














第 12 章 库文件(lib)

12.1 概述

	Name	Size	Last modified (GMT)	Description
	Makefile	2602 bytes	1991-12-02 03:16:05	
	_exit.c	198 bytes	1991-10-02 14:16:29	
	close.c	131 bytes	1991-10-02 14:16:29	
	ctype.c	1202 bytes	1991-10-02 14:16:29	
	dup.c	127 bytes	1991-10-02 14:16:29	
	errno.c	73 bytes	1991-10-02 14:16:29	
	execve.c	170 bytes	1991-10-02 14:16:29	
	malloc.c	7469 bytes	1991-12-02 03:15:20	
	open.c	389 bytes	1991-10-02 14:16:29	
	setsid.c	128 bytes	1991-10-02 14:16:29	
	string.c	177 bytes	1991-10-02 14:16:29	
	wait.c	253 bytes	1991-10-02 14:16:29	
	write.c	160 bytes	1991-10-02 14:16:29	

12.2 Makefile 文件

12.2.1 功能描述

12.2.2 代码注释

列表 linux/lib/Makefile 文件

```

1 #
2 # Makefile for some libs needed in the kernel.
3 #
4 # Note! Dependencies are done automagically by 'make dep', which also
5 # removes any old dependencies. DON'T put your own dependencies here
6 # unless it's something special (ie not a .c file).
7 #
  # 内核需要用到的 libs 库文件程序的 Makefile。
  #
  # 注意！依赖关系是由 'make dep' 自动进行的，它也会自动去除原来的依赖信息。不要把你自己的

```

依赖关系信息放在这里，除非是特别文件的（也即不是一个.c 文件的信息）。

```

8
9 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。
10 AS      =gas      # GNU 的汇编程序。
11 LD      =gld      # GNU 的连接程序。
12 LDFLAGS =-s -x    # 连接程序所有的参数，-s 输出文件中省略所有符号信息。-x 删除所有局部符号。
13 CC      =gcc      # GNU C 语言编译器。
14 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
15         -finline-functions -mstring-insns -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；-mstring-insns Linus 自己添加的优化选项，以后不再使用；
# -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)。

16 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。

17
# 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。

18 .c.s:
19     $(CC) $(CFLAGS) \
20     -S -o $.s $<
# 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。

21 .s.o:
22     $(AS) -c -o $.o $<
23 .c.o:                                # 类似上面，*.c 文件→*.o 目标文件。不进行连接。
24     $(CC) $(CFLAGS) \
25     -c -o $.o $<
26
# 下面定义目标文件变量 OBJS。
27 OBJS = ctype.o _exit.o open.o close.o errno.o write.o dup.o setsid.o \
28     execve.o wait.o string.o malloc.o
29
# 在有了先决条件 OBJS 后使用下面的命令连接成目标 lib.a 库文件。
30 lib.a: $(OBJS)
31     $(AR) rcs lib.a $(OBJS)
32     sync
33
# 下面的规则用于清理工作。当执行'make clean'时，就会执行下面的命令，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。
34 clean:
35     rm -f core *.o *.a tmp_make
36     for i in *.c;do rm -f `basename $$i .c`.s;done
37
# 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（即是本文件）进行处理，输出为删除 Makefile
# 文件中'### Dependencies'行后面的所有行（下面从 45 开始的行），并生成 tmp_make
# 临时文件（39 行的作用）。然后对 kernel/blk_drv/目录下的每个 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。

```

38 dep:
39     sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
40     (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,``" "; \
41         $(CPP) -M $$i;done) >> tmp_make
42     cp tmp_make Makefile
43
44 ### Dependencies:
45 _exit.s _exit.o : _exit.c ../include/unistd.h ../include/sys/stat.h \
46     ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
47     ../include/utime.h
48 close.s close.o : close.c ../include/unistd.h ../include/sys/stat.h \
49     ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
50     ../include/utime.h
51 ctype.s ctype.o : ctype.c ../include/ctype.h
52 dup.s dup.o : dup.c ../include/unistd.h ../include/sys/stat.h \
53     ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
54     ../include/utime.h
55 errno.s errno.o : errno.c
56 execve.s execve.o : execve.c ../include/unistd.h ../include/sys/stat.h \
57     ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
58     ../include/utime.h
59 malloc.s malloc.o : malloc.c ../include/linux/kernel.h ../include/linux/mm.h \
60     ../include/asm/system.h
61 open.s open.o : open.c ../include/unistd.h ../include/sys/stat.h \
62     ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
63     ../include/utime.h ../include/stdarg.h
64 setuid.s setuid.o : setuid.c ../include/unistd.h ../include/sys/stat.h \
65     ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
66     ../include/utime.h
67 string.s string.o : string.c ../include/string.h
68 wait.s wait.o : wait.c ../include/unistd.h ../include/sys/stat.h \
69     ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
70     ../include/utime.h ../include/sys/wait.h
71 write.s write.o : write.c ../include/unistd.h ../include/sys/stat.h \
72     ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
73     ../include/utime.h

```

12.3 _exit.c 程序

12.3.1 功能描述

12.3.2 代码注释

列表 linux/lib/_exit.c 程序

```

1 /*
2  * linux/lib/_exit.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 #define LIBRARY          // 定义一个符号常量，见下行说明。
8 #include <unistd.h>      // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                          // 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()
等。
9
10 // 内核使用的程序(退出)终止函数。
11 // 直接调用系统中断 int 0x80，功能号__NR_exit。
12 // 参数: exit_code - 退出码。
13 volatile void _exit(int exit_code)
14 {
15     // %0 - eax(系统调用号__NR_exit); %1 - ebx(退出码 exit_code)。
16     __asm__ ("int $0x80"::"a" (__NR_exit), "b" (exit_code));
17 }
18

```

12.3.3 相关信息

参见 include/unistd.h 中的说明。

12.4 close.c 程序

12.4.1 功能描述

12.4.2 代码注释

列表 linux/lib/close.c 程序

```

1 /*
2  * linux/lib/close.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 #define LIBRARY
8 #include <unistd.h>      // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                          // 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()
等。
9
10 // 关闭文件函数。
11 // 下面该调用宏函数对应: int close(int fd)。直接调用了系统中断 int 0x80，参数是__NR_close。
12 // 其中 fd 是文件描述符。
13 _syscall1(int, close, int, fd)
14

```

12.5 ctype.c 程序

12.5.1 功能描述

12.5.2 代码注释

列表 linux/lib/ctype.c 程序

```

1  /*
2  *  linux/lib/ctype.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  #include <ctype.h>          // 字符类型头文件。定义了一些有关字符类型判断和转换的宏。
8
9  char _ctmp;                // 一个临时字符变量，供 ctype.h 文件中转换字符宏函数使用。
10 // 字符特性数组(表)，定义了各个字符对应的属性，这些属性类型(如_C 等)在 ctype.h 中定义。
11 // 用于判断字符是控制字符(_C)、大写字母(_U)、小写字母(_L)等所属类型。
12 unsigned char _ctype[] = {0x00,          /* EOF */
13   _C, _C, _C, _C, _C, _C, _C, _C,      /* 0-7 */
14   _C, _C|_S, _C|_S, _C|_S, _C|_S, _C|_S, _C, _C, /* 8-15 */
15   _C, _C, _C, _C, _C, _C, _C, _C,      /* 16-23 */
16   _C, _C, _C, _C, _C, _C, _C, _C,      /* 24-31 */
17   _S|_SP, _P, _P, _P, _P, _P, _P, _P, /* 32-39 */
18   _P, _P, _P, _P, _P, _P, _P, _P,      /* 40-47 */
19   _D, _D, _D, _D, _D, _D, _D, _D,      /* 48-55 */
20   _D, _D, _P, _P, _P, _P, _P, _P,      /* 56-63 */
21   _P, _U|_X, _U|_X, _U|_X, _U|_X, _U|_X, _U|_X, _U, /* 64-71 */
22   _U, _U, _U, _U, _U, _U, _U, _U,      /* 72-79 */
23   _U, _U, _U, _U, _U, _U, _U, _U,      /* 80-87 */
24   _U, _U, _P, _P, _P, _P, _P, _P,      /* 88-95 */
25   _P, _L|_X, _L|_X, _L|_X, _L|_X, _L|_X, _L|_X, _L, /* 96-103 */
26   _L, _L, _L, _L, _L, _L, _L, _L,      /* 104-111 */
27   _L, _L, _L, _L, _L, _L, _L, _L,      /* 112-119 */
28   _L, _L, _L, _P, _P, _P, _P, _C,      /* 120-127 */
29   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 128-143 */
30   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 144-159 */
31   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 160-175 */
32   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 176-191 */
33   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 192-207 */
34   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 208-223 */
35   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, /* 224-239 */
36   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0; /* 240-255 */

```

12.6 dup.c 程序

12.6.1 功能描述

该程序包括一个创建文件描述符拷贝的函数 `dup()`。在成功返回之后，新的和原来的描述符可以交替使用。它们共享锁定、文件读写指针以及文件标志。例如，如果文件读写位置指针被其中一个描述符使用 `lseek()` 修改过之后，则对于另一个描述符来讲，文件读写指针也被改变。该函数使用数值最小的未使用描述符来建立新描述符。但是这两个描述符并不共享执行时关闭标志(`close-on-exec`)。

12.6.2 代码注释

列表 linux/lib/dup.c 程序

```

1  /*
2  *  linux/lib/dup.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  #define  LIBRARY
8  #include <unistd.h>          // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                                // 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()
等。
9
   /// 复制文件描述符函数。
   // 下面该调用宏函数对应: int dup(int fd)。直接调用了系统中断 int 0x80，参数是__NR_dup。
   // 其中 fd 是文件描述符。
10 syscall1(int, dup, int, fd)
11

```

12.7 errno.c 程序

12.7.1 功能描述

该程序仅定义了一个出错号变量 `errno`。用于在函数调用失败时存放出错号。

12.7.2 代码注释

列表 linux/lib/errno.c 程序

```

1  /*
2  *  linux/lib/errno.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  int errno;

```

12.8 execve.c 程序

12.8.1 功能描述

12.8.2 代码注释

列表 linux/lib/execve.c 程序

```

1  /*
2  *  linux/lib/execve.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  #define  LIBRARY
8  #include <unistd.h>          // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                                // 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()
等。
9
    ///// 加载并执行子进程(其它程序)函数。
    // 下面该调用宏函数对应: int execve(const char * file, char ** argv, char ** envp)。
    // 参数: file - 被执行程序文件名; argv - 命令行参数指针数组; envp - 环境变量指针数组。
    // 直接调用了系统中断 int 0x80, 参数是__NR_execve。参见 include/unistd.h 和 fs/exec.c 程序。
10 syscall13(int, execve, const char *, file, char **, argv, char **, envp)
11

```

12.9 malloc.c 程序

12.9.1 功能描述

该程序中主要包括内存分配函数 malloc()。该函数使用了存储桶(bucket)的原理对分配的内存进行管理。基本思想是对不同请求的内存块大小(长度)，使用存储桶目录(下面简称目录)分别进行处理。比如对于请求内存块的长度在 32 字节或 32 字节以下但大于 16 字节时，就使用存储桶目录第二项对应的存储桶描述符链表分配内存块。其基本结构示意图见下图所示。该函数目前一次所能分配的最大内存长度是一个内存页面，即 4096 字节。

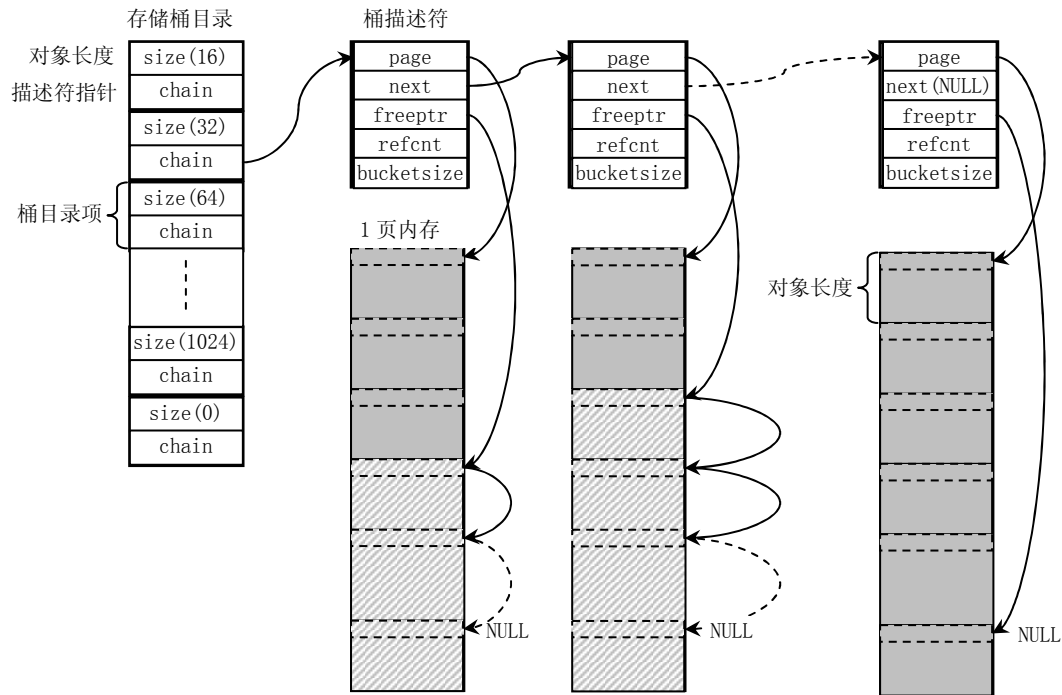


图 使用存储桶原理进行内存分配管理的结构示意图

在第一次调用 `malloc()` 函数时，首先要建立一个页面的空闲存储桶描述符(下面简称描述符)链表，其中存放着还未使用或已经使用完毕而收回的描述符。该链表结构示意图见下图所示。其中 `free_bucket_desc` 是链表头指针。从链表中取出或放入一个描述符都是从链表头开始操作。当取出一个描述符时，就将链表头指针所指向的头一个描述符取出；当释放一个空闲描述符时也是将其放在链表头处。

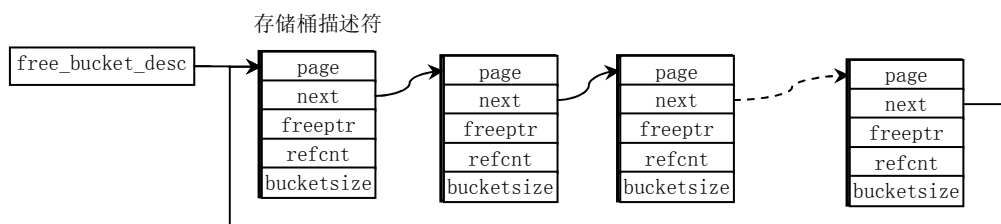


图 空闲存储桶描述符链表结构示意图

`malloc()` 函数执行的基本步骤如下：

1. 首先搜索目录，寻找适合请求内存块大小的目录项对应的描述符链表。当目录项的对象字节长度大于请求的字节长度，就算找到了相应的目录项。如果搜索完整个目录都没有找到合适的目录项，则说明用户请求的内存块太大。
2. 在目录项对应的描述符链表中查找具有空闲空间的描述符。如果某个描述符的空闲内存指针 `freeptr` 不为 `NULL`，则表示找到了相应的描述符。如果没有找到具有空闲空间的描述符，那么我们就需要新建一个描述符。新建描述符的过程如下：
 - a. 如果空闲描述符链表头指针还是 `NULL` 的话，说明是第一次调用 `malloc()` 函数，此时需要 `init_bucket_desc()` 来创建空闲描述符链表。

- b. 然后从空闲描述符链表头处取一个描述符，初始化该描述符，令其对象引用计数为 0，对象大小等于对应目录项指定对象的长度值，并申请一内存页面，让描述符的页面指针 page 指向该内存页，描述符的空闲内存指针 freeptr 也指向页开始位置。
 - c. 对该内存页面根据本目录项所用对象长度进行页面初始化，建立所有对象的一个链表。也即每个对象的头部都存放一个指向下一个对象的指针，最后一个对象的开始处存放一个 NULL 指针值。
 - d. 然后将该描述符插入到对应目录项的描述符链表开始处。
3. 将该描述符的空闲内存指针 freeptr 复制为返回给用户的内存指针，然后调整该 freeptr 指向描述符对应内存页面中下一个空闲对象位置，并使该描述符引用计数值增 1。

free_s() 函数用于回收用户释放的内存块。基本原理是首先根据该内存块的地址换算出该内存块对应页面的地址(用页面长度进行模运算)，然后搜索目录中的所有描述符，找到对应该页面的描述符。将该释放的内存块链入 freeptr 所指向的空闲对象链表中，并将描述符的对象引用计数值减 1。如果引用计数值此时等于零，则表示该描述符对应的页面已经完全空出，可以释放该内存页面并将该描述符收回到空闲描述符链表中。

12.9.2 代码注释

列表 linux/lib/malloc.c 程序

```

1 /*
2  * malloc.c --- a general purpose kernel memory allocator for Linux.
3  *
4  * Written by Theodore Ts'o (tytso@mit.edu), 11/29/91
5  *
6  * This routine is written to be as fast as possible, so that it
7  * can be called from the interrupt level.
8  *
9  * Limitations: maximum size of memory we can allocate using this routine
10 *      is 4k, the size of a page in Linux.
11 *
12 * The general game plan is that each page (called a bucket) will only hold
13 * objects of a given size.  When all of the object on a page are released,
14 * the page can be returned to the general free pool.  When malloc() is
15 * called, it looks for the smallest bucket size which will fulfill its
16 * request, and allocate a piece of memory from that bucket pool.
17 *
18 * Each bucket has as its control block a bucket descriptor which keeps
19 * track of how many objects are in use on that page, and the free list
20 * for that page.  Like the buckets themselves, bucket descriptors are
21 * stored on pages requested from get_free_page().  However, unlike buckets,
22 * pages devoted to bucket descriptor pages are never released back to the
23 * system.  Fortunately, a system should probably only need 1 or 2 bucket
24 * descriptor pages, since a page can hold 256 bucket descriptors (which
25 * corresponds to 1 megabyte worth of bucket pages.)  If the kernel is using
26 * that much allocated memory, it's probably doing something wrong.  :-)
27 *
28 * Note: malloc() and free() both call get_free_page() and free_page()
29 *      in sections of code where interrupts are turned off, to allow
30 *      malloc() and free() to be safely called from an interrupt routine.

```

```

31 *      (We will probably need this functionality when networking code,
32 *      particularly things like NFS, is added to Linux.) However, this
33 *      presumes that get_free_page() and free_page() are interrupt-level
34 *      safe, which they may not be once paging is added. If this is the
35 *      case, we will need to modify malloc() to keep a few unused pages
36 *      "pre-allocated" so that it can safely draw upon those pages if
37 *      it is called from an interrupt routine.
38 *
39 *      Another concern is that get_free_page() should not sleep; if it
40 *      does, the code is carefully ordered so as to avoid any race
41 *      conditions. The catch is that if malloc() is called re-entrantly,
42 *      there is a chance that unnecessary pages will be grabbed from the
43 *      system. Except for the pages for the bucket descriptor page, the
44 *      extra pages will eventually get released back to the system, though,
45 *      so it isn't all that bad.
46 */
47
/*
 *      malloc.c - Linux 的通用内核内存分配函数。
 *
 *      由 Theodore Ts'o 编制 (tytso@mit.edu), 11/29/91
 *
 *      该函数被编写成尽可能地快，从而可以从中断层调用此函数。
 *
 *      限制：使用该函数一次所能分配的最大内存是 4k，也即 Linux 中内存页面的大小。
 *
 *      编写该函数所遵循的一般规则是每页（被称为一个存储桶）仅分配所要容纳对象的大小。
 *      当一页上的所有对象都释放后，该页就可以返回通用空闲内存池。当 malloc() 被调用
 *      时，它会寻找满足要求的最小的存储桶，并从该存储桶中分配一块内存。
 *
 *      每个存储桶都有一个作为其控制用的存储桶描述符，其中记录了页面上有多少对象正被
 *      使用以及该页上空闲内存的列表。就象存储桶自身一样，存储桶描述符也是存储在使用
 *      get_free_page() 申请到的页面上的，但是与存储桶不同的是，桶描述符所占用的页面
 *      将不再会释放给系统。幸运的是一个系统大约只需要 1 到 2 页的桶描述符页面，因为一
 *      个页面可以存放 256 个桶描述符（对应 1MB 内存的存储桶页面）。如果系统为桶描述符分
 *      配了许多内存，那么肯定系统什么地方出了问题☺。
 *
 *      注意！malloc() 和 free() 两者关闭了中断的代码部分都调用了 get_free_page() 和
 *      free_page() 函数，以使 malloc() 和 free() 可以安全地被从中断程序中调用
 *      （当网络代码，尤其是 NFS 等被加入到 Linux 中时就可能需要这种功能）。但前
 *      提是假设 get_free_page() 和 free_page() 是可以安全地在中断级程序中使用的，
 *      这在一旦加入了分页处理之后就可能不是安全的。如果真是这种情况，那么我们就
 *      需要修改 malloc() 来“预先分配”几页不用的内存，如果 malloc() 和 free()
 *      被从中断程序中调用时就可以安全地使用这些页面。
 *
 *      另外需要考虑到的是 get_free_page() 不应该睡眠；如果会睡眠的话，则为了防止
 *      任何竞争条件，代码需要仔细地安排顺序。关键在于如果 malloc() 是可以重入地
 *      被调用的话，那么就会存在不必要的页面被从系统中取走的机会。除了用于桶描述
 *      符的页面，这些额外的页面最终会释放给系统，所以并不是象想象的那样不好。
 */
48 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
49 #include <linux/mm.h>     // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。

```

```

50 #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。
51 // 存储桶描述符结构。
52 struct bucket_desc { /* 16 bytes */
53     void *page; // 该桶描述符对应的内存页面指针。
54     struct bucket_desc *next; // 下一个描述符指针。
55     void *freeptr; // 指向本桶中空闲内存位置的指针。
56     unsigned short refcnt; // 引用计数。
57     unsigned short bucket_size; // 本描述符对应存储桶的大小。
58 };
59 // 存储桶描述符目录结构。
60 struct bucket_dir { /* 8 bytes */
61     int size; // 该存储桶的大小(字节数)。
62     struct bucket_desc *chain; // 该存储桶目录项的桶描述符链表指针。
63 };
64 /*
65  * The following is the where we store a pointer to the first bucket
66  * descriptor for a given size.
67  *
68  * If it turns out that the Linux kernel allocates a lot of objects of a
69  * specific size, then we may want to add that specific size to this list,
70  * since that will allow the memory to be allocated more efficiently.
71  * However, since an entire page must be dedicated to each specific size
72  * on this list, some amount of temperance must be exercised here.
73  *
74  * Note that this list must be kept in order.
75  */
76 /*
77  * 下面是我们存放第一个给定大小存储桶描述符指针的地方。
78  *
79  * 如果 Linux 内核分配了许多指定大小的对象，那么我们就希望将该指定的大小加到
80  * 该列表(链表)中，因为这样可以使内存的分配更有效。但是，因为一页完整内存页面
81  * 必须用于列表中指定大小的所有对象，所以需要总做总数方面的测试操作。
82  */
83 // 存储桶目录列表(数组)。
84 struct bucket_dir bucket_dir[] = {
85     { 16, (struct bucket_desc *) 0}, // 16 字节长度的内存块。
86     { 32, (struct bucket_desc *) 0}, // 32 字节长度的内存块。
87     { 64, (struct bucket_desc *) 0}, // 64 字节长度的内存块。
88     { 128, (struct bucket_desc *) 0}, // 128 字节长度的内存块。
89     { 256, (struct bucket_desc *) 0}, // 256 字节长度的内存块。
90     { 512, (struct bucket_desc *) 0}, // 512 字节长度的内存块。
91     { 1024, (struct bucket_desc *) 0}, // 1024 字节长度的内存块。
92     { 2048, (struct bucket_desc *) 0}, // 2048 字节长度的内存块。
93     { 4096, (struct bucket_desc *) 0}, // 4096 字节(1 页)内存。
94     { 0, (struct bucket_desc *) 0}}; /* End of list marker */
95 /*
96  * This contains a linked list of free bucket descriptor blocks
97  */
98 /*

```

```

    * 下面是含有空闲桶描述符内存块的链表。
    */
92 struct bucket\_desc *free\_bucket\_desc = (struct bucket\_desc *) 0;
93
94 /*
95  * This routine initializes a bucket description page.
96  */
    /*
    * 下面的子程序用于初始化一页桶描述符页面。
    */
    /*/// 初始化桶描述符。
    // 建立空闲桶描述符链表，并让 free_bucket_desc 指向第一个空闲桶描述符。
97 static inline void init\_bucket\_desc()
98 {
99     struct bucket\_desc *bdesc, *first;
100     int i;
101
    // 申请一页内存，用于存放桶描述符。如果失败，则显示初始化桶描述符时内存不够出错信息，死机。
102     first = bdesc = (struct bucket\_desc *) get\_free\_page();
103     if (!bdesc)
104         panic("Out of memory in init_bucket_desc()");
    // 首先计算一页内存中可存放的桶描述符数量，然后对其建立单向连接指针。
105     for (i = PAGE\_SIZE/sizeof(struct bucket\_desc); i > 1; i--) {
106         bdesc->next = bdesc+1;
107         bdesc++;
108     }
109     /*
110     * This is done last, to avoid race conditions in case
111     * get_free_page() sleeps and this routine gets called again....
112     */
    /*
    * 这是在最后处理的，目的是为了避免在 get_free_page() 睡眠时该子程序又被
    * 调用而引起的竞争条件。
    */
    // 将空闲桶描述符指针 free_bucket_desc 加入链表中。
113     bdesc->next = free\_bucket\_desc;
114     free\_bucket\_desc = first;
115 }
116
    /*/// 分配动态内存函数。
    // 参数: len - 请求的内存块长度。
    // 返回: 指向被分配内存的指针。如果失败则返回 NULL。
117 void *malloc(unsigned int len)
118 {
119     struct bucket\_dir *bdir;
120     struct bucket\_desc *bdesc;
121     void *retval;
122
123     /*
124     * First we search the bucket_dir to find the right bucket change
125     * for this request.
126     */
    /*

```

```

    * 首先我们搜索存储桶目录 bucket_dir 来寻找适合请求的桶大小。
    */
// 搜索存储桶目录，寻找适合申请内存块大小的桶描述符链表。如果目录项的桶字节数大于请求的字节
// 数，就找到了对应的桶目录项。
127     for (bdir = bucket\_dir; bdir->size; bdir++)
128         if (bdir->size >= len)
129             break;
// 如果搜索完整个目录都没有找到合适大小的目录项，则表明所请求的内存块大小太大，超出了该
// 程序的分配限制(最长为 1 个页面)。于是显示出错信息，死机。
130     if (!bdir->size) {
131         printk("malloc called with impossibly large argument (%d)\n",
132             len);
133         panic("malloc: bad arg");
134     }
135     /*
136     * Now we search for a bucket descriptor which has free space
137     */
    /*
    * 现在我们来搜索具有空闲空间的桶描述符。
    */
138     cli(); /* Avoid race conditions */ /* 为了避免出现竞争条件，首先关中断 */
// 搜索对应桶目录项中描述符链表，查找具有空闲空间的桶描述符。如果桶描述符的空闲内存指针
// freeptr 不为空，则表示找到了相应的桶描述符。
139     for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next)
140         if (bdesc->freeptr)
141             break;
142     /*
143     * If we didn't find a bucket with free space, then we'll
144     * allocate a new one.
145     */
    /*
    * 如果没有找到具有空闲空间的桶描述符，那么我们就需要新建一个该目录项的描述符。
    */
146     if (!bdesc) {
147         char      *cp;
148         int       i;
149
// 若 free_bucket_desc 还为空时，表示第一次调用该程序，则对描述符链表进行初始化。
// free_bucket_desc 指向第一个空闲桶描述符。
150         if (!free\_bucket\_desc)
151             init\_bucket\_desc();
// 取 free_bucket_desc 指向的空闲桶描述符，并让 free_bucket_desc 指向下一个空闲桶描述符。
152         bdesc = free\_bucket\_desc;
153         free\_bucket\_desc = bdesc->next;
// 初始化该新的桶描述符。令其引用数量等于 0；桶的大小等于对应桶目录的大小；申请一内存页面，
// 让描述符的页面指针 page 指向该页面；空闲内存指针也指向该页开头，因为此时全为空闲。
154         bdesc->refcnt = 0;
155         bdesc->bucket_size = bdir->size;
156         bdesc->page = bdesc->freeptr = (void *) cp = get\_free\_page();
// 如果申请内存页面操作失败，则显示出错信息，死机。
157         if (!cp)
158             panic("Out of memory in kernel malloc()");
159         /* Set up the chain of free objects */

```

```

/* 在该页空闲内存中建立空闲对象链表 */
// 以该桶目录项指定的桶大小为对象长度，对该页内存进行划分，并使每个对象的开始 4 字节设置
// 成指向下一对象的指针。
160     for (i=PAGE_SIZE/bdir->size; i > 1; i--) {
161         *((char **) cp) = cp + bdir->size;
162         cp += bdir->size;
163     }
// 最后一个对象开始处的指针设置为 0(NULL)。
// 然后让该桶描述符的下一描述符指针字段指向对应桶目录项指针 chain 所指的描述符，而桶目录的
// chain 指向该桶描述符，也即将该描述符插入到描述符链链头处。
164     *((char **) cp) = 0;
165     bdesc->next = bdir->chain; /* OK, link it in! */ /* OK, 将其链入! */
166     bdir->chain = bdesc;
167 }
// 返回指针即等于该描述符对应页面的当前空闲指针。然后调整该空闲空间指针指向下一个空闲对象，
// 并使描述符中对应页面中对象引用计数增 1。
168     retval = (void *) bdesc->freeptr;
169     bdesc->freeptr = *((void **) retval);
170     bdesc->refcnt++;
// 最后开放中断，并返回指向空闲内存对象的指针。
171     sti(); /* OK, we're safe again */ /* OK, 现在我们又安全了*/
172     return(retval);
173 }
174
175 /*
176  * Here is the free routine. If you know the size of the object that you
177  * are freeing, then free_s() will use that information to speed up the
178  * search for the bucket descriptor.
179  *
180  * We will #define a macro so that "free(x)" is becomes "free_s(x, 0)"
181  */
/*
* 下面是释放子程序。如果你知道释放对象的大小，则 free_s() 将使用该信息加速
* 搜寻对应桶描述符的速度。
*
* 我们将定义一个宏，使得"free(x)"成为"free_s(x, 0)"。
*/
///// 释放存储桶对象。
// 参数: obj - 对应对象指针; size - 大小。
182 void free_s(void *obj, int size)
183 {
184     void *page;
185     struct bucket_dir *bdir;
186     struct bucket_desc *bdesc, *prev;
187
188     /* Calculate what page this object lives in */
    /* 计算该对象所在的页面 */
189     page = (void *) ((unsigned long) obj & 0xfffff000);
190     /* Now search the buckets looking for that page */
    /* 现在搜索存储桶目录项所链接的桶描述符，寻找该页面 */
//
191     for (bdir = bucket_dir; bdir->size; bdir++) {
192         prev = 0;

```



```

193      /* If size is zero then this conditional is always false */
194      /* 如果参数 size 是 0，则下面条件肯定是 false */
195      if (bdir->size < size)
196          continue;
197      // 搜索对应目录项中链接的所有描述符，查找对应页面。如果某描述符页面指针等于 page 则表示找到
198      // 了相应的描述符，跳转到 found。如果描述符不含有对应 page，则让描述符指针 prev 指向该描述符。
199      for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next) {
200          if (bdesc->page == page)
201              goto found;
202          prev = bdesc;
203      }
204      // 若搜索了对应目录项的所有描述符都没有找到指定的页面，则显示出错信息，死机。
205      panic("Bad address passed to kernel free_s()");
206 found:
207      // 找到对应的桶描述符后，首先关中断。然后将该对象内存块链入空闲块对象链表中，并使该描述符
208      // 的对象引用计数减 1。
209      cli(); /* To avoid race conditions */ /* 为了避免竞争条件 */
210      *((void **)obj) = bdesc->freeptr;
211      bdesc->freeptr = obj;
212      bdesc->refcnt--;
213      // 如果引用计数已等于 0，则我们就可以释放对应的内存页面和该桶描述符。
214      if (bdesc->refcnt == 0) {
215          /*
216          * We need to make sure that prev is still accurate. It
217          * may not be, if someone rudely interrupted us....
218          */
219          /*
220          * 我们需要确信 prev 仍然是正确的，若某程序粗鲁地中断了我们
221          * 就有可能不是了。
222          */
223          // 如果 prev 已经不是搜索到的描述符的前一个描述符，则重新搜索当前描述符的前一个描述符。
224          if ((prev && (prev->next != bdesc)) ||
225              (!prev && (bdir->chain != bdesc)))
226              for (prev = bdir->chain; prev; prev = prev->next)
227                  if (prev->next == bdesc)
228                      break;
229          // 如果找到该前一个描述符，则从描述符链中删除当前描述符。
230          if (prev)
231              prev->next = bdesc->next;
232          // 如果 prev==NULL，则说明当前一个描述符是该目录项首个描述符，也即目录项中 chain 应该直接
233          // 指向当前描述符 bdesc，否则表示链表有问题，则显示出错信息，死机。因此，为了将当前描述符
234          // 从链表中删除，应该让 chain 指向下一个描述符。
235          else {
236              if (bdir->chain != bdesc)
237                  panic("malloc bucket chains corrupted");
238              bdir->chain = bdesc->next;
239          }
240          // 释放当前描述符所操作的内存页面，并将该描述符插入空闲描述符链表开始处。
241          free_page((unsigned long) bdesc->page);
242          bdesc->next = free_bucket_desc;
243          free_bucket_desc = bdesc;
244      }

```

```

    // 开中断，返回。
229     sti();
230     return;
231 }
232
233

```

12.10 open.c 程序

12.10.1 功能描述

open() 系统调用用于将一个文件名转换成一个文件描述符。当调用成功时，返回的文件描述符将是进程没有打开的最小数值的描述符。该调用创建一个新的打开文件，并不与任何其它进程共享。在执行 exec 函数时，该新的文件描述符将始终保持着打开状态。文件的读写指针被设置在文件开始位置。

参数 flag 是 O_RDONLY、O_WRONLY、O_RDWR 之一，分别代表文件只读打开、只写打开和读写打开方式，可以与其它一些标志一起使用。(参见 fs/open.c, 138 行)

12.10.2 代码注释

列表 linux/lib/open.c 程序

```

1  /*
2   *  linux/lib/open.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #define  __LIBRARY__
8  #include <unistd.h>          // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                                // 如定义了 __LIBRARY__，则还包括系统调用号和内嵌汇编 _syscall0()
等。
9  #include <stdarg.h>          // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了一个
                                // 类型 (va_list) 和三个宏 (va_start, va_arg 和 va_end)，用于
                                // vsprintf、vprintf、vfprintf 函数。
10
    // 打开文件函数。
    // 打开并有可能创建一个文件。
    // 参数: filename - 文件名; flag - 文件打开标志; ...
    // 返回: 文件描述符，若出错则置出错码，并返回-1。
11 int open(const char * filename, int flag, ...)
12 {
13     register int res;
14     va_list arg;
15
    // 利用 va_start() 宏函数，取得 flag 后面参数的指针，然后调用系统中断 int 0x80，功能 open 进行
    // 文件打开操作。
    // %0 - eax(返回的描述符或出错码); %1 - eax(系统中断调用功能号 __NR_open);
    // %2 - ebx(文件名 filename); %3 - ecx(打开文件标志 flag); %4 - edx(后随参数文件属性 mode)。

```

```

16     va_start(arg, flag);
17     __asm__ ("int $0x80"
18             : "=a" (res)
19             : "" ( _NR_open), "b" (filename), "c" (flag),
20               "d" (va_arg(arg, int)));
    // 系统中断调用返回值大于或等于 0，表示是一个文件描述符，则直接返回之。
21     if (res >= 0)
22         return res;
    // 否则说明返回值小于 0，则代表一个出错码。设置该出错码并返回-1。
23     errno = -res;
24     return -1;
25 }
26

```

12.11 setsid.c 程序

12.11.1 功能描述

该程序包括一个 setsid() 系统调用函数。如果调用的进程不是一个组的领导时，该函数用于创建一个新会话。则调用进程将成为该新会话的领导、新进程组的组领导，并且没有控制终端。调用进程的组 id 和会话 id 被设置成进程的 PID(进程标识符)。调用进程将成为新进程组和新会话中的唯一进程。

12.11.2 代码注释

列表 linux/lib/setsid.c 程序

```

1  /*
2   *  linux/lib/setsid.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #define  _LIBRARY
8  #include <unistd.h>          // Linux 标准头文件。定义了各种符号常数和类型，并申明了各种函数。
                                // 如定义了 __LIBRARY__，则还包括系统调用号和内嵌汇编 _syscall0()
等。
9
    // 创建一个会话并设置进程组号。
    // 下面系统调用宏对应于函数：pid_t setsid()。
    // 返回：调用进程的会话标识符(session ID)。
10 _syscall0(pid_t, setsid)
11

```

12.12 string.c 程序

12.12.1 功能描述

所有字符串操作函数已经在 string.h 中实现，因此 string.c 程序仅包含 string.h 头文件。

12.12.2 代码注释

列标 linux/lib/string.c 程序

```

1  /*
2   *  linux/lib/string.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #ifndef __GNUC__           // 需要 GNU 的 C 编译器编译。
8  #error I want gcc!
9  #endif
10
11 #define extern
12 #define inline
13 #define LIBRARY
14 #include <string.h>
15

```

12.13 wait.c 程序

12.13.1 功能描述

该程序包括函数 waitpid() 和 wait()。这两个函数允许进程获取与其子进程之一的状态信息。各种选项允许获取已经终止或停止的子进程状态信息。如果存在两个或两个以上子进程的状态信息，则报告的顺序是不指定的。

wait() 将挂起当前进程，直到其子进程之一退出（终止），或者收到要求终止该进程的信号，或者是需要调用一个信号句柄（信号处理程序）。

waitpid() 挂起当前进程，直到 pid 指定的子进程退出（终止）或者收到要求终止该进程的信号，或者是需要调用一个信号句柄（信号处理程序）。

如果 pid = -1, options = 0, 则 waitpid() 的作用与 wait() 函数一样。否则其行为将随 pid 和 options 参数的不同而不同。（参见 kernel/exit.c, 142）

12.13.2 代码注释

列表 linux/lib/wait.c 程序

```

1  /*
2   * linux/lib/wait.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  #define LIBRARY
8  #include <unistd.h>          // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                                // 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()
等。
9  #include <sys/wait.h>       // 等待调用头文件。定义系统调用 wait() 和 waitpid() 及相关常数符号。
10
    // 等待进程终止系统调用函数。
    // 该下面宏结构对应于函数：pid_t waitpid(pid_t pid, int * wait_stat, int options)
    //
    // 参数：pid - 等待被终止进程的进程 id，或者是用于指定特殊情况的其它特定数值；
    //        wait_stat - 用于存放状态信息；options - WNOHANG 或 WUNTRACED 或是 0。
11  _syscall3(pid_t, waitpid, pid_t, pid, int *, wait_stat, int, options)
12
    // 等待进程终止系统调用函数。直接调用 waitpid() 函数。
13  pid_t wait(int * wait_stat)
14  {
15      return waitpid(-1, wait_stat, 0);
16  }
17

```

12.14 write.c 程序

12.14.1 功能描述

该程序中包括一个向文件描述符写操作函数 write()。该函数向文件描述符指定的文件写入 count 字节的数据到缓冲区 buf 中。

12.14.2 代码注释

列表 linux/lib/write.c 程序

```

1  /*
2   * linux/lib/write.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  #define LIBRARY
8  #include <unistd.h>          // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。

```

```
// 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall10()
```

等。

9

```
///// 写文件系统调用函数。
```

```
// 该宏结构对应于函数: int write(int fd, const char * buf, off_t count)
```

```
// 参数: fd - 文件描述符; buf - 写缓冲区指针; count - 写字节数。
```

```
// 返回: 成功时返回写入的字节数(0 表示写入 0 字节); 出错时将返回-1, 并且设置了出错号。
```

```
10 \_syscall13(int, write, int, fd, const char *, buf, off\_t, count)
```

```
11
```
