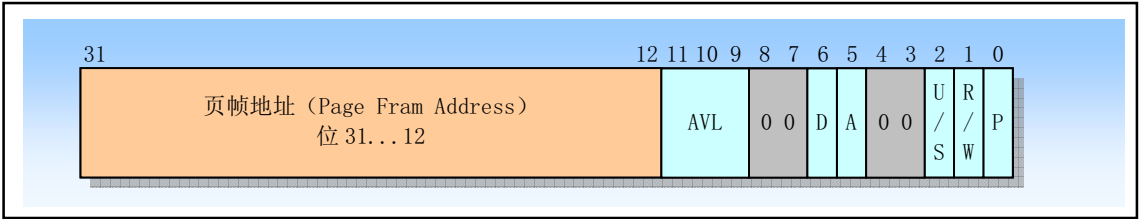


图 13-4 页目录和页表表项结构

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。

图中的存在位 (PRESENT - 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, 则该表项是无效的, 不能用于地址转换过程。此时该表项的所有其他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 (Accessed - A) 和已修改 (Dirty - 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。页目录项和页表项的小区别在于页表项有个已写位 D (Dirty), 而页目录项则没有。

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 (Read/Write - R/W) 和用户/超级用户位 (User/Supervisor - 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。

13.1.2 Linux 中物理内存的管理和分配

有了以上概念, 我们就可以说明 Linux 进行内存管理的方法了。但还需要了解一下 Linux 0.11 内核使用内存空间的情况。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。整个物理内存各部分的功能示意图如图 13-5 所示。

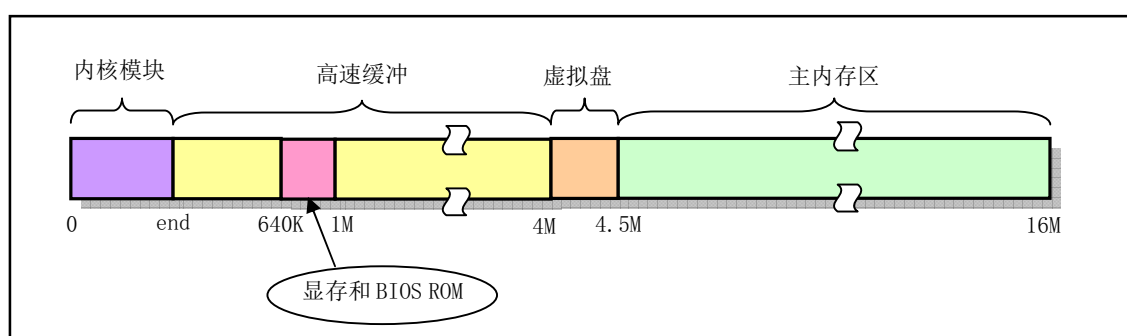


图 13-5 主内存区域示意图

在启动引导一章中, 我们已经知道, Linux 的页目录和页表是在程序 head.s 中设置的。head.s 程序在物理地址 0 处存放了一个页目录表, 紧随其后是 4 个页表。这 4 个页表将被用于内核所占内存区域的影射操作。由于任务 0 的代码和数据包含在内核区域中, 因此任务 0 也使用这些页表。其他的派生进程将在主内存区申请内存页来存放自己的页表。本章中的两个程序就是用于对这些表进行管理操作, 从而实

现对主内存区中内存页面的分配使用。

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。

`page.s` 程序用于实现页异常中断处理过程（`int 14`）。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。

13.1.3 Linux 内核对线性地址空间的使用分配

在阅读本章代码时，我们还需要了解一个执行程序进程的代码和数据在其逻辑地址空间中的分布情况，参见下面图 5-12 所示。

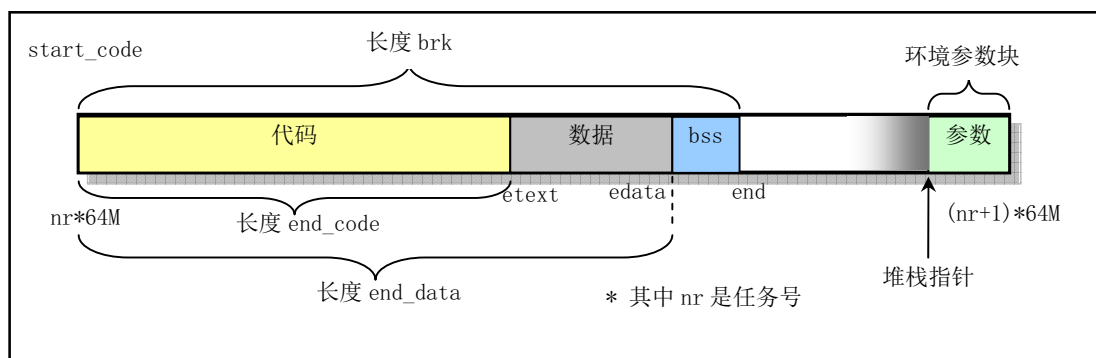


图 13-6 进程代码和数据在其逻辑地址空间中的分布

每个进程在线性地址中都是从 $nr \times 64\text{MB}$ 的地址位置开始（ nr 是任务号），占用逻辑地址空间的范围是 64MB （当然也是线性地址空间的范围）。其中最后部的环境参数数据块最长为 128K ，其左面是起始堆栈指针。另外，图中 `bss` 是进程未初始化的数据段，在进程创建时 `bss` 段的第一页会被初始化为全 0。

13.1.4 页面出错异常处理

在运行于开启了分页机制（`PG=1`）的状态下，若 CPU 在执行线性地址变换到物理地址的过程中检测到以下条件，就会引起页出错异常中断 `int 14`：

- 地址变换过程中用到的页目录项或页表项中存在位（`P`）等于 0；
- 当前执行程序没有足够的特权访问指定的页面。

此时 CPU 会向页出错异常处理程序提供以下两方面信息来协助诊断和纠正错误：

- 栈中的一个出错码（error code）。出错码的格式是一个 32 位的长字。但只有最低 3 个比特有用，它们的名称与页表项中的最后三位相同（`U/S`、`W/R`、`P`）。它们的含义和作用分别是：
 - ◆ 位 0（`P`），异常是由于页面不存在或违反访问特权而引发。`P=0`，表示页不存在；`P=1` 表示违反页级保护权限。
 - ◆ 位 1（`W/R`），异常是由于内存读或写操作引起。`W/R=0`，表示由读操作引起；`W/R=1`，表示由写操作引起。
 - ◆ 位 2（`U/S`），发生异常时 CPU 执行的代码级别。`U/S=0`，表示 CPU 正在执行超级用户代码；`U/S=1`，表示 CPU 正在执行一般用户代码。
- 在控制寄存器 `CR2` 中的线性地址。CPU 会把引起异常的访问使用的线性地址存放在 `CR2` 中。页出错异常处理程序可以使用这个地址来定位相关的页目录和页表项。

后面将要描述的 `page.s` 程序就是利用以上信息来区分是缺页异常还是写保护异常，从而确定调用

memory.c 程序中的缺页处理函数 `do_no_page()` 或写保护函数 `do_wp_page()` 函数。

13.1.5 写时复制 (copy on write) 机制

写时复制是一种推迟或免除复制数据的一种方法。此时内核并不去复制进程整个地址空间中的数据，而是让父进程和子进程共享同一个拷贝。当进程 A 使用系统调用 `fork` 创建出一个子进程 B 时，由于子进程 B 实际上是父进程 A 的一个拷贝，因此会拥有与父进程相同的物理页面。也即为了达到节约内存和加快创建进程速度的目标，`fork()` 函数会让子进程 B 以只读方式共享父进程 A 的物理页面。同时将父进程 A 对这些物理页面的访问权限也设成只读（详见 memory.c 程序中的 `copy_page_tables()` 函数）。这样一来，当父进程 A 或子进程 B 任何一方对这些共享物理页面执行写操作时，都会产生页面出错异常 (page_fault int14) 中断，此时 CPU 就会执行系统提供的异常处理函数 `do_wp_page()` 来试图解决这个异常。这就是写时复制机制。

`do_wp_page()` 会对这块导致写入异常中断的物理页面进行取消共享操作（使用 `un_wp_page()` 函数），并为写进程复制一新的物理页面，使父进程 A 和子进程 B 各自拥有一块内容相同的物理页面。这时才真正地进行了复制操作（只复制这一块物理页面）。并且把将要执行写入操作的这块物理页面标记成可以写访问的。最后，从异常处理函数中返回时，CPU 就会重新执行刚才导致异常的写入操作指令，使进程能够继续执行下去。

因此，对于进程在自己的虚拟地址范围内进行写操作时，就会使用上面这种被动的写时复制操作，也即：写操作 -> 页面异常中断 -> 处理写保护异常 -> 重新执行写操作指令。而对于系统内核代码，当在某个进程的虚拟地址范围内执行写操作时，例如进程调用某个系统调用，若该系统调用会将数据复制到进程的缓冲区域中，则内核会通过 `verify_area()` 函数首先主动地调用内存页面验证函数 `write_verify()`，来判断是否有页面共享的情况存在，如果有，就进行页面的写时复制操作。

另外，值得注意的一点是在 Linux 0.11 内核中，在内核代码地址空间（线性地址 < 1MB）执行 `fork()` 来创建进程使并没有采用写时复制技术。因此当进程 0（idle 进程）在内核空间创建进程 1（init 进程）时将使用同一段代码和数据段。但由于进程 1 复制的页表项也是只读的，因此当进程 1 需要执行堆栈（写）操作时也会引起页面异常，从而在这种情况下内存管理程序也会在主内存区中为该进程分配内存。

由此可见，写时复制把对内存页面的复制操作推迟到实际要进行写操作的时刻，在页面不会被写的情况下就可以根本不用进行页面复制操作。例如，当 `fork()` 创建了一个进程后立即调用 `execve()` 去执行一个新程序的时候。因此这种技术可以避免不必要的内存页面复制的开销。

13.1.6 需求加载 (Load on demand) 机制

在使用 `execve()` 系统调用加载运行文件系统上的一个执行映像文件时，内核除了在 CPU 的 4G 线性地址空间中为对应进程分配了 64MB 的连续空间，并为其环境参数和命令行参数分配和映射了一定数量的物理内存页面以外，实际上并没有给执行程序分配其它任何物理内存页面。当然也谈不上从文件系统上加载执行映像文件中的代码和数据。因此一旦该程序从设定的入口执行点开始运行就会立刻引起 CPU 产生一个缺页异常（执行指针所在的内存页面不存在）。此时内核的缺页异常处理程序才会根据引起缺页异常的具体线性地址把执行文件中相关的代码页从文件系统中加载到物理内存页面中，并映射到进程逻辑地址中指定的页面位置处。当异常处理程序返回后 CPU 就会重新执行引起异常的指令，使得执行程序能够得以继续执行。若在执行过程中又要运行到另一页中还未加载的代码，或者代码指令需要访问还未加载的数据，那么 CPU 同样会产生一个缺页异常中断，此时内核就又会把执行程序中的其他对应页面内容加载到内存中。就这样，执行文件中只有运行到（用到）的代码或数据页面才会被内核加载到物理内存中。这种仅在实际需要时才加载执行文件中页面的方法被称为需求加载 (Load on demand) 技术或需求分页 (demand-paging) 技术。

采用需求加载技术的一个明显优点是在调用 `execve()` 系统后能够让执行程序立刻开始运行，而无需等待多次的块设备 I/O 操作把整个执行文件映像加载到内存中后才开始运行。因此系统对执行程序的加

载执行速度将大大地提高。但这种技术对被加载执行目标文件的格式有一定要求。它要求被执行的文件目标格式是 ZMAGIC 类型的，即需求分页格式的目标文件格式。在这种目标文件格式中，程序的代码段和数据段都从页面边界开始存放，以适应内核以一个页面为单位读取代码或数据内容。

13.2 Makefile 文件

13.2.1 功能描述

本文件是 mm 目录中程序的编译管理配置文件，共 make 程序使用。

13.2.2 代码注释

程序 13-1 linux/mm/Makefile

```

1 CC      =gcc      # GNU C 语言编译器。
2 CFLAGS  =-O -Wall -fstrength-reduce -fcombine-regs -fomit-frame-pointer \
3          -finline-functions -nostdinc -I../include
# C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
# -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
# 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
# 单短小的函数代码嵌入调用程序中；-nostdinc -I../include 不使用默认路径中的包含文件，而
# 使用这里指定目录中的(../include)。
4 AS      =gas      # GNU 的汇编程序。
5 AR      =gar      # GNU 的二进制文件处理程序，用于创建、修改以及从归档文件中抽取文件。
6 LD      =gld      # GNU 的连接程序。
7 CPP     =gcc -E -nostdinc -I../include
# C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
# 出设备或指定的输出文件中；-nostdinc -I../include 同前。
8
# 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
# 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止（-S），从而产生与
# 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
# 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。其中$.s（或$@）是自动目标变量，
# $<代表第一个先决条件，这里即是符合条件*.c 的文件。
# 下面这 3 个不同规则分别用于不同的操作要求。若目标是.s 文件，而源文件是.c 文件则会使
# 用第一个规则；若目标是.o，而原文件是.s，则使用第 2 个规则；若目标是.o 文件而原文件
# 是.c 文件，则可直接使用第 3 个规则。
9 .c.o:
10      $(CC) $(CFLAGS) \
11      -c -o $.o $<
# 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。22 行是实现该操作的具体命令。
12 .s.o:
13      $(AS) -o $.o $<
14 .c.s:      # 类似上面，*.c 文件→*.s 汇编程序文件。不进行连接。
15      $(CC) $(CFLAGS) \
16      -S -o $.s $<
17
18 OBJS     = memory.o page.o  # 定义目标文件变量 OBJS。
19
20 all: mm.o
21
# 在有了先决条件 OBJS 后使用下面的命令连接成目标 mm.o。

```

```

# 选项 '-r' 用于指示生成可重定位的输出，即产生可以作为链接器 ld 输入的目标文件。
22 mm.o: $(OBJS)
23     $(LD) -r -o mm.o $(OBJS)
24
# 下面的规则用于清理工作。当执行'make clean'时，就会执行 26--27 行上的命令，去除所有编译
# 连接生成的文件。'rm' 是文件删除命令，选项-f 含义是忽略不存在的文件，并且不显示删除信息。
25 clean:
26     rm -f core *.o *.a tmp_make
27     for i in *.c;do rm -f `basename $$i .c`.s;done
28
# 下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
# 使用字符串编辑程序 sed 对 Makefile 文件（这里即是自己）进行处理，输出为删除 Makefile
# 文件中'### Dependencies' 行后面的所有行（下面从 35 开始的行），并生成 tmp_make
# 临时文件（30 行的作用）。然后对 mm/目录下的每一个 C 文件执行 gcc 预处理操作。
# -M 标志告诉预处理程序输出描述每个目标文件相关性的规则，并且这些规则符合 make 语法。
# 对于每一个源文件，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标
# 文件名加上其依赖关系—该源文件中包含的所有头文件列表。把预处理结果都添加到临时
# 文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。

29 dep:
30     sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
31     (for i in *.c;do $(CPP) -M $$i;done) >> tmp_make
32     cp tmp_make Makefile
33
34 ### Dependencies:
35 memory.o : memory.c ../include/signal.h ../include/sys/types.h \
36     ../include/asm/system.h ../include/linux/sched.h ../include/linux/head.h \
37     ../include/linux/fs.h ../include/linux/mm.h ../include/linux/kernel.h

```

13.3 memory.c 程序

13.3.1 功能描述

本程序进行内存分页的管理。实现了对主内存区内存页面的动态分配和回收操作。对于内核代码和数据所占物理内存区域以外的内存（1MB 以上内存区域），内核使用了一个字节数组 `mem_map[]` 来表示物理内存页面的状态。每个字节描述一个物理内存页的占用状态。其中的值表示被占用的次数，0 表示对应的物理内存空闲着。当申请一页物理内存时，就将对应字节的值增 1。

在内存管理初始化过程中，系统首先计算出 1MB 以上内存区域对于的内存页面数（`PAGING_PAGES`），并把 `mem_map[]` 所有项都置为 100（占用），然后把主内存区域对应的 `mem_map[]` 项中的值清零。因此内核所使用的位于 1MB 地址以上的高速缓冲区域以及虚拟磁盘区域（若有的话）都已经被初始化成占用状态。`mem_map[]` 中对应主内存区域的项则在系统使用过程中进行设置或复位。例如，对于图 13-5 所示的具有 16MB 物理内存并设置了 512KB 虚拟磁盘的机器，`mem_map[]` 数组共有 $(16\text{MB} - 1\text{MB})/4\text{KB} = 3840$ 项，即对应 3840 个页面。其中主内存区拥有的页面数为 $(16\text{MB} - 4.5\text{MB})/4\text{KB} = 2944$ 个，对应 `mem_map[]` 数组的最后 2944 项，而前 896 项则对应 1MB 以上的高速缓冲区和虚拟磁盘所占有的物理页面。因此在内存管理初始化过程中，`mem_map[]` 的前 896 项被设置为占用状态（值为 100），不可再被分配使用。而后 2944 项的值被清 0，可被内存管理程序分配使用。参见图 13-7 所示。

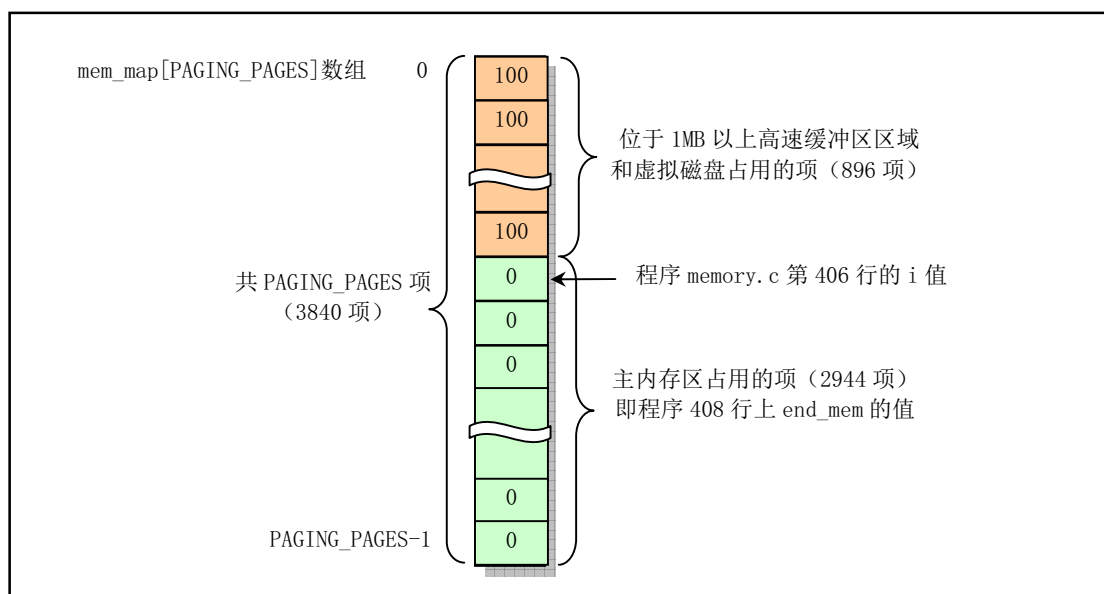


图 13-7 具有 16MB 物理内存和 512KB 虚拟磁盘区机器的 mem_map[] 数组初始化情况

对于进程虚拟地址（或逻辑地址）的管理，内核使用了处理器的页目录表和页表结构来管理。而物理内存页与线性地址之间的映射关系则是通过修改页目录和页表项的内容来处理。下面对程序中所提供的几个主要函数进行详细说明。

get_free_page()和 free_page()这两个函数是专门用来管理主内存区中物理内存的占用和空闲情况，与每个进程的线性地址无关。

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。

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。

free_page_tables()和 copy_page_tables()这两个函数则以一个页表对应的物理内存块（4M）为单位，释放或复制指定线性地址和长度（页表个数）对应的物理内存页块。不仅对管理线性地址的页目录和页表中的对应项内容进行修改，而且也对每个页表中所有页表项对应的物理内存页进行释放或占用操作。

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。

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。

`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。

`do_wp_page()` 是页异常中断过程（在 `mm/page.s` 中实现）中调用的页写保护处理函数。它首先判断地址是否在进程的代码区域，若是则终止程序（代码不能被改动）；然后执行写时复制页面的操作（Copy on Write）。

`do_no_page()` 是页异常中断过程中调用的缺页处理函数。它首先判断指定的线性地址在一个进程空间中相对于进程基址的偏移长度值。如果它大于代码加数据长度，或者进程刚开始创建，则立刻申请一页物理内存，并映射到进程线性地址中，然后返回；接着尝试进行页面共享操作，若成功，则立刻返回；否则申请一页内存并从设备中读入一页信息；若加入该页信息时，指定线性地址+1 页长度超过了进程代码加数据的长度，则将超过的部分清零。然后将该页映射到指定的线性地址处。

`get_empty_page()` 用于取得一页空闲物理内存并映射到指定线性地址处。主要使用了 `get_free_page()` 和 `put_page()` 函数来实现该功能。

13.3.2 代码注释

程序 13-2 linux/mm/memory.c

```

1  /*
2   *  linux/mm/memory.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *  demand-loading started 01.12.91 - seems it is high on the list of
9   *  things wanted, and it should be easy to implement. - Linus
10  */
11  /*
12   * 需求加载是从 01.12.91 开始编写的 - 在程序编制表中似乎是最重要的程序，
13   * 并且应该是很容易编制的 - Linus
14   */
15
16  /*
17   *  Ok, demand-loading was easy, shared pages a little bit trickier. Shared
18   *  pages started 02.12.91, seems to work. - Linus.
19   *
20   *  Tested sharing by executing about 30 /bin/sh: under the old kernel it
21   *  would have taken more than the 6M I have free, but it worked well as
22   *  far as I could see.
23   *
24   *  Also corrected some "invalidate()"s - I wasn't doing enough of them.
25   */
26  /*
27   *  OK, 需求加载是比较容易编写的，而共享页面却需要有点技巧。共享页面程序是
28   *  02.12.91 开始编写的，好象能够工作 - Linus。

```

```

*
* 通过执行大约 30 个/bin/sh 对共享操作进行了测试：在老内核当中需要占用多于
* 6M 的内存，而目前却不用。现在看来工作得很好。
*
* 对"invalidate()"函数也进行了修正 - 在这方面我还做的不够。
*/

22
23 #include <signal.h>          // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。
24
25 #include <asm/system.h>     // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
26
27 #include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                               // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
28 #include <linux/head.h>     // head 头文件，定义了段描述符的简单结构，和几个选择符常量。
29 #include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原型定义。
30
// 函数名前的关键字 volatile 用于告诉编译器 gcc 该函数不会返回。这样可让 gcc 产生更好一
// 些的代码，更重要的是使用这个关键字可以避免产生某些（未初始化变量的）假警告信息。
31 volatile void do_exit(long code); // 进程退出处理函数，在 kernel/exit.c, 102 行。
32
//// 显示内存已用完出错信息，并退出。
33 static inline volatile void oom(void)
34 {
35     printk("out of memory\n");
36     do_exit(SIGSEGV);          // do_exit() 应该使用退出代码，这里用了信号值 SIGSEGV(11)
37 }                             // 相同值的出错码含义是“资源暂时不可用”，正好同义。
38
// 刷新页变换高速缓冲宏函数。
// 为了提高地址转换的效率，CPU 将最近使用的页表数据存放在芯片中高速缓冲中。在修改过
// 页表信息之后，就需要刷新该缓冲区。这里使用重新加载页目录基址寄存器 cr3 的方法来
// 进行刷新。下面 eax = 0，是页目录的基址。
39 #define invalidate() \
40 __asm__ ("movl %%eax, %%cr3::: \"a\" (0)")
41
42 /* these are not to be changed without changing head.s etc */
/* 下面定义若需要改动，则需要与 head.s 等文件中的相关信息一起改变 */
// Linux 0.11 内核默认支持的最大内存容量是 16MB，可以修改这些定义以适合更多的内存。
43 #define LOW_MEM 0x100000      // 内存低端（1MB）。
44 #define PAGING_MEMORY (15*1024*1024) // 分页内存 15MB。主内存区最多 15M。
45 #define PAGING_PAGES (PAGING_MEMORY>>12) // 分页后的物理内存页面数（3840）。
46 #define MAP_NR(addr) (((addr)-LOW_MEM)>>12) // 指定内存地址映射为页号。
47 #define USED 100              // 页面被占用标志，参见 405 行。
48
// CODE_SPACE(addr) (((addr)+0xfff)&~0xfff)<current->start_code+current->end_code)。
// 该宏用于判断给定线性地址是否位于当前进程的代码段中，“((addr)+4095)&~4095”用于
// 取得线性地址 addr 所在内存页面的末端地址。参见 252 行。
49 #define CODE_SPACE(addr) (((addr)+4095)&~4095) < \
50 current->start_code + current->end_code)
51
52 static long HIGH_MEMORY = 0; // 全局变量，存放实际物理内存最高端地址。
53
// 从 from 处复制 1 页内存到 to 处（4K 字节）。
54 #define copy_page(from,to) \

```

```

55 __asm__( "cld ; rep ; movsl":: "S" (from), "D" (to), "c" (1024): "cx", "di", "si")
56
    // 物理内存映射字节图（1 字节代表 1 页内存）。每个页面对应的字节用于标志页面当前被引用
    // （占用）次数。它最大可以映射 15Mb 的内存空间。在初始化函数 mem_init() 中，对于不能用
    // 作主内存区页面的位置均都预先被设置成 USED（100）。
57 static unsigned char mem_map [ PAGING_PAGES ] = {0,};
58
59 /*
60  * Get physical address of first (actually last :-) free page, and mark it
61  * used. If no free pages left, return 0.
62  */
    /*
    * 获取首个（实际上是最后 1 个:-）空闲页面，并标记为已使用。如果没有空闲页面，
    * 就返回 0。
    */
    // 在主内存区中取空闲物理页面。如果已经没有可用物理内存页面，则返回 0。
    // 输入：%1(ax=0) - 0；%2(LOW_MEM) 内存字节位图管理的起始位置；%3(cx= PAGING_PAGES)；
    // %4(edi=mem_map+PAGING_PAGES-1)。
    // 输出：返回%0 (ax = 物理页面起始地址)。
    // 上面%4 寄存器实际指向 mem_map[] 内存字节位图的最后一个字节。本函数从位图末端开始向
    // 前扫描所有页面标志（页面总数为 PAGING_PAGES），若有页面空闲（内存位图字节为 0）则
    // 返回页面地址。注意！本函数只是指出在主内存区的一页空闲物理页面，但并没有映射到某
    // 个进程的地址空间中去。后面的 put_page() 函数即用于把指定页面映射到某个进程的地址
    // 空间中。当然对于内核使用本函数并不需要再使用 put_page() 进行映射，因为内核代码和
    // 数据空间（16MB）已经对等地映射到物理地址空间。
    // 第 65 行定义了一个局部寄存器变量。该变量将被保存在 eax 寄存器中，以便于高效访问和
    // 操作。这种定义变量的方法主要用于内嵌汇编程序中。详细说明参见 gcc 手册“在指定寄存
    // 器中的变量”。
63 unsigned long get_free_page(void)
64 {
65     register unsigned long __res asm("ax");
66
67     __asm__( "std ; repne ; scasb\n\t" // 置方向位，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" // 1 =>[1+edi]，将对应页面内存映像比特位置 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，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，即将该页面清零）。
76             "movl %%edx,%%eax\n\t" // 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; // 返回空闲物理页面地址（若无空闲页面则返回 0）。
83 }
84
85 /*
86  * Free a page of memory at physical address 'addr'. Used by
87  * 'free_page_tables()'

```

```

88  */
/*
   * 释放物理地址'addr'处的一页内存。用于函数'free_page_tables()'。
   */
//// 释放物理地址 addr 开始的 1 页面内存。
// 物理地址 1MB 以下的内存空间用于内核程序和缓冲，不作为分配页面的内存空间。因此
// 参数 addr 需要大于 1MB。
89 void free_page(unsigned long addr)
90 {
    // 首先判断参数给定的物理地址 addr 的合理性。如果物理地址 addr 小于内存低端（1MB），
    // 则表示在内核程序或高速缓冲中，对此不予处理。如果物理地址 addr >= 系统所含物理
    // 内存最高端，则显示出错信息并且内核停止工作。
91     if (addr < LOW_MEM) return;
92     if (addr >= HIGH_MEMORY)
93         panic("trying to free nonexistent page");
    // 如果对参数 addr 验证通过，那么就根据这个物理地址换算出从内存低端开始计起的内存
    // 页面号。页面号 = (addr - LOW_MEM)/4096。可见页面号从 0 号开始计起。此时 addr
    // 中存放着页面号。如果该页面号对应的页面映射字节不等于 0，则减 1 返回。此时该映射
    // 字节值应该为 0，表示页面已释放。如果对应页面字节原本就是 0，表示该物理页面本来
    // 就是空闲的，说明内核代码出问题。于是显示出错信息并停机。
94     addr -= LOW_MEM;
95     addr >>= 12;
96     if (mem_map[addr]-->0) return;
97     mem_map[addr]=0;
98     panic("trying to free free page");
99 }
100
101 /*
102  * This function frees a continuous block of page tables, as needed
103  * by 'exit()'. As does copy_page_tables(), this handles only 4Mb blocks.
104  */
/*
   * 下面函数释放页表连续的内存块，'exit()'需要该函数。与 copy_page_tables()
   * 类似，该函数仅处理 4Mb 长度的内存块。
   */
//// 根据指定的线性地址和限长（页表个数），释放对应内存页表指定的内存块并置表项空闲。
// 页目录位于物理地址 0 开始处，共 1024 项，每项 4 字节，共占 4K 字节。每个目录项指定一
// 个页表。内核页表从物理地址 0x1000 处开始（紧接着目录空间），共 4 个页表。每个页表有
// 1024 项，每项 4 字节。因此也占 4K（1 页）内存。各进程（除了在内核代码中的进程 0 和 1）
// 的页表所占据的页面在进程被创建时由内核为其在主内存区申请得到。每个页表项对应 1 页
// 物理内存，因此一个页表最多可映射 4MB 的物理内存。
// 参数：from - 起始线性基地址；size - 释放的字节长度。
105 int free_page_tables(unsigned long from, unsigned long size)
106 {
107     unsigned long *pg_table;
108     unsigned long *dir, nr;
109
    // 首先检测参数 from 给出的线性基地址是否在 4MB 的边界处。因为该函数只能处理这种情况。
    // 若 from = 0，则出错。说明试图释放内核和缓冲所占空间。
110     if (from & 0x3ffff)
111         panic("free_page_tables called with wrong alignment");
112     if (!from)
113         panic("Trying to free up swapper memory space");

```



```

// 然后计算参数 size 给出的长度所占的页目录项数（4MB 的进位整数倍），也即所占页表数。
// 因为 1 个页表可管理 4MB 物理内存，所以这里用右移 22 位的方式把需要复制的内存长度值
// 除以 4MB。其中加上 0x3fffff（即 4Mb -1）用于得到进位整数倍结果，即除操作若有余数
// 则进 1。例如，如果原 size = 4.01Mb，那么可得到结果 size = 2。接着计算给出的线性
// 基地址对应的起始目录项。对应的目录项号 = from >> 22。因为每项占 4 字节，并且由于
// 页目录表从物理地址 0 开始存放，因此实际目录项指针 = 目录项号<<2，也即 (from>>20)。
// “与”上 0xffc 确保目录项指针范围有效，即用于屏蔽目录项指针最后 2 位。因为只移动
// 了 20 位，因此最后 2 位是页表项索引的内容，应屏蔽掉。
114     size = (size + 0x3fffff) >> 22;
115     dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */

// 此时 size 是释放的页表个数，即页目录项数，而 dir 是起始目录项指针。现在开始循环
// 操作页目录项，依次释放每个页表中的页表项。如果当前目录项无效（P 位=0），表示该
// 目录项没有使用（对应的页表不存在），则继续处理下一个目录项。否则从目录项中取出
// 页表地址 pg_table，并对该页表中的 1024 个表项进行处理，释放有效页表项（P 位=1）
// 对应的物理内存页面。然后把该页表项清零，并继续处理下一页表项。当一个页表所有
// 表项都处理完毕就释放该页表自身占据的内存页面，并继续处理下一页目录项。最后刷新
// 页变换高速缓冲，并返回 0。
116     for ( ; size-->0 ; dir++) {
117         if (!(1 & *dir))
118             continue;
119         pg_table = (unsigned long *) (0xfffff000 & *dir); // 取页表地址。
120         for (nr=0 ; nr<1024 ; nr++) {
121             if (1 & *pg_table) // 若该项有效，则释放对应页。
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。
127         *dir = 0; // 对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 }
132
133 /*
134  * Well, here is one of the most complicated functions in mm. It
135  * copies a range of linear addresses by copying only the pages.
136  * Let's hope this is bug-free, 'cause this one I don't want to debug :-)
137  *
138  * Note! We don't copy just any chunks of memory - addresses have to
139  * be divisible by 4Mb (one page-directory entry), as this makes the
140  * function easier. It's used only by fork anyway.
141  *
142  * NOTE 2!! When from==0 we are copying kernel space for the first
143  * fork(). Then we DONT want to copy a full page-directory entry, as
144  * that would lead to some serious memory waste - we just copy the
145  * first 160 pages - 640kB. Even that is more than we need, but it
146  * doesn't take any more memory - we don't copy-on-write in the low
147  * 1 Mb-range, so the pages can be shared with the kernel. Thus the
148  * special case for nr=xxxx.
149  */
150 /*

```



```

* 好了，下面是内存管理 mm 中最为复杂的程序之一。它通过只复制内存页面
* 来拷贝一定范围内线性地址中的内容。希望代码中没有错误，因为我不想
* 再调试这块代码了:-)。
*
* 注意！我们并不复制任何内存块 - 内存块的地址需要是 4Mb 的倍数（正好
* 一个页目录项对应的内存长度），因为这样处理可使函数很简单。不管怎
* 样，它仅被 fork() 使用。
*
* 注意 2！！当 from==0 时，说明是在为第一次 fork() 调用复制内核空间。
* 此时我们就不想复制整个页目录项对应的内存，因为这样做会导致内存严
* 重浪费 - 我们只须复制开头 160 个页面 - 对应 640kB。即使是复制这些
* 页面也已经超出我们的需求，但这不会占用更多的内存 - 在低 1Mb 内存
* 范围内我们不执行写时复制操作，所以这些页面可以与内核共享。因此这
* 是 nr=xxxx 的特殊情况（nr 在程序中指页面数）。
*/
//// 复制页目录表项和页表项。
// 复制指定线性地址和长度内存对应的页目录项和页表项，从而被复制的页目录和页表对应
// 的原物理内存页面区被两套页表映射而共享使用。复制时，需申请新页面来存放新页表，
// 原物理内存区将被共享。此后两个进程（父进程和其子进程）将共享内存区，直到有一个
// 进程执行写操作时，内核才会为写操作进程分配新的内存页（写时复制机制）。
// 参数 from、to 是线性地址，size 是需要复制（共享）的内存长度，单位是字节。
150 int copy_page_tables(unsigned long from, unsigned long to, long size)
151 {
152     unsigned long * from_page_table;
153     unsigned long * to_page_table;
154     unsigned long this_page;
155     unsigned long * from_dir, * to_dir;
156     unsigned long nr;
157
// 首先检测参数给出的源地址 from 和目的地址 to 的有效性。源地址和目的地址都需要在 4Mb
// 内存边界地址上。否则出错死机。作这样的要求是因为一个页表的 1024 项可管理 4Mb 内存。
// 源地址 from 和目的地址 to 只有满足这个要求才能保证从一个页表的第 1 项开始复制页表
// 项，并且新页表的最初所有项都是有效的。然后取得源地址和目的地址的起始目录项指针
// （from_dir 和 to_dir）。再根据参数给出的长度 size 计算要复制的内存块占用的页表数
// （即目录项数）。参见前面对 114、115 行的解释。
158     if ((from & 0x3ffff) || (to & 0x3ffff))
159         panic("copy_page_tables called with wrong alignment");
160     from_dir = (unsigned long *) ((from >> 20) & 0xffc); /* _pg_dir = 0 */
161     to_dir = (unsigned long *) ((to >> 20) & 0xffc);
162     size = ((unsigned) (size + 0x3ffff)) >> 22;
// 在得到了源起始目录项指针 from_dir 和目的起始目录项指针 to_dir 以及需要复制的页表
// 个数 size 后，下面开始对每个页目录项依次申请 1 页内存来保存对应的页表，并且开始
// 页表项复制操作。如果目的目录项指定的页表已经存在（P=1），则出错死机。如果源目
// 录项无效，即指定的页表不存在（P=0），则继续循环处理下一个页目录项。
163     for( ; size-->0 ; from_dir++, to_dir++) {
164         if (1 & *to_dir)
165             panic("copy_page_tables: already exist");
166         if (!(1 & *from_dir))
167             continue;
// 在验证了当前源目录项和目的项正常之后，我们取源目录项中页表地址 from_page_table。
// 为了保存目的目录项对应的页表，需要在主内存区中申请 1 页空闲内存页。如果取空闲页面
// 函数 get_free_page() 返回 0，则说明没有申请到空闲内存页面，可能是内存不够。于是返
// 回-1 值退出。

```

```

168         from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
169         if (!(to_page_table = (unsigned long *) get_free_page()))
170             return -1;          /* Out of memory, see freeing */
// 否则我们设置目的目录项信息，把最后 3 位置位，即当前目的目录项“或”上 7，表示对应
// 页表映射的内存页面是用户级的，并且可读写、存在 (Usr, R/W, Present)。(如果 U/S
// 位是 0，则 R/W 就没有作用。如果 U/S 是 1，而 R/W 是 0，那么运行在用户层的代码就只能
// 读页面。如果 U/S 和 R/W 都置位，则就有读写的权限)。然后针对当前处理的页目录项对应
// 的页表，设置需要复制的页面项数。如果是在内核空间，则仅需复制头 160 页对应的页表项
// (nr = 160)，对应于开始 640KB 物理内存。否则需要复制一个页表中的所有 1024 个页表项
// (nr = 1024)，可映射 4MB 物理内存。
171         *to_dir = ((unsigned long) to_page_table) | 7;
172         nr = (from==0)?0xA0:1024;
// 此时对于当前页表，开始循环复制指定的 nr 个内存页面表项。先取出源页表项内容，如果
// 当前源页面没有使用，则不用复制该表项，继续处理下一项。否则复位页表项中 R/W 标志
// (位 1 置 0)，即让页表项对应的内存页面只读。然后将该页表项复制到目的页表中。
173         for ( ; nr-- > 0 ; from_page_table++, to_page_table++) {
174             this_page = *from_page_table;
175             if (!(1 & this_page))
176                 continue;
177             this_page &= ~2;
178             *to_page_table = this_page;
// 如果该页表项所指物理页面的地址在 1MB 以上，则需要设置内存页面映射数组 mem_map[]，
// 于是计算页面号，并以它为索引在页面映射数组相应项中增加引用次数。而对于位于 1MB
// 以下的页面，说明是内核页面，因此不需要对 mem_map[] 进行设置。因为 mem_map[] 仅用
// 于管理主内存区中的页面使用情况。因此对于内核移动到任务 0 中并且调用 fork() 创建
// 任务 1 时 (用于运行 init())，由于此时复制的页面还仍然都在内核代码区域，因此以下
// 判断中的语句不会执行，任务 0 的页面仍然可以随时读写。只有当调用 fork() 的父进程
// 代码处于主内存区 (页面位置大于 1MB) 时才会执行。这种情况需要在进程调用 execve()，
// 并装载执行了新程序代码时才会出现。
// 180 行语句含义是令源页表项所指内存页也为只读。因为现在开始有两个进程共用内存区了。
// 若其中 1 个进程需要进行写操作，则可以通过页异常写保护处理为执行写操作的进程分配
// 1 页新空闲页面，也即进行写时复制 (copy on write) 操作。
179             if (this_page > LOW_MEM) {
180                 *from_page_table = this_page; // 令源页表项也只读。
181                 this_page -= LOW_MEM;
182                 this_page >>= 12;
183                 mem_map[this_page]++;
184             }
185         }
186     }
187     invalidate();          // 刷新页变换高速缓冲。
188     return 0;
189 }
190
191 /*
192  * This function puts a page in memory at the wanted address.
193  * It returns the physical address of the page gotten, 0 if
194  * out of memory (either when trying to access page-table or
195  * page.)
196  */
/*
* 下面函数将一内存页面放置 (映射) 到指定线性地址处。它返回页面
* 的物理地址，如果内存不够 (在访问页表或页面时)，则返回 0。

```

```

    */
    // 把一物理内存页面映射到线性地址空间指定处。
    // 或者说是把线性地址空间中指定地址 address 处的页面映射到主内存区页面 page 上。主要
    // 工作是在相关页目录项和页表项中设置指定页面的信息。若成功则返回物理页面地址。在
    // 处理缺页异常的 C 函数 do_no_page() 中会调用此函数。对于缺页引起的异常，由于任何缺
    // 页缘故而对页表作修改时，并不需要刷新 CPU 的页变换缓冲（或称 Translation Lookaside
    // Buffer - TLB），即使页表项中标志 P 被从 0 修改成 1。因为无效页项不会被缓冲，因此当
    // 修改了一个无效的页表项时不需要刷新。在此就表现为不用调用 Invalidate() 函数。
    // 参数 page 是分配的主内存区中某一页面（页帧，页框）的指针；address 是线性地址。
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
    // 首先判断参数给定物理内存页面 page 的有效性。如果该页面位置低于 LOW_MEM（1MB）或
    // 超出系统实际含有内存高端 HIGH_MEMORY，则发出警告。LOW_MEM 是主内存区可能有的最
    // 小起始位置。当系统物理内存小于或等于 6MB 时，主内存区起始于 LOW_MEM 处。再查看一
    // 下该 page 页面是否是已经申请的页面，即判断其在内存页面映射字节图 mem_map[] 中相
    // 应字节是否已经置位。若没有则需发出警告。
203     if (page < LOW_MEM || page >= HIGH_MEMORY)
204         printk("Trying to put page %p at %p\n", page, address);
205     if (mem_map[(page-LOW_MEM)>>12] != 1)
206         printk("mem_map disagrees with %p at %p\n", page, address);
    // 然后根据参数指定的线性地址 address 计算其在页目录表中对应的目录项指针，并从中取得
    // 二级页表地址。如果该目录项有效（P=1），即指定的页表在内存中，则从中取得指定页表
    // 地址放到 page_table 变量中。否则就申请一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）。然后将该页表地址放到 page_table 变量中。参见对 115
    // 行语句的说明。
207     page_table = (unsigned long *) ((address>>20) & 0xffc);
208     if ((*page_table)&1)
209         page_table = (unsigned long *) (0xfffff000 & *page_table);
210     else {
211         if (!(tmp=get_free_page()))
212             return 0;
213         *page_table = tmp|7;
214         page_table = (unsigned long *) tmp;
215     }
    // 最后在找到的页表 page_table 中设置相关页表项内容，即把物理页面 page 的地址填入表
    // 项同时置位 3 个标志（U/S、W/R、P）。该页表项在页表中的索引值等于线性地址位 21 --
    // 位 12 组成的 10 比特的值。每个页表共可有 1024 项（0 -- 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;                // 返回物理页面地址。
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）。
    // 在内核创建进程时，新进程与父进程被设置成共享代码和数据内存页面，并且所有这些页面
    // 均被设置成只读页面。而当新进程或原进程需要向内存页面写数据时，CPU 就会检测到这个
    // 情况并产生页面写保护异常。于是在这个函数中内核就会首先判断要写的页面是否被共享。
    // 若没有则把页面设置成可写然后退出。若页面是出于共享状态，则需要重新申请一新页面并

```

```

// 复制被写页面内容，以供写进程单独使用。共享被取消。本函数供下面 do_wp_page() 调用。
// 输入参数为页表项指针，是物理地址。[ un_wp_page -- Un-Write Protect Page]
221 void un_wp_page(unsigned long * table_entry)
222 {
223     unsigned long old_page, new_page;
224
// 首先取参数指定的页表项中物理页面位置（地址）并判断该页面是否是共享页面。如果原
// 页面地址大于内存低端 LOW_MEM（表示在主内存区中），并且其在页面映射字节图数组中
// 值为 1（表示页面仅被引用 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。即如果该内存页面此时只被一个进程使用，
// 并且不是内核中的进程，就直接把属性改为可写即可，不用再重新申请一个新页面。
225     old_page = 0xfffff000 & *table_entry; // 取指定页表项中物理页面地址。
226     if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
227         *table_entry |= 2;
228         invalidate();
229         return;
230     }
// 否则就需要在主内存区内申请一页空闲页面给执行写操作的进程单独使用，取消页面共享。
// 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
// 面映射字节数组值递减 1。然后将指定页表项内容更新为新页面地址，并置可读写等标志
// （U/S、R/W、P）。在刷新页变换高速缓冲之后，最后将原页面内容复制到新页面上。
231     if (!(new_page=get_free_page()))
232         oom(); // Out of Memory。内存不够处理。
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 }
239
240 /*
241  * This routine handles present pages, when users try to write
242  * to a shared page. It is done by copying the page to a new address
243  * and decrementing the shared-page counter for the old page.
244  *
245  * If it's in code space we exit with a segment error.
246  */
/*
 * 当用户试图往一共享页面上写时，该函数处理已存在的内存页面（写时复制），
 * 它是通过将页面复制到一个新地址上并且递减原页面的共享计数值实现的。
 *
 * 如果它在代码空间，我们就显示段出错信息并退出。
 */
///// 执行写保护页面处理。
// 是写共享页面处理函数。是页异常中断处理过程中调用的 C 函数。在 page.s 程序中被调用。
// 参数 error_code 是进程在写写保护页面时由 CPU 自动产生，address 是页面线性地址。
// 写共享页面时，需复制页面（写时复制）。
247 void do_wp_page(unsigned long error_code, unsigned long address)
248 {
249     #if 0
250     /* we cannot do this yet: the estdio library writes to code space */
251     /* stupid, stupid. I really want the libc.a from GNU */
    /* 我们现在还不能这样做：因为 estdio 库会在代码空间执行写操作 */

```

```

/* 真是太愚蠢了。我真想从 GNU 得到 libc.a 库。*/
252     if (CODE_SPACE(address)) // 如果地址位于代码空间，则终止执行程序。
253         do_exit(SIGSEGV);
254 #endif
// 调用上面函数 un_wp_page() 来处理取消页面保护。但首先需要为其准备好参数。参数是
// 线性地址 address 指定页面在页表中的页表项指针，其计算方法是：
// ① ((address>>10) & 0xffc)：计算指定线性地址中页表项在页表中的偏移地址；因为
// 根据线性地址结构，(address>>12) 就是页表项中的索引，但每项占 4 个字节，因此乘
// 4 后：(address>>12)<<2 = (address>>10)&0xffc 就可得到页表项在表中的偏移地址。
// 与操作&0xffc 用于限制地址范围在一个页面内。又因为只移动了 10 位，因此最后 2 位
// 是线性地址低 12 位中的最高 2 位，也应屏蔽掉。因此求线性地址中页表项在页表中偏
// 移地址直观一些表示方法是(((address>>12) & 0x3ff)<<2)。
// ② (0xfffff000 & *((address>>20) & 0xffc))：用于取目录项中页表的地址值；其中，
// ((address>>20) & 0xffc) 用于取线性地址中的目录索引项在目录表中的偏移位置。因为
// address>>22 是目录项索引值，但每项 4 个字节，因此乘以 4 后：(address>>22)<<2
// = (address>>20) 就是指定项在目录表中的偏移地址。&0xffc 用于屏蔽目录项索引值
// 中最后 2 位。因为只移动了 20 位，因此最后 2 位是页表索引的内容，应该屏蔽掉。而
// *((address>>20) & 0xffc) 则是取指定目录表项内容中对应页表的物理地址。最后与上
// 0xfffff000 用于屏蔽掉页目录项内容中的一些标志位（目录项低 12 位）。直观表示为
// (0xfffff000 & *((unsigned long *) ((address>>22) & 0x3ff)<<2)))。
// ③ 由①中页表项在页表中偏移地址加上 ②中目录表项内容中对应页表的物理地址即可
// 得到页表项的指针（物理地址）。这里对共享的页面进行复制。
255     un_wp_page((unsigned long *)
256                (((address>>10) & 0xffc) + (0xfffff000 &
257                *((unsigned long *) ((address>>20) & 0xffc)))));
258 }
259 }
260
//// 写页面验证。
// 若页面不可写，则复制页面。在 fork.c 中第 34 行被内存验证通用函数 verify_area() 调用。
// 参数 address 是指定页面的线性地址。
261 void write_verify(unsigned long address)
262 {
263     unsigned long page;
264
// 首先取指定线性地址对应的页目录项，根据目录项中的存在位（P）判断目录项对应的页表
// 是否存在（存在位 P=1?），若不存在（P=0）则返回。这样处理是因为对于不存在的页面没
// 有共享和写时复制可言，并且若程序对此不存在的页面执行写操作时，系统就会因为缺页异
// 常而去执行 do_no_page()，并为这个地方使用 put_page() 函数映射一个物理页面。
// 接着程序从目录项中取页表地址，加上指定页面在页表中的页表项偏移值，得对应地址的页
// 表项指针。在该表项中包含着给定线性地址对应的物理页面。
265     if (!(page = *((unsigned long *) ((address>>20) & 0xffc)) & 1))
266         return;
267     page &= 0xfffff000;
268     page += ((address>>10) & 0xffc);
// 然后判断该页表项中的位 1（R/W）、位 0（P）标志。如果该页面不可写（R/W=0）且存在，
// 那么就执行共享检验和复制页面操作（写时复制）。否则什么也不做，直接退出。
269     if ((3 & *((unsigned long *) page) == 1) /* non-writeable, present */
270         un_wp_page((unsigned long *) page);
271     return;
272 }
273
//// 取得一页空闲内存页并映射到指定线性地址处。

```



```

// get_free_page() 仅是申请取得了主内存区的一页物理内存。而本函数则不仅是获取到一页
// 物理内存页面，还进一步调用 put_page(), 将物理页面映射到指定的线性地址处。
// 参数 address 是指定页面的线性地址。
274 void get_empty_page(unsigned long address)
275 {
276     unsigned long tmp;
277
// 若不能取得一空闲页面，或者不能将所取页面放置到指定地址处，则显示内存不够的信息。
// 279 行上英文注释的含义是：free_page() 函数的参数 tmp 是 0 也没有关系，该函数会忽略
// 它并能正常返回。
278     if (!(tmp=get_free_page()) || !put_page(tmp, address)) {
279         free_page(tmp);          /* 0 is ok - ignored */
280         oom();
281     }
282 }
283
284 /*
285  * try_to_share() checks the page at address "address" in the task "p",
286  * to see if it exists, and if it is clean. If so, share it with the current
287  * task.
288  *
289  * NOTE! This assumes we have checked that p != current, and that they
290  * share the same executable.
291  */
/*
 * try_to_share() 在任务 "p" 中检查位于地址 "address" 处的页面，看页面是否存在，
 * 是否干净。如果是干净的话，就与当前任务共享。
 *
 * 注意！这里我们已假定 p != 当前任务，并且它们共享同一个执行程序。
 */
///// 尝试对当前进程指定地址处的页面进行共享处理。
// 当前进程与进程 p 是同一执行代码，也可以认为当前进程是由 p 进程执行 fork 操作产生的
// 进程，因此它们的代码内容一样。如果未对数据段内容作过修改那么数据段内容也应一样。
// 参数 address 是进程中的逻辑地址，即是当前进程欲与 p 进程共享页面的逻辑页面地址。
// 进程 p 是将被共享页面的进程。如果 p 进程 address 处的页面存在并且没有被修改过的话，
// 就让当前进程与 p 进程共享之。同时还需要验证指定的地址处是否已经申请了页面，若是
// 则出错，死机。返回：1 - 页面共享处理成功；0 - 失败。
292 static int try_to_share(unsigned long address, struct task_struct * p)
293 {
294     unsigned long from;
295     unsigned long to;
296     unsigned long from_page;
297     unsigned long to_page;
298     unsigned long phys_addr;
299
// 首先分别求得指定进程 p 中和当前进程中逻辑地址 address 对应的页目录项。为了计算方便
// 先求出指定逻辑地址 address 处的'逻辑'页目录项号，即以进程空间 (0 - 64MB) 算出的页
// 目录项号。该'逻辑'页目录项号加上进程 p 在 CPU 4G 线性空间中起始地址对应的页目录项，
// 即得到进程 p 中地址 address 处页面所对应的 4G 线性空间中的实际页目录项 from_page。
// 而'逻辑'页目录项号加上当前进程 CPU 4G 线性空间中起始地址对应的页目录项，即可最后
// 得到当前进程中地址 address 处页面所对应的 4G 线性空间中的实际页目录项 to_page。
300     from_page = to_page = ((address>>20) & 0xffc);
301     from_page += ((p->start_code>>20) & 0xffc);    // p 进程目录项。

```



```

302         to_page += ((current->start_code>>20) & 0xffc); // 当前进程目录项。

// 在得到 p 进程和当前进程 address 对应的目录项后，下面分别对进程 p 和当前进程进行处理。
// 下面首先对 p 进程的表项进行操作。目标是取得 p 进程中 address 对应的物理内存页面地址，
// 并且该物理页面存在，而且干净（没有被修改过，不脏）。
// 方法是先取目录项内容。如果该目录项无效（P=0），表示目录项对应的二级页表不存在，
// 于是返回。否则取该目录项对应页表地址 from，从而计算出逻辑地址 address 对应的页表项
// 指针，并取出该页表项内容临时保存在 phys_addr 中。
303 /* is there a page-directory at from? */
/* 在 from 处是否存在页目录项? */
304         from = *(unsigned long *) from_page; // p 进程目录项内容。
305         if (!(from & 1))
306             return 0;
307         from &= 0xfffff000; // 页表指针（地址）。
308         from_page = from + ((address>>10) & 0xffc); // 页表项指针。
309         phys_addr = *(unsigned long *) from_page; // 页表项内容。
310 /* is the page clean and present? */
/* 物理页面干净并且存在吗? */
// 接着看看页表项映射的物理页面是否存在并且干净。0x41 对应页表项中的 D (Dirty) 和
// P (Present) 标志。如果页面不干净或无效则返回。然后我们从该表项中取出物理页面地址
// 再保存在 phys_addr 中。最后我们再检查一下这个物理页面地址的有效性，即它不应该超过
// 机器最大物理地址值，也不应该小于内存低端(1MB)。
311         if ((phys_addr & 0x41) != 0x01)
312             return 0;
313         phys_addr &= 0xfffff000; // 物理页面地址。
314         if (phys_addr >= HIGH_MEMORY || phys_addr < LOW_MEM)
315             return 0;

// 下面首先对当前进程的表项进行操作。目标是取得当前进程中 address 对应的页表项地址，
// 并且该页表项还没有映射物理页面，即其 P=0。
// 首先取当前进程页目录项内容→to。如果该目录项无效（P=0），即目录项对应的二级页表
// 不存在，则申请一空闲页面来存放页表，并更新目录项 to_page 内容，让其指向该内存页面。
316         to = *(unsigned long *) to_page; // 当前进程目录项内容。
317         if (!(to & 1))
318             if (to = get_free_page())
319                 *(unsigned long *) to_page = to | 7;
320         else
321             oom();
// 否则取目录项中的页表地址→to，加上页表项索引值<<2，即页表项在表中偏移地址，得到
// 页表项地址→to_page。针对该页表项，如果此时我们检查出其对应的物理页面已经存在，
// 即页表项的存在位 P=1，则说明原本我们想共享进程 p 中对应的物理页面，但现在我们自己
// 已经占有了（映射有）物理页面。于是说明内核出错，死机。
322         to &= 0xfffff000; // 页表地址。
323         to_page = to + ((address>>10) & 0xffc); // 页表项地址。
324         if (1 & *(unsigned long *) to_page)
325             panic("try_to_share: to_page already exists");

// 在找到了进程 p 中逻辑地址 address 处对应的干净且存在的物理页面，而且也确定了当前
// 进程中逻辑地址 address 所对应的二级页表项地址之后，我们现在对他们进行共享处理。
// 方法很简单，就是首先对 p 进程的页表项进行修改，设置其写保护（R/W=0，只读）标志，
// 然后让当前进程复制 p 进程的这个页表项。此时当前进程逻辑地址 address 处页面即被
// 映射到 p 进程逻辑地址 address 处页面映射的物理页面上。
326 /* share them: write-protect */

```

```

/* 对它们进行共享处理：写保护 */
327     *(unsigned long *) from_page &= ~2;
328     *(unsigned long *) to_page = *(unsigned long *) from_page;
// 随后刷新页变换高速缓冲。计算所操作物理页面的页面号，并将对应页面映射字节数组项中
// 的引用递增 1。最后返回 1，表示共享处理成功。
329     invalidate();
330     phys_addr -= LOW_MEM;
331     phys_addr >>= 12; // 得页面号。
332     mem_map[phys_addr]++;
333     return 1;
334 }
335
336 /*
337  * share_page() tries to find a process that could share a page with
338  * the current one. Address is the address of the wanted page relative
339  * to the current data space.
340  *
341  * We first check if it is at all feasible by checking executable->i_count.
342  * It should be >1 if there are other tasks sharing this inode.
343  */
/*
 * share_page() 试图找到一个进程，它可以与当前进程共享页面。参数 address 是
 * 当前进程数据空间中期望共享的某页面地址。
 *
 * 首先我们通过检测 executable->i_count 来查证是否可行。如果有其他任务已共享
 * 该 inode，则它应该大于 1。
 */
///// 共享页面处理。
// 在发生缺页异常时，首先看看能否与运行同一个执行文件的其他进程进行页面共享处理。
// 该函数首先判断系统中是否有另一个进程也在运行当前进程一样的执行文件。若有，则在
// 系统当前所有任务中寻找这样的任务。若找到了这样的任务就尝试与其共享指定地址处的
// 页面。若系统中没有其他任务正在运行与当前进程相同的执行文件，那么共享页面操作的
// 前提条件不存在，因此函数立刻退出。判断系统中是否有另一个进程也在执行同一个执行
// 文件的方法是利用进程任务数据结构中的 executable 字段。该字段指向进程正在执行程
// 序在内存中的 i 节点。根据该 i 节点的引用次数 i_count 我们可以进行这种判断。若
// executable->i_count 值大于 1，则表明系统中可能有两个进程在运行同一个执行文件，
// 于是可以再对任务结构数组中所有任务比较是否有相同的 executable 字段来最后确定多
// 个进程运行着相同执行文件的情况。
// 参数 address 是进程中的逻辑地址，即是当前进程欲与 p 进程共享页面的逻辑页面地址。
// 返回 1 - 共享操作成功，0 - 失败。
344 static int share_page(unsigned long address)
345 {
346     struct task_struct ** p;
347
// 首先检查一下当前进程的 executable 字段是否指向某执行文件的 i 节点，以判断本进程
// 是否有对应的执行文件。如果没有，则返回 0。如果 executable 的确指向某个 i 节点，
// 则检查该 i 节点引用计数值。如果当前进程运行的执行文件的内存 i 节点引用计数等于
// 1 (executable->i_count == 1)，表示当前系统中只有 1 个进程（即当前进程）在运行该
// 执行文件。因此无共享可言，直接退出函数。
348     if (!current->executable)
349         return 0;
350     if (current->executable->i_count < 2)
351         return 0;

```

```

// 否则搜索任务数组中所有任务。寻找与当前进程可共享页面的进程，即运行相同执行文件
// 的另一个进程，并尝试对指定地址的页面进行共享。如果找到某个进程 p，其 executable
// 字段值与当前进程的相同，则调用 try_to_share() 尝试页面共享。若共享操作成功，则
// 函数返回 1。否则返回 0，表示共享页面操作失败。
352     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
353         if (!*p)                                // 如果该任务项空闲，则继续寻找。
354             continue;
355         if (current == *p)                        // 如果就是当前任务，也继续寻找。
356             continue;
// 如果 executable 不等，表示运行的不是与当前进程相同的执行文件，因此也继续寻找。
357         if ((*p)->executable != current->executable)
358             continue;
359         if (try_to_share(address,*p))            // 尝试共享页面。
360             return 1;
361     }
362     return 0;
363 }
364
//// 执行缺页处理。
// 是访问不存在页面处理函数。页异常中断处理过程中调用的函数。在 page.s 程序中被调用。
// 函数参数 error_code 和 address 是进程在访问页面时由 CPU 因缺页产生异常而自动生成。
// error_code 指出出错类型，参见本章开始处的“内存页面出错异常”一节；address 是产生
// 异常的页面线性地址。
// 该函数首先尝试与已加载的相同文件进行页面共享，或者只是由于进程动态申请内存页面而
// 只需映射一页物理内存页即可。若共享操作不成功，那么只能从相应文件中读入所缺的数据
// 页面到指定线性地址处。
365 void do_no_page(unsigned long error_code,unsigned long address)
366 {
367     int nr[4];
368     unsigned long tmp;
369     unsigned long page;
370     int block,i;
371
// 首先取线性空间中指定地址 address 处页面地址。从而可算出指定线性地址在进程空间中
// 相对于进程基址的偏移长度值 tmp，即对应的逻辑地址。
372     address &= 0xfffff000;                        // address 处缺页页面地址。
373     tmp = address - current->start_code;          // 缺页页面对应逻辑地址。
// 若当前进程的 executable 节点指针空，或者指定地址超出（代码 + 数据）长度，则申请
// 一页物理内存，并映射到指定的线性地址处。executable 是进程正在运行的执行文件的 i
// 节点结构。由于任务 0 和任务 1 的代码在内核中，因此任务 0、任务 1 以及任务 1 派生的
// 没有调用过 execve() 的所有任务的 executable 都为 0。若该值为 0，或者参数指定的线性
// 地址超出代码加数据长度，则表明进程在申请新的内存页面存放堆或栈中数据。因此直接
// 调用取空闲页面函数 get_empty_page() 为进程申请一页物理内存并映射到指定线性地址
// 处。进程任务结构 字段 start_code 是线性地址空间中进程代码段地址，字段 end_data
// 是代码加数据长度。对于 Linux 0.11 内核，它的代码段和数据段起始基址相同。
374     if (!current->executable || tmp >= current->end_data) {
375         get_empty_page(address);
376         return;
377     }
// 否则说明所缺页面在进程执行影像文件范围内，于是就尝试共享页面操作，若成功则退出。
// 若不成功就只能申请一页物理内存页面 page，然后从设备上读取执行文件中的相应页面并
// 放置（映射）到进程页面逻辑地址 tmp 处。
378     if (share_page(tmp))                        // 尝试逻辑地址 tmp 处页面的共享。

```

```

379         return;
380         if (!(page = get_free_page()))           // 申请一页物理内存。
381             oom();
382 /* remember that 1 block is used for header */
383 /* 记住，（程序）头要使用 1 个数据块 */
384 // 因为块设备上存放的执行文件映像第 1 块数据是程序头结构，因此在读取该文件时需要跳过
385 // 第 1 块数据。所以需要首先计算缺页所在的数据块号。因为每块数据长度为 BLOCK_SIZE =
386 // 1KB，因此一页内存可存放 4 个数据块。进程逻辑地址 tmp 除以数据块大小再加 1 即可得出
387 // 缺少的页面在执行映像文件中的起始块号 block。根据这个块号和执行文件的 i 节点，我们
388 // 就可以从映射位图中找到对应块设备中对应的设备逻辑块号（保存在 nr[] 数组中）。利用
389 // bread_page() 即可把这 4 个逻辑块读入到物理页面 page 中。
390         block = 1 + tmp/BLOCK_SIZE;             // 执行文件中起始数据块号。
391         for (i=0 ; i<4 ; block++, i++)
392             nr[i] = bmap(current->executable, block); // 设备上对应的逻辑块号。
393         bread_page(page, current->executable->i_dev, nr); // 读设备上 4 个逻辑块。

394 // 在读设备逻辑块操作时，可能会出现这样一种情况，即在执行文件中的读取页面位置可能离
395 // 文件尾不到 1 个页面的长度。因此就可能读入一些无用的信息。下面的操作就是把这部分超
396 // 出执行文件 end_data 以后的部分清零处理。
397         i = tmp + 4096 - current->end_data;      // 超出的字节长度值。
398         tmp = page + 4096;                       // tmp 指向页面末端。
399         while (i-- > 0) {                         // 页面末端 i 字节清零。
400             tmp--;
401             *(char *)tmp = 0;
402         }
403 // 最后把引起缺页异常的一页物理页面映射到指定线性地址 address 处。若操作成功就返回。
404 // 否则就释放内存页，显示内存不够。
405         if (put_page(page, address))
406             return;
407         free_page(page);
408         oom();
409     }
410 }
411
412 //// 物理内存管理初始化。
413 // 该函数对 1MB 以上内存区域以页面为单位进行管理前的初始化设置工作。一个页面长度为
414 // 4KB 字节。该函数把 1MB 以上所有物理内存划分成一个个页面，并使用一个页面映射字节
415 // 数组 mem_map[] 来管理所有这些页面。对于具有 16MB 内存容量的机器，该数组共有 3840
416 // 项 ((16MB - 1MB)/4KB)，即可管理 3840 个物理页面。每当一个物理内存页面被占用时就
417 // 把 mem_map[] 中对应的的字节值增 1；若释放一个物理页面，就把对应字节值减 1。若字
418 // 节值为 0，则表示对应页面空闲；若字节值大于或等于 1，则表示对应页面被占用或被不
419 // 同程序共享占用。
420 // 在该版本的 Linux 内核中，最多能管理 16MB 的物理内存，大于 16MB 的内存将弃置不用。
421 // 对于具有 16MB 内存的 PC 机系统，在没有设置虚拟盘 RAMDISK 的情况下 start_mem 通常
422 // 是 4MB，end_mem 是 16MB。因此此时主内存区范围是 4MB—16MB，共有 3072 个物理页面可
423 // 供分配。而范围 0 - 1MB 内存空间用于内核系统（其实内核只使用 0 —640Kb，剩下的部
424 // 分被部分高速缓冲和设备内存占用）。
425 // 参数 start_mem 是用作页面分配的主内存区起始地址（已去除 RAMDISK 所占内存空间）。
426 // end_mem 是实际物理内存最大地址。而地址范围 start_mem 到 end_mem 是主内存区。
427 void mem_init(long start_mem, long end_mem)
428 {
429     int i;
430
431     // 首先将 1MB 到 16MB 范围内所有内存页面对应的内存映射字节数组项置为已占用状态，即各

```

```

// 项字节值全部设置成 USED (100)。PAGING_PAGES 被定义为(PAGING_MEMORY>>12)，即 1MB
// 以上所有物理内存分页后的内存页面数(15MB/4KB = 3840)。
403     HIGH_MEMORY = end_mem;                // 设置内存最高端 (16MB)。
404     for (i=0 ; i<PAGING_PAGES ; i++)
405         mem_map[i] = USED;
// 然后计算主内存区起始内存 start_mem 处页面对应内存映射字节数组中项号 i 和主内存区
// 页面数。此时 mem_map[] 数组的第 i 项正对应主内存区中第 1 个页面。最后将主内存区中
// 页面对应的数组项清零 (表示空闲)。对于具有 16MB 物理内存的系统，mem_map[] 中对应
// 4Mb--16Mb 主内存区的项被清零。
406     i = MAP_NR(start_mem);                // 主内存区起始位置处页面号。
407     end_mem -= start_mem;
408     end_mem >>= 12;                        // 主内存区中的总页面数。
409     while (end_mem-->0)
410         mem_map[i++]=0;                    // 主内存区页面对应字节值清零。
411 }
412
//// 计算内存空闲页面数并显示。
// [?? 内核中没有地方调用该函数，Linux 调试过程中用的 ]
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。然后扫描所有页目录项 (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。页目录项 0—3 被内核
// 使用，因此应该从第 5 个目录项 (i=4) 开始扫描。
418     for(i=0 ; i<PAGING_PAGES ; i++)
419         if (!mem_map[i]) free++;
420     printk("%d pages free (of %d)\n", free, PAGING_PAGES);
421     for(i=2 ; i<1024 ; i++) {
422         if (l&pg_dir[i]) {
423             pg_tbl=(long *) (0xffffffff & 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

```

13.4 page.s 程序

13.4.1 功能描述

该文件包括页异常中断处理程序 (中断 14)，主要分两种情况处理。一是由于缺页引起的页异常中断，通过调用 `do_no_page(error_code, address)` 来处理；二是由页写保护引起的页异常，此时调用页写保护处理函数 `do_wp_page(error_code, address)` 进行处理。其中的出错码(`error_code`)是由 CPU 自动产生并压入堆栈的，出现异常时访问的线性地址是从控制寄存器 CR2 中取得的。CR2 是专门用来存放页出错时的线性地址。

13.4.2 代码注释

程序 13-3 linux/mm/page.s

```

1  /*
2  *  linux/mm/page.s
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  page.s contains the low-level page-exception code.
9  *  the real work is done in mm.c
10 */
11 /*
12  *  page.s 程序包含底层页异常处理代码。实际工作在 memory.c 中完成。
13  */
14 .globl _page_fault          # 声明为全局变量。将在 traps.c 中用于设置页异常描述符。
15 _page_fault:
16     xchgl %eax, (%esp)      # 取出错码到 eax。
17     pushl %ecx
18     pushl %edx
19     push %ds
20     push %es
21     push %fs
22     movl $0x10, %edx       # 置内核数据段选择符。
23     mov %dx, %ds
24     mov %dx, %es
25     mov %dx, %fs
26     movl %cr2, %edx        # 取引起页面异常的线性地址。
27     pushl %edx             # 将该线性地址和出错码压入栈中，作为将调用函数的参数。
28     pushl %eax
29     testl $1, %eax         # 测试页存在标志 P（位 0），如果不是缺页引起的异常则跳转。
30     jne 1f
31     call _do_no_page       # 调用缺页处理函数（mm/memory.c, 365 行）。
32 1:    call _do_wp_page     # 调用写保护处理函数（mm/memory.c, 247 行）。
33 2:    addl $8, %esp        # 丢弃压入栈的两个参数，弹出栈中寄存器并退出中断。
34     pop %fs
35     pop %es
36     pop %ds
37     popl %edx
38     popl %ecx
39     popl %eax
40     iret

```

13.4.3 其他信息

13.4.3.1 页出错异常处理

当处理器在转换线性地址到物理地址的过程中检测到以下两种条件时，就会发生页异常中断，中断

14。

- o 当 CPU 发现对应页目录项或页表项的存在位（Present）标志为 0。
- o 当前进程没有访问指定页面的权限。

对于页异常处理中断，CPU 提供了两项信息用来诊断页异常和从中恢复运行。

- (1) 放在堆栈上的出错码。该出错码指出了异常是由于页不存在引起的还是违反了访问权限引起的；在发生异常时 CPU 的当前特权层；以及是读操作还是写操作。出错码的格式是一个 32 位的长字。但只用了最后的 3 个比特位。分别说明导致异常发生时的原因：
 - 位 2(U/S) - 0 表示在超级用户模式下执行，1 表示在用户模式下执行；
 - 位 1(W/R) - 0 表示读操作，1 表示写操作；
 - 位 0(P) - 0 表示页不存在，1 表示页级保护。
- (2) CR2(控制寄存器 2)。CPU 将造成异常的用于访问的线性地址存放在 CR2 中。异常处理程序可以使用这个地址来定位相应的页目录和页表项。如果在页异常处理程序执行期间允许发生另一个页异常，那么处理程序应该将 CR2 压入堆栈中。

