System Interrupt Discovery Subsystem (SIDS)

Timothy Daly

January 8, 2007

Contents

Abstract

Certain library routines use standard system calls, known here as INT 80 calls, to ask for services.

This whitepaper reviews the current information available to us by knowing that a system library routine contains an INT 80 system call. After a detailed information review we propose a method to recognize and extract the system service semantics. This can be used to replace the system call by inline semantics.

1 INT 80 system calls[6]

By convention, **INT 80** is used as a system calling hook in all known operating systems. The EAX register[6] is loaded with a byte. The value of this byte is a code used by the system subroutine to select which system service to perform.

We can find every occurence of the **INT 80** instruction in the assembler output. This give us a strong indication that a system service is being performed.

In order to decide what service is being performed we need two pieces of information. First we need to know what operating system type and version the code where the code will execute. We need to know this because the service being performed varies between operating systems.

We also need to know the value of the EAX register at the time of the call, as well as the value of other registers depending on the type of call being made (e.g. a change directory request has to have a pointer to the desired directory in another register).

Armed with these two facts we can begin to unravel the meaning of a short stretch of the binary code (similar in spirit to finding a marker section for a particular gene).

The general flow of discovery seems to be:

- find the int 80s
- search for the EAX code
- find other parameters
- try to match library code
- on match, extract info
- inline CCAs

In more detail this expands into the following sequence. Each interesting line has a number of the subsection discussing some of the issues in detail. We will use the **chdir** and sometimes the **fchdir** system calls to illustrate the issues.

```
Find the {\bf INT 80}                                    (1)
Repeat
  add previous assembler instruction                     (2)
```

```
  extract semantics                                    (3)
Until (EAX has code)                                   (4)
Lookup INT 80 code profile                             (5)
If (more registers needed)                             (6)
  Repeat
    add previous assembler instruction
    extract semantics
  Until (all registers covered)                        (7)
Select all known system services for EAX=nn            (8)
Repeat
  add previous assembler instruction                   (9)
  compare with selected set                            (10)
Until (almost empty)                                   (11)
Repeat
  add next assembler instruction                       (12)
  compare with selected set
Until (almost empty)                                   (13)
Case
 No Match ==> not a known library call                 (14)
 nn Match ==> ambiguous library call                   (15)
  1 Match ==> recognized library call                  (16)
     Extract compiler information                      (17)
     Replace library routine with CCAs                 (18)
     Flag extent of library routine                    (19)
For each library call                                  (20)
  replace call by CCAs                                 (21)
```

Mark points out that the above procedure appears to be a heuristic. An alternate approach will be to replace the **INT 80** call with a case statement that includes all 256 cases. Once this substitution occurs we can process the whole of the subroutine as normal code. In theory this is correct. The downside is that this will eventually involve the whole operating system being substituted in the case statement. The INT 80 call is the main interface between a user and the operating system.

In any case, this is a point that needs more discussion.

2 The chdir system call

In order to explain the steps above we are going to use the **chdir** system call and its close cousin the **fchdir** system call. This will give us a basis for being specific.

First we look at the UNIX man page for **chdir** and **fchdir**, included below. Here we see that there is a close relation between the two functions, differing only in the form of the argument. This will become important during the explanation of line (6) above.

2.1 The chdir man page[5]

```
CHDIR(2)                  Linux Programmers Manual                  CHDIR(2)

NAME
       chdir, fchdir - change working directory

SYNOPSIS
       #include <unistd.h>

       int chdir(const char *path);
       int fchdir(int fd);

DESCRIPTION
       chdir changes the current directory to that specified in path.

       fchdir  is  identical  to chdir, only that the directory is given
       as an open file descriptor.

RETURN VALUE
       On success, zero is returned.  On error, -1 is returned, and
       errno is set appropriately.

ERRORS
       Depending  on  the file system, other errors can be returned.
       The more general errors for chdir are listed below:

       EFAULT (14) path points outside your accessible address space.

       ENAMETOOLONG (36) path is too long.

       ENOENT (2) The file does not exist.

       ENOMEM (12)  Insufficient kernel memory was available.

       ENOTDIR (20) A component of path is not a directory.

       EACCES (13) Search permission is denied on a component of path.

       ELOOP (40) Too many symbolic links were encountered
                  in resolving path.

       EIO (5) An I/O error occurred.

       The general errors for fchdir are listed below:
```

```
       EBADF (9)  fd is not a valid file descriptor.

       EACCES (13)  Search permission was denied on the directory open on fd.


NOTES
       The prototype for fchdir is only available if  _BSD_SOURCE  is  defined
       (either  explicitly,  or  implicitly,  by not defining _POSIX_SOURCE or
       compiling with the -ansi flag).

CONFORMING TO
       The chdir call is compatible with SVr4, SVID,  POSIX,  X/OPEN,  4.4BSD.
       SVr4  documents  additional  EINTR, ENOLINK, and EMULTIHOP error condi-
       tions but has no ENOMEM.  POSIX.1 does not have ENOMEM or  ELOOP  error
       conditions.   X/OPEN  does  not have EFAULT, ENOMEM or EIO error condi-
       tions.

       The fchdir call is compatible with SVr4, 4.4BSD and X/OPEN.  SVr4 docu-
       ments additional EIO, EINTR, and ENOLINK error conditions.  X/OPEN doc-
       uments additional EINTR and EIO error conditions.

SEE ALSO
       getcwd(3), chroot(2)

Linux 2.0.30                        1997-08-21                          CHDIR(2)
```

2.2 The chdir and fchdir source code[1, 2]

When we look at the above definition it is clear that **chdir** and **fchdir** are functionally close. We can see this from the source code:

CHDIR SOURCE CODE

```
#include <errno.h>
#include <stddef.h>
#include <unistd.h>

/* Change the current directory to PATH.  */
int
__chdir (path)
     const char *path;
{
  if (path == NULL)
    {
      __set_errno (EINVAL);
      return -1;
    }
```

```
  __set_errno (ENOSYS);
  return -1;
}
stub_warning (chdir)

weak_alias (__chdir, chdir)
#include <stub-tag.h>
```

FCHDIR SOURCE CODE

```
#include <errno.h>
#include <stddef.h>
#include <unistd.h>

/* Change the current directory to FD.  */
int
fchdir (fd)
     int fd;
{
  __set_errno (ENOSYS);
  return -1;
}

stub_warning (fchdir)
#include <stub-tag.h>
```

Notice that these reference external symbols which we find in `errno.h`[8]:

```
#define EINVAL 22 /* Invalid argument */
#define ENOSYS 38 /* Function not implemented */
```

However, it is clear that the error codes which the man page references must be set elsewhere because they are never set here. Thus we really don't understand how **chdir** works from looking at the "user level" source code. That's because **chdir** is actually implemented at the system level.

3 The VFS Virtual File System[18]

3.1 What is it?

The Virtual File System (otherwise known as the Virtual Filesystem Switch) is the software layer in the kernel that provides the filesystem interface to userspace programs. It also provides an abstraction within the kernel which allows different filesystem implementations to co-exist.

3.2 A Quick Look At How It Works

In this section I'll briefly describe how things work, before launching into the details. I'll start with describing what happens when user programs open and manipulate files, and then look from the other view which is how a filesystem is supported and subsequently mounted.

3.3 Opening a File

The VFS implements the open(2), stat(2), chmod(2) and similar system calls. The pathname argument is used by the VFS to search through the directory entry cache (dentry cache or "dcache"). This provides a very fast lookup mechanism to translate a pathname (filename) into a specific dentry.

An individual dentry usually has a pointer to an inode. Inodes are the things that live on disc drives, and can be regular files (you know: those things that you write data into), directories, FIFOs and other beasts. Dentries live in RAM and are never saved to disc: they exist only for performance. Inodes live on disc and are copied into memory when required. Later any changes are written back to disc. The inode that lives in RAM is a VFS inode, and it is this which the dentry points to. A single inode can be pointed to by multiple dentries (think about hardlinks).

The dcache is meant to be a view into your entire filespace. Unlike Linus, most of us losers can't fit enough dentries into RAM to cover all of our filespace, so the dcache has bits missing. In order to resolve your pathname into a dentry, the VFS may have to resort to creating dentries along the way, and then loading the inode. This is done by looking up the inode.

To lookup an inode (usually read from disc) requires that the VFS calls the lookup() method of the parent directory inode. This method is installed by the specific filesystem implementation that the inode lives in. There will be more on this later.

Once the VFS has the required dentry (and hence the inode), we can do all those boring things like open(2) the file, or stat(2) it to peek at the inode data. The stat(2) operation is fairly simple: once the VFS has the dentry, it peeks at the inode data and passes some of it back to userspace.

Opening a file requires another operation: allocation of a file structure (this is the kernel-side implementation of file descriptors). The freshly allocated file structure is initialised with a pointer to the dentry and a set of file operation member functions. These are taken from the inode data. The open() file method is then called so the specific filesystem implementation can do it's work. You can see that this is another switch performed by the VFS.

The file structure is placed into the file descriptor table for the process.

Reading, writing and closing files (and other assorted VFS operations) is done by using the userspace file descriptor to grab the appropriate file structure, and then calling the required file structure method function to do whatever is required.

For as long as the file is open, it keeps the dentry "open" (in use), which in turn means that the VFS inode is still in use.

All VFS system calls (i.e. open(2), stat(2), read(2), write(2), chmod(2) and so on) are called from a process context. You should assume that these calls are made without any kernel locks being held. This means that the processes may be executing the same piece of filesystem or driver code at the same time, on different processors. You should ensure that access to shared resources is protected by appropriate locks.

3.4 Registering and Mounting a Filesystem

If you want to support a new kind of filesystem in the kernel, all you need to do is call register_filesystem(). You pass a structure describing the filesystem implementation (struct file_system_type) which is then added to an internal table of supported filesystems. You can do:

```
% cat /proc/filesystems
```

to see what filesystems are currently available on your system.

When a request is made to mount a block device onto a directory in your filespace the VFS will call the appropriate method for the specific filesystem. The dentry for the mount point will then be updated to point to the root inode for the new filesystem.

It's now time to look at things in more detail.

3.5 The file_system_type struct

This describes the filesystem. As of kernel 2.1.99, the following members are defined:

```
struct file_system_type {
const char *name;
int fs_flags;
struct super_block *(*read_super) (struct super_block *, void *, int);
struct file_system_type * next;
};
```

- name: the name of the filesystem type, such as "ext2", "iso9660", "msdos" and so on
- fs_flags: various flags (i.e. FS_REQUIRES_DEV, FS_NO_DCACHE, etc.)
- read_super: the method to call when a new instance of this filesystem should be mounted.
- next: for internal VFS use: you should initialise this to NULL

The read_super() method has the following arguments:

- struct super_block *sb: the superblock structure. This is partially initialised by the VFS and the rest must be initialised by the read_super() method
- void *data: arbitrary mount options, usually comes as an ASCII string
- int silent: whether or not to be silent on error

The read_super() method must determine if the block device specified in the superblock contains a filesystem of the type the method supports. On success the method returns the superblock pointer, on failure it returns NULL.

The most interesting member of the superblock structure that the read_super() method fills in is the "s_op" field. This is a pointer to a "struct super_operations" which describes the next level of the filesystem implementation.

3.6 The super_operations struct

This describes how the VFS can manipulate the superblock of your filesystem. As of kernel 2.1.99, the following members are defined:

```
struct super_operations {
void (*read_inode) (struct inode *);
void (*write_inode) (struct inode *, int);
void (*put_inode) (struct inode *);
void (*delete_inode) (struct inode *);
int (*notify_change) (struct dentry *, struct iattr *);
void (*put_super) (struct super_block *);
void (*write_super) (struct super_block *);
int (*statfs) (struct super_block *, struct statfs *, int);
int (*remount_fs) (struct super_block *, int *, char *);
void (*clear_inode) (struct inode *);
};
```

All methods are called without any locks being held, unless otherwise noted. This means that most methods can block safely. All methods are only called from a process context (i.e. not from an interrupt handler or bottom half).

- read_inode: this method is called to read a specific inode from the mounted filesystem. The "i_ino" member in the "struct inode" will be initialised by the VFS to indicate which inode to read. Other members are filled in by this method
- write_inode: this method is called when the VFS needs to write an inode to disc. The second parameter indicates whether the write should be synchronous or not, not all filesystems check this flag.
- put_inode: called when the VFS inode is removed from the inode cache. This method is optional

- delete_inode: called when the VFS wants to delete an inode
- notify_change: called when VFS inode attributes are changed. If this is NULL the VFS falls back to the write_inode() method. This is called with the kernel lock held
- put_super: called when the VFS wishes to free the superblock (i.e. unmount). This is called with the superblock lock held
- write_super: called when the VFS superblock needs to be written to disc. This method is optional
- statfs: called when the VFS needs to get filesystem statistics. This is called with the kernel lock held
- remount_fs: called when the filesystem is remounted. This is called with the kernel lock held
- clear_inode: called then the VFS clears the inode. Optional

The read_inode() method is responsible for filling in the "i_op" field. This is a pointer to a "struct inode_operations" which describes the methods that can be performed on individual inodes.

3.7 The inode_operations struct[13]

This describes how the VFS can manipulate an inode in your filesystem. As of kernel 2.1.99, the following members are defined:

```
struct inode_operations {
struct file_operations * default_file_ops;
int (*create) (struct inode *,struct dentry *,int);
int (*lookup) (struct inode *,struct dentry *);
int (*link) (struct dentry *,struct inode *,struct dentry *);
int (*unlink) (struct inode *,struct dentry *);
int (*symlink) (struct inode *,struct dentry *,const char *);
int (*mkdir) (struct inode *,struct dentry *,int);
int (*rmdir) (struct inode *,struct dentry *);
int (*mknod) (struct inode *,struct dentry *,int,int);
int (*rename) (struct inode *, struct dentry *,
struct inode *, struct dentry *);
int (*readlink) (struct dentry *, char *,int);
struct dentry * (*follow_link) (struct dentry *, struct dentry *);
int (*readpage) (struct file *, struct page *);
int (*writepage) (struct file *, struct page *);
int (*bmap) (struct inode *,int);
void (*truncate) (struct inode *);
int (*permission) (struct inode *, int);
int (*smap) (struct inode *,int);
```

```
int (*updatepage) (struct file *, struct page *, const char *,
unsigned long, unsigned int, int);
int (*revalidate) (struct dentry *);
};
```

Again, all methods are called without any locks being held, unless otherwise noted.

- default_file_ops: this is a pointer to a "struct file_operations" which describes how to open and then manipulate open files
- create: called by the open(2) and creat(2) system calls. Only required if you want to support regular files. The dentry you get should not have an inode (i.e. it should be a negative dentry). Here you will probably call d_instantiate() with the dentry and the newly created inode
- lookup: called when the VFS needs to lookup an inode in a parent directory. The name to look for is found in the dentry. This method must call d_add() to insert the found inode into the dentry. The "i_count" field in the inode structure should be incremented. If the named inode does not exist a NULL inode should be inserted into the dentry (this is called a negative dentry). Returning an error code from this routine must only be done on a real error, otherwise creating inodes with system calls like create(2), mknod(2), mkdir(2) and so on will fail. If you wish to overload the dentry methods then you should initialise the "d_dop" field in the dentry; this is a pointer to a struct "dentry_operations". This method is called with the directory inode semaphore held
- link: called by the link(2) system call. Only required if you want to support hard links. You will probably need to call d_instantiate() just as you would in the create() method
- unlink: called by the unlink(2) system call. Only required if you want to support deleting inodes
- symlink: called by the symlink(2) system call. Only required if you want to support symlinks. You will probably need to call d_instantiate() just as you would in the create() method
- mkdir: called by the mkdir(2) system call. Only required if you want to support creating subdirectories. You will probably need to call d_instantiate() just as you would in the create() method
- rmdir: called by the rmdir(2) system call. Only required if you want to support deleting subdirectories
- mknod: called by the mknod(2) system call to create a device (char, block) inode or a named pipe (FIFO) or socket. Only required if you want to support creating these types of inodes. You will probably need to call d_instantiate() just as you would in the create() method

- readlink: called by the readlink(2) system call. Only required if you want to support reading symbolic links
- follow_link: called by the VFS to follow a symbolic link to the inode it points to. Only required if you want to support symbolic links

3.8 The file_operations struct[13]

This describes how the VFS can manipulate an open file. As of kernel 2.1.99, the following members are defined:

```
struct file_operations {
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *);
int (*fasync) (struct file *, int);
int (*check_media_change) (kdev_t dev);
int (*revalidate) (kdev_t dev);
int (*lock) (struct file *, int, struct file_lock *);
};
```

Again, all methods are called without any locks being held, unless otherwise noted.

- llseek: called when the VFS needs to move the file position index
- read: called by read(2) and related system calls
- write: called by write(2) and related system calls
- readdir: called when the VFS needs to read the directory contents
- poll: called by the VFS when a process wants to check if there is activity on this file and (optionally) go to sleep until there is activity. Called by the select(2) and poll(2) system calls
- ioctl: called by the ioctl(2) system call
- mmap: called by the mmap(2) system call

- open: called by the VFS when an inode should be opened. When the VFS opens a file, it creates a new "struct file" and initialises the "f_op" file operations member with the "default_file_ops" field in the inode structure. It then calls the open method for the newly allocated file structure. You might think that the open method really belongs in "struct inode_operations", and you may be right. I think it's done the way it is because it makes filesystems simpler to implement. The open() method is a good place to initialise the "private_data" member in the file structure if you want to point to a device structure

- release: called when the last reference to an open file is closed

- fsync: called by the fsync(2) system call

- fasync: called by the fcntl(2) system call when asynchronous (non-blocking) mode is enabled for a file

Note that the file operations are implemented by the specific filesystem in which the inode resides. When opening a device node (character or block special) most filesystems will call special support routines in the VFS which will locate the required device driver information. These support routines replace the filesystem file operations with those for the device driver, and then proceed to call the new open() method for the file. This is how opening a device file in the filesystem eventually ends up calling the device driver open() method. Note the devfs (the Device FileSystem) has a more direct path from device node to device driver (this is an unofficial kernel patch).

3.9 The dentry_operations struct[15]

This describes how a filesystem can overload the standard dentry operations. Dentries and the dcache are the domain of the VFS and the individual filesystem implementations. Device drivers have no business here. These methods may be set to NULL, as they are either optional or the VFS uses a default. As of kernel 2.1.99, the following members are defined:

```
struct dentry_operations {
int (*d_revalidate)(struct dentry *);
int (*d_hash) (struct dentry *, struct qstr *);
int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
void (*d_delete)(struct dentry *);
void (*d_release)(struct dentry *);
void (*d_iput)(struct dentry *, struct inode *);
};
```

- d_revalidate: called when the VFS needs to revalidate a dentry. This is called whenever a name lookup finds a dentry in the dcache. Most filesystems leave this as NULL, because all their dentries in the dcache are valid

- d_hash: called when the VFS adds a dentry to the hash table
- d_compare: called when a dentry should be compared with another
- d_delete: called when the last reference to a dentry is deleted. This means no-one is using the dentry, however it is still valid and in the dcache
- d_release: called when a dentry is really deallocated
- d_iput: called when a dentry looses its inode (just prior to its being deallocated). The default when this is NULL is that the VFS calls iput(). If you define this method, you must call iput() yourself

Each dentry has a pointer to its parent dentry, as well as a hash list of child dentries. Child dentries are basically like files in a directory.

There are a number of functions defined which permit a filesystem to manipulate dentries:

- dget: open a new handle for an existing dentry (this just increments the usage count)
- dput: close a handle for a dentry (decrements the usage count). If the usage count drops to 0, the "d_delete" method is called and the dentry is placed on the unused list if the dentry is still in its parents hash list. Putting the dentry on the unused list just means that if the system needs some RAM, it goes through the unused list of dentries and deallocates them. If the dentry has already been unhashed and the usage count drops to 0, in this case the dentry is deallocated after the "d_delete" method is called
- d_drop: this unhashes a dentry from its parents hash list. A subsequent call to dput() will dellocate the dentry if its usage count drops to 0
- d_delete: delete a dentry. If there are no other open references to the dentry then the dentry is turned into a negative dentry (the d_iput() method is called). If there are other references, then d_drop() is called instead
- d_add: add a dentry to its parents hash list and then calls d_instantiate()
- d_instantiate: add a dentry to the alias hash list for the inode and updates the "d_inode" member. The "i_count" member in the inode structure should be set/incremented. If the inode pointer is NULL, the dentry is called a "negative dentry". This function is commonly called when an inode is created for an existing negative dentry

4 The chdir system internals

Here we take a walk into the Linux RedHat 9 version of the **chdir** routine in order to understand how it really functions. We will start with the internal

function, walk into its underlying implementation, and then show how it is connected to the system so the linker can find it. This may seem painful but we can't really say we understand how **chdir** works unless we do this.

4.1 The sys_chdir function

We present the system version of the **chdir** function and then explain various important parts. All of the incidental data structures and functions are in the last section. Here we are concerned with information manipulated by the primary data flow.

The code creates a new, temporary data structure **nd** which is a **nameidata** struct. This will hold information about the result of the call to **__user_walk**.

There are three possible exits from this program. The first case is an error exit because we cannot find the directory. The second case is an error exit because we do not have permission to change to that directory. The third case is a correct exit which updates the current working directory with the newly discovered directory.

```
asmlinkage long sys_chdir(const char * filename)
{
  int error;
  struct nameidata nd;
  error =
    __user_walk(filename,LOOKUP_POSITIVE|LOOKUP_FOLLOW|LOOKUP_DIRECTORY,&nd);
  if (error)
    goto out;

  error = permission(nd.dentry->d_inode,MAY_EXEC);
  if (error)
    goto dput_and_out;

  set_fs_pwd(current->fs, nd.mnt, nd.dentry);

dput_and_out:
  path_release(&nd);
out:
  return error;
}
```

4.2 The nameidata struct[13]

The **sys_chdir** directory function manipulates a structure called **nameidata**.

```
struct nameidata {
struct dentry *dentry;
struct vfsmount *mnt;
struct qstr last;
```

```
unsigned int flags;
int last_type;
};
```

- dentry is the cached inode information for the search
- mnt is the virtual file system mount point for this search
- last is the "quick string" for this path
- flags controls the action during the query The flags bitmask can have the following values:

  ```
  #define LOOKUP_FOLLOW (1)  -- follow symlinks
  #define LOOKUP_DIRECTORY (2)  -- look for a directory
  #define LOOKUP_CONTINUE (4)  -- resume
  #define LOOKUP_POSITIVE (8)  -- must exist
  #define LOOKUP_PARENT (16) -- walk upward
  #define LOOKUP_NOALT (32) -- used from root search
  #define LOOKUP_ATOMIC (64) -- network lookups
  ```

- last_type is used during LOOKUP_PARENT searches and can be one of:

  ```
  enum {LAST\_NORM, LAST\_ROOT, LAST\_DOT, LAST\_DOTDOT, LAST\_BIND};
  ```

There are three interesting functions in sys_chdir:

- __user_walk to search for the directory
- permission to check that we are allowed to make the change
- set_fs_pwd to update our current working directory

4.3 The __user_walk function[12]

```
int __user_walk(const char *name, unsigned flags, struct nameidata *nd)
{
char *tmp;
int err;

tmp = getname(name);
err = PTR_ERR(tmp);
if (!IS_ERR(tmp)) {
err = 0;
err = path_lookup(tmp, flags, nd);
putname(tmp);
}
return err;
}
```

4.4 The getname function[12]

```
char * getname(const char * filename)
{
char *tmp, *result;

result = ERR_PTR(-ENOMEM);
tmp = __getname();
if (tmp)  {
int retval = do_getname(filename, tmp);

result = tmp;
if (retval < 0) {
putname(tmp);
result = ERR_PTR(retval);
}
}
return result;
}
```

4.5 Return Conventions

For code called in user context, it's very common to defy C convention, and return 0 for success, and a negative error number (eg. -EFAULT) for failure. This can be unintuitive at first, but it's fairly widespread in the networking code, for example.

The filesystem code uses ERR_PTR() in include/linux/fs.h; to encode a negative error number into a pointer, and IS_ERR() and PTR_ERR() to get it back out again: avoids a separate pointer parameter for the error number.

4.6 The ERR_PTR function[13]

```
static inline void *ERR_PTR(long error)
{
return (void *) error;
}
```

4.7 The PTR_ERR function[13]

```
static inline long PTR_ERR(const void *ptr)
{
return (long) ptr;
}
```

4.8 The IS_ERR function[13]

```
static inline long IS_ERR(const void *ptr)
```

```
{
return (unsigned long)ptr > (unsigned long)-1000L;
}
```

4.9 Memory Cache Slab allocator[14, 30, 31]

The memory is organized in caches, one cache for each object type. (e.g. inode_cache, dentry_cache, buffer_head, vm_area_struct) Each cache consists out of many slabs (they are small (usually one page long) and always contiguous), and each slab contains multiple initialized objects.

Each cache can only support one memory type (GFP_DMA, GFP_HIGHMEM, normal). If you need a special memory type, then must create a new cache for that memory type.

In order to reduce fragmentation, the slabs are sorted in 3 groups:

- full slabs with 0 free objects
- partial slabs
- empty slabs with no allocated objects

If partial slabs exist, then new allocations come from these slabs, otherwise from empty slabs or new slabs are allocated.

kmem_cache_destroy() CAN CRASH if you try to allocate from the cache during kmem_cache_destroy(). The caller must prevent concurrent allocs.

On SMP systems, each cache has a short per-cpu head array, most allocs and frees go into that array, and if that array overflows, then 1/2 of the entries in the array are given back into the global cache. This reduces the number of spinlock operations.

The c_cpuarray may not be read with enabled local interrupts.

SMP synchronization:

- constructors and destructors are called without any locking.
- Several members in kmem_cache_t and slab_t never change, they are accessed without any locking.
- The per-cpu arrays are never accessed from the wrong cpu, no locking.
- The non-constant members are protected with a per-cache irq spinlock.

The global cache-chain is protected by the semaphore 'cache_chain_sem'. The sem is only needed when accessing/extending the cache-chain, which can never happen inside an interrupt (kmem_cache_create(), kmem_cache_shrink() and kmem_cache_reap()).

To prevent kmem_cache_shrink() trying to shrink a 'growing' cache (which maybe be sleeping and therefore not holding the semaphore/lock), the growing field is used. This also prevents reaping from a cache.

At present, each engine can be growing a cache. This should be blocked.

4.10 The __getname function[13]

```
#define __getname() kmem_cache_alloc(names_cachep, SLAB_KERNEL)
```

4.11 The putname function[13]

```
#define putname(name) kmem_cache_free(names_cachep, (void *)(name))
```

4.12 The kmem_cache_alloc function[14]

Allocate an object from this cache.

```
/**
 * kmem_cache_alloc - Allocate an object
 * @cachep: The cache to allocate from.
 * @flags: See kmalloc().
 *
 */
void * kmem_cache_alloc (kmem_cache_t *cachep, int flags)
{
return __kmem_cache_alloc(cachep, flags);
}
```

- cachep is the cache to alloc from
- flags is one of

  ```
  %GFP_USER - Allocate memory on behalf of user.  May sleep.
  %GFP_KERNEL - Allocate normal kernel ram.  May sleep.
  %GFP_ATOMIC - Allocation will not sleep.  Use inside interrupt handlers.
  Additionally, the %GFP_DMA flag may be set to indicate the memory
  must be suitable for DMA.  This can mean different things on different
  platforms.  For example, on i386, it means that the memory must come
  from the first 16MB.
  ```

  The flags are only relevant if the cache has no available objects.

4.13 The __kmem_cache_alloc function[14]

```
static inline void * __kmem_cache_alloc (kmem_cache_t *cachep, int flags)
{
unsigned long save_flags;
void* objp;

kmem_cache_alloc_head(cachep, flags);
try_again:
local_irq_save(save_flags);
#ifdef CONFIG_SMP
```

```
{
cpucache_t *cc = cc_data(cachep);

if (cc) {
if (cc->avail) {
STATS_INC_ALLOCHIT(cachep);
objp = cc_entry(cc)[--cc->avail];
} else {
STATS_INC_ALLOCMISS(cachep);
objp = kmem_cache_alloc_batch(cachep,cc,flags);
if (!objp)
goto alloc_new_slab_nolock;
}
} else {
spin_lock(&cachep->spinlock);
objp = kmem_cache_alloc_one(cachep);
spin_unlock(&cachep->spinlock);
}
}
#else
objp = kmem_cache_alloc_one(cachep);
#endif
local_irq_restore(save_flags);
return objp;
alloc_new_slab:
#ifdef CONFIG_SMP
spin_unlock(&cachep->spinlock);
alloc_new_slab_nolock:
#endif
local_irq_restore(save_flags);
if (kmem_cache_grow(cachep, flags))
/* Someone may have stolen our objs.  Doesn't matter, we'll
 * just come back here again.
 */
goto try_again;
return NULL;
}
```

4.14 The local_irq_save function[32]

```
#define local_irq_save(flags) __save_and_cli(flags)
```

4.15 The local_irq_restore function[32]

```
#define local_irq_restore(flags) __restore_flags(flags)
```

4.16 The __save_and_cli function[32]

```
#define __save_and_cli(flags) do { (flags) = swpipl(IPL_MAX); barrier(); } while(0)
```

4.17 The __restore_flags function[32]

```
#define __restore_flags(flags) do { barrier(); setipl(flags); barrier(); } while(0)
```

4.18 The setipl function[32]

```
#define setipl(ipl) ((void) swpipl(ipl))
```

4.19 The swpipl function[32]

```
This function is undefined.
```

4.20 The function[32]

4.21 The kmem_cache_alloc_head function[14]

This function appears to do nothing but error checking.

```
static inline void kmem_cache_alloc_head(kmem_cache_t *cachep, int flags)
{
if (flags & SLAB_DMA) {
if (!(cachep->gfpflags & GFP_DMA))
BUG();
} else {
if (cachep->gfpflags & GFP_DMA)
BUG();
}
}
```

4.22 The cpucache_t typedef[14]

Per cpu structures in which the limit is stored in the per-cpu structure to reduce the data cache footprint.

```
typedef struct cpucache_s {
unsigned int avail;
unsigned int limit;
} cpucache_t;
```

4.23 The cc_data function[14]

```
#define cc_data(cachep) \
((cachep)->cpudata[smp_processor_id()])
```

4.24 The STATS_INC_ALLOCHIT function[14]

```
#define STATS_INC_ALLOCHIT(x) atomic_inc(&(x)->allochit)
```

4.25 The cc_entry function[14]

```
#define cc_entry(cpucache) \
((void **)(((cpucache_t*)(cpucache))+1))
```

4.26 The STATS_INC_ALLOCMISS function[14]

```
#define STATS_INC_ALLOCMISS(x) atomic_inc(&(x)->allocmiss)
```

4.27 The kmem_cache_alloc_batch function[14]

```
void* kmem_cache_alloc_batch(kmem_cache_t* cachep, cpucache_t* cc, int flags)
{
int batchcount = cachep->batchcount;

spin_lock(&cachep->spinlock);
while (batchcount--) {
struct list_head * slabs_partial, * entry;
slab_t *slabp;
/* Get slab alloc is to come from. */
slabs_partial = &(cachep)->slabs_partial;
entry = slabs_partial->next;
if (unlikely(entry == slabs_partial)) {
struct list_head * slabs_free;
slabs_free = &(cachep)->slabs_free;
entry = slabs_free->next;
if (unlikely(entry == slabs_free))
break;
list_del(entry);
list_add(entry, slabs_partial);
}

slabp = list_entry(entry, slab_t, list);
cc_entry(cc)[cc->avail++] =
kmem_cache_alloc_one_tail(cachep, slabp);
}
spin_unlock(&cachep->spinlock);
```

```
if (cc->avail)
return cc_entry(cc)[--cc->avail];
return NULL;
}
```

4.28 The kmem_cache_alloc_one function[14]

Returns a ptr to an obj in the given cache. Caller must guarantee synchronization. Use #define for the goto optimization 8-)

```
#define kmem_cache_alloc_one(cachep) \
({ \
struct list_head * slabs_partial, * entry; \
slab_t *slabp; \
\
slabs_partial = &(cachep)->slabs_partial; \
entry = slabs_partial->next; \
if (unlikely(entry == slabs_partial)) { \
struct list_head * slabs_free; \
slabs_free = &(cachep)->slabs_free; \
entry = slabs_free->next; \
if (unlikely(entry == slabs_free)) \
goto alloc_new_slab; \
list_del(entry); \
list_add(entry, slabs_partial); \
} \
\
slabp = list_entry(entry, slab_t, list); \
kmem_cache_alloc_one_tail(cachep, slabp); \
})
```

4.29 The kmem_cache_alloc_one_tail function[14]

```
static inline void * kmem_cache_alloc_one_tail (kmem_cache_t *cachep,
slab_t *slabp)
{
void *objp;

STATS_INC_ALLOCED(cachep);
STATS_INC_ACTIVE(cachep);
STATS_SET_HIGH(cachep);

/* get obj pointer */
slabp->inuse++;
objp = slabp->s_mem + slabp->free*cachep->objsize;
slabp->free=slab_bufctl(slabp)[slabp->free];
```

```
if (unlikely(slabp->free == BUFCTL_END)) {
list_del(&slabp->list);
list_add(&slabp->list, &cachep->slabs_full);
}
#if DEBUG
if (cachep->flags & SLAB_POISON)
if (kmem_check_poison_obj(cachep, objp))
BUG();
if (cachep->flags & SLAB_RED_ZONE) {
/* Set alloc red-zone, and check old one. */
if (xchg((unsigned long *)objp, RED_MAGIC2) !=
 RED_MAGIC1)
BUG();
if (xchg((unsigned long *)(objp+cachep->objsize -
  BYTES_PER_WORD), RED_MAGIC2) != RED_MAGIC1)
BUG();
objp += BYTES_PER_WORD;
}
#endif
return objp;
}
```

4.30 The kmem_cache_grow function[14]

Grow (by 1) the number of slabs within a cache. This is called by kmem_cache_alloc() when there are no active objs left in a cache.

```
static int kmem_cache_grow (kmem_cache_t * cachep, int flags)
{
slab_t *slabp;
struct page *page;
void *objp;
size_t  offset;
unsigned int  i, local_flags;
unsigned long  ctor_flags;
unsigned long  save_flags;

/* Be lazy and only check for valid flags here,
   * keeping it out of the critical path in kmem_cache_alloc().
 */
if (flags & ~(SLAB_DMA|SLAB_LEVEL_MASK|SLAB_NO_GROW))
BUG();
if (flags & SLAB_NO_GROW)
return 0;
```

```
/*
 * The test for missing atomic flag is performed here, rather than
 * the more obvious place, simply to reduce the critical path length
 * in kmem_cache_alloc(). If a caller is seriously mis-behaving they
 * will eventually be caught here (where it matters).
 */
if (in_interrupt() && (flags & SLAB_LEVEL_MASK) != SLAB_ATOMIC)
BUG();

ctor_flags = SLAB_CTOR_CONSTRUCTOR;
local_flags = (flags & SLAB_LEVEL_MASK);
if (local_flags == SLAB_ATOMIC)
/*
 * Not allowed to sleep.  Need to tell a constructor about
 * this - it might need to know...
 */
ctor_flags |= SLAB_CTOR_ATOMIC;

/* About to mess with non-constant members - lock. */
spin_lock_irqsave(&cachep->spinlock, save_flags);

/* Get colour for the slab, and cal the next value. */
offset = cachep->colour_next;
cachep->colour_next++;
if (cachep->colour_next >= cachep->colour)
cachep->colour_next = 0;
offset *= cachep->colour_off;
cachep->dflags |= DFLGS_GROWN;

cachep->growing++;
spin_unlock_irqrestore(&cachep->spinlock, save_flags);

/* A series of memory allocations for a new slab.
 * Neither the cache-chain semaphore, or cache-lock, are
 * held, but the incrementing c_growing prevents this
 * cache from being reaped or shrunk.
 * Note: The cache could be selected in for reaping in
 * kmem_cache_reap(), but when the final test is made the
 * growing value will be seen.
 */

/* Get mem for the objs. */
if (!(objp = kmem_getpages(cachep, flags)))
goto failed;

/* Get slab management. */
```

```
if (!(slabp = kmem_cache_slabmgmt(cachep, objp, offset, local_flags)))
goto opps1;

/* Nasty!!!!!! I hope this is OK. */
i = 1 << cachep->gfporder;
page = virt_to_page(objp);
do {
SET_PAGE_CACHE(page, cachep);
SET_PAGE_SLAB(page, slabp);
PageSetSlab(page);
page++;
} while (--i);

kmem_cache_init_objs(cachep, slabp, ctor_flags);

spin_lock_irqsave(&cachep->spinlock, save_flags);
cachep->growing--;

/* Make slab active. */
list_add_tail(&slabp->list, &cachep->slabs_free);
STATS_INC_GROWN(cachep);
cachep->failures = 0;

spin_unlock_irqrestore(&cachep->spinlock, save_flags);
return 1;
opps1:
kmem_freepages(cachep, objp);
failed:
spin_lock_irqsave(&cachep->spinlock, save_flags);
cachep->growing--;
spin_unlock_irqrestore(&cachep->spinlock, save_flags);
return 0;
}
```

4.31 The do_getname function[12]

```
static inline int do_getname(const char *filename, char *page)
{
int retval;
unsigned long len = PATH_MAX;

if ((unsigned long) filename >= TASK_SIZE) {
if (!segment_eq(get_fs(), KERNEL_DS))
return -EFAULT;
} else if (TASK_SIZE - (unsigned long) filename < PATH_MAX)
len = TASK_SIZE - (unsigned long) filename;
```

```
retval = strncpy_from_user((char *)page, filename, len);
if (retval > 0) {
if (retval < len)
return 0;
return -ENAMETOOLONG;
} else if (!retval)
retval = -ENOENT;
return retval;
}
```

4.32 The permission function[12]

```
int permission(struct inode * inode,int mask)
{
if (inode->i_op && inode->i_op->permission) {
int retval;
lock_kernel();
retval = inode->i_op->permission(inode, mask);
unlock_kernel();
return retval;
}
return vfs_permission(inode, mask);
}
```

4.33 The set_fs_pwd function[29]

Replace the fs-¿pwdmnt,pwd with mnt,dentry. Put the old values. It can block. Requires the big lock held.

```
static inline void set_fs_pwd(struct fs_struct *fs,
struct vfsmount *mnt,
struct dentry *dentry)
{
struct dentry *old_pwd;
struct vfsmount *old_pwdmnt;
write_lock(&fs->lock);
old_pwd = fs->pwd;
old_pwdmnt = fs->pwdmnt;
fs->pwdmnt = mntget(mnt);
fs->pwd = dget(dentry);
write_unlock(&fs->lock);
if (old_pwd) {
dput(old_pwd);
mntput(old_pwdmnt);
}
```

```
}
```

5 The chdir user level code

5.1 The chdir ISA translation[3]

```
00000000  89 DA              mov        edx,ebx
00000002  8B 5C 24 04        mov        ebx,DWORD [esp+4]
00000006  B8 0C 00 00 00     mov        eax,0x0000000C
0000000B  CD 80              int        BYTE 128
0000000D  89 D3              mov        ebx,edx
0000000F  3D 01 F0 FF FF     cmp        eax,0xFFFFF001
00000014  0F 83 FC FF FF FF  jae        NEAR $+2
0000001A  C3                 ret
```

5.2 The chdir CCA translation[3]

```
00000000  89 DA                                 mov        edx,ebx
EDX := EBX
00000002  8B 5C 24 04                           mov        ebx,DWORD [esp+4]
EBX := ACCESS_32(M,DS,ADD_32(ESP,4))
00000006  B8 0C 00 00 00                        mov        eax,0x0000000C
COMMENT: NOT CHECKED B8
EAX := 12
0000000B  CD 80                                 int        BYTE 128
COMMENT: ERROR NO CCA CD
0000000D  89 D3                                 mov        ebx,edx
EBX := EDX
0000000F  3D 01 F0 FF FF                        cmp        eax,0xFFFFF001
COMMENT: NOT CHECKED 3D 32
OF := IS_IN_INTERVAL(EAX,2147483648,2147483647)
SF := IS_NEG_SIGNED_32(ADD_32(EAX,4095))
ZF := INT_EQUAL(EAX,4294963201)
AF := INT_EQUAL(MOD(EAX,16),0)
PF := IS_EVEN_PARITY_LOWBYTE(ADD_32(EAX,-1))
CF := LESS_OR_EQUAL(EAX,4294963200)
00000014  0F 83 FC FF FF FF                     jae        NEAR $+2
COMMENT: NOT CHECKED 0F 83
NOT(CF) -> 22 | CF -> 26
0000001A  C3                                    ret
ESP := ADD_32(ESP,4)
CASE(TRUE,ACCESS_32(M,SS,ESP))
```

5.3 The chdir system call[4]

```
#define __NR_chdir  12
```

```
#define __NR_fchdir 133
```

5.4 entry.S[9]

```
/*
 *  linux/arch/i386/entry.S
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 */

/*
 * entry.S contains the system-call and fault low-level handling routines.
 * This also contains the timer-interrupt handler, as well as all interrupts
 * and faults that can result in a task-switch.
 *
 * NOTE: This code handles signal-recognition, which happens every time
 * after a timer-interrupt and after each system call.
 *
 * I changed all the .align's to 4 (16 byte alignment), as that's faster
 * on a 486.
 *
 * Stack layout in 'ret_from_system_call':
 *  ptrace needs to have all regs on the stack.
 * if the order here is changed, it needs to be
 * updated in fork.c:copy_process, signal.c:do_signal,
 * ptrace.c and ptrace.h
 *
 *  0(%esp) - %ebx
 *  4(%esp) - %ecx
 *  8(%esp) - %edx
 *       C(%esp) - %esi
 * 10(%esp) - %edi
 * 14(%esp) - %ebp
 * 18(%esp) - %eax
 * 1C(%esp) - %ds
 * 20(%esp) - %es
 * 24(%esp) - orig_eax
 * 28(%esp) - %eip
 * 2C(%esp) - %cs
 * 30(%esp) - %eflags
 * 34(%esp) - %oldesp
 * 38(%esp) - %oldss
 *
 * "current" is in register %ebx during any slow entries.
 */
```

```
#include <linux/config.h>
#include <linux/sys.h>
#include <linux/linkage.h>
#include <asm/segment.h>
#include <asm/smp.h>

EBX = 0x00
ECX = 0x04
EDX = 0x08
ESI = 0x0C
EDI = 0x10
EBP = 0x14
EAX = 0x18
DS = 0x1C
ES = 0x20
ORIG_EAX = 0x24
EIP = 0x28
CS = 0x2C
EFLAGS = 0x30
OLDESP = 0x34
OLDSS = 0x38

CF_MASK = 0x00000001
TF_MASK = 0x00000100
IF_MASK = 0x00000200
DF_MASK = 0x00000400
NT_MASK = 0x00004000
VM_MASK = 0x00020000

/*
 * these are offsets into the task-struct.
 */
state =  0
flags =  4
sigpending =  8
addr_limit = 12
exec_domain = 16
need_resched = 20
tsk_ptrace = 24
cpu = 32

ENOSYS = 38


#define SAVE_ALL \
cld; \
```

```
pushl %es; \
pushl %ds; \
pushl %eax; \
pushl %ebp; \
pushl %edi; \
pushl %esi; \
pushl %edx; \
pushl %ecx; \
pushl %ebx; \
movl $(__KERNEL_DS),%edx; \
movl %edx,%ds; \
movl %edx,%es;

#define RESTORE_ALL \
popl %ebx; \
popl %ecx; \
popl %edx; \
popl %esi; \
popl %edi; \
popl %ebp; \
popl %eax; \
1: popl %ds; \
2: popl %es; \
addl $4,%esp; \
3: iret; \
.section .fixup,"ax"; \
4: movl $0,(%esp); \
jmp 1b; \
5: movl $0,(%esp); \
jmp 2b; \
6: pushl %ss; \
popl %ds; \
pushl %ss; \
popl %es; \
pushl $11; \
call do_exit; \
.previous; \
.section __ex_table,"a";\
.align 4; \
.long 1b,4b; \
.long 2b,5b; \
.long 3b,6b; \
.previous

#define GET_CURRENT(reg) \
movl $-8192, reg; \
```

```
andl %esp, reg

ENTRY(lcall7)
pushfl # We get a different stack layout with call gates,
pushl %eax # which has to be cleaned up later..
SAVE_ALL
movl EIP(%esp),%eax # due to call gates, this is eflags, not eip..
movl CS(%esp),%edx # this is eip..
movl EFLAGS(%esp),%ecx # and this is cs..
movl %eax,EFLAGS(%esp) #
andl $~(NT_MASK|TF_MASK|DF_MASK), %eax
pushl %eax
popfl
movl %edx,EIP(%esp) # Now we move them to their "normal" places
movl %ecx,CS(%esp) #
movl %esp,%ebx
pushl %ebx
andl $-8192,%ebx # GET_CURRENT
movl exec_domain(%ebx),%edx # Get the execution domain
movl 4(%edx),%edx # Get the lcall7 handler for the domain
pushl $0x7
call *%edx
addl $4, %esp
popl %eax
jmp ret_from_sys_call

ENTRY(lcall27)
pushfl # We get a different stack layout with call gates,
pushl %eax # which has to be cleaned up later..
SAVE_ALL
movl EIP(%esp),%eax # due to call gates, this is eflags, not eip..
movl CS(%esp),%edx # this is eip..
movl EFLAGS(%esp),%ecx # and this is cs..
movl %eax,EFLAGS(%esp) #
andl $~(NT_MASK|TF_MASK|DF_MASK), %eax
pushl %eax
popfl
movl %edx,EIP(%esp) # Now we move them to their "normal" places
movl %ecx,CS(%esp) #
movl %esp,%ebx
pushl %ebx
andl $-8192,%ebx # GET_CURRENT
movl exec_domain(%ebx),%edx # Get the execution domain
movl 4(%edx),%edx # Get the lcall7 handler for the domain
pushl $0x27
call *%edx
```

```
addl $4, %esp
popl %eax
jmp ret_from_sys_call


ENTRY(ret_from_fork)
#if CONFIG_SMP
pushl %ebx
call SYMBOL_NAME(schedule_tail)
addl $4, %esp
#endif
GET_CURRENT(%ebx)
testb $0x02,tsk_ptrace(%ebx) # PT_TRACESYS
jne tracesys_exit
jmp ret_from_sys_call

/*
 * Return to user mode is not as complex as all this looks,
 * but we want the default path for a system call return to
 * go as quickly as possible which is why some of this is
 * less clear than it otherwise should be.
 */

ENTRY(system_call)
pushl %eax # save orig_eax
SAVE_ALL
GET_CURRENT(%ebx)
testb $0x02,tsk_ptrace(%ebx) # PT_TRACESYS
jne tracesys
cmpl $(NR_syscalls),%eax
jae badsys
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
movl %eax,EAX(%esp) # save the return value
ENTRY(ret_from_sys_call)
cli # need_resched and signals atomic test
cmpl $0,need_resched(%ebx)
jne reschedule
cmpl $0,sigpending(%ebx)
jne signal_return
restore_all:
RESTORE_ALL

ALIGN
signal_return:
sti # we can get here from an interrupt handler
testl $(VM_MASK),EFLAGS(%esp)
```

```
movl %esp,%eax
jne v86_signal_return
xorl %edx,%edx
call SYMBOL_NAME(do_signal)
jmp restore_all

ALIGN
v86_signal_return:
call SYMBOL_NAME(save_v86_state)
movl %eax,%esp
xorl %edx,%edx
call SYMBOL_NAME(do_signal)
jmp restore_all

ALIGN
tracesys:
movl $-ENOSYS,EAX(%esp)
call SYMBOL_NAME(syscall_trace)
movl ORIG_EAX(%esp),%eax
cmpl $(NR_syscalls),%eax
jae tracesys_exit
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
movl %eax,EAX(%esp) # save the return value
tracesys_exit:
call SYMBOL_NAME(syscall_trace)
jmp ret_from_sys_call
badsys:
movl $-ENOSYS,EAX(%esp)
jmp ret_from_sys_call

ALIGN
ENTRY(ret_from_intr)
GET_CURRENT(%ebx)
ret_from_exception:
movl EFLAGS(%esp),%eax # mix EFLAGS and CS
movb CS(%esp),%al
testl $(VM_MASK | 3),%eax # return to VM86 mode or non-supervisor?
jne ret_from_sys_call
jmp restore_all

ALIGN
reschedule:
call SYMBOL_NAME(schedule)    # test
jmp ret_from_sys_call

ENTRY(divide_error)
```

```
pushl $0 # no error code
pushl $ SYMBOL_NAME(do_divide_error)
ALIGN
error_code:
pushl %ds
pushl %eax
xorl %eax,%eax
pushl %ebp
pushl %edi
pushl %esi
pushl %edx
decl %eax # eax = -1
pushl %ecx
pushl %ebx
cld
movl %es,%ecx
movl ORIG_EAX(%esp), %esi # get the error code
movl ES(%esp), %edi # get the function address
movl %eax, ORIG_EAX(%esp)
movl %ecx, ES(%esp)
movl %esp,%edx
pushl %esi # push the error code
pushl %edx # push the pt_regs pointer
movl $(__KERNEL_DS),%edx
movl %edx,%ds
movl %edx,%es
GET_CURRENT(%ebx)
call *%edi
addl $8,%esp
jmp ret_from_exception

ENTRY(coprocessor_error)
pushl $0
pushl $ SYMBOL_NAME(do_coprocessor_error)
jmp error_code

ENTRY(simd_coprocessor_error)
pushl $0
pushl $ SYMBOL_NAME(do_simd_coprocessor_error)
jmp error_code

ENTRY(device_not_available)
pushl $-1 # mark this as an int
SAVE_ALL
GET_CURRENT(%ebx)
movl %cr0,%eax
```

```
testl $0x4,%eax # EM (math emulation bit)
jne device_not_available_emulate
call SYMBOL_NAME(math_state_restore)
jmp ret_from_exception
device_not_available_emulate:
pushl $0 # temporary storage for ORIG_EIP
call  SYMBOL_NAME(math_emulate)
addl $4,%esp
jmp ret_from_exception

ENTRY(debug)
pushl $0
pushl $ SYMBOL_NAME(do_debug)
jmp error_code

ENTRY(nmi)
pushl %eax
SAVE_ALL
movl %esp,%edx
pushl $0
pushl %edx
call SYMBOL_NAME(do_nmi)
addl $8,%esp
RESTORE_ALL

ENTRY(int3)
pushl $0
pushl $ SYMBOL_NAME(do_int3)
jmp error_code

ENTRY(overflow)
pushl $0
pushl $ SYMBOL_NAME(do_overflow)
jmp error_code

ENTRY(bounds)
pushl $0
pushl $ SYMBOL_NAME(do_bounds)
jmp error_code

ENTRY(invalid_op)
pushl $0
pushl $ SYMBOL_NAME(do_invalid_op)
jmp error_code

ENTRY(coprocessor_segment_overrun)
```

```
pushl $0
pushl $ SYMBOL_NAME(do_coprocessor_segment_overrun)
jmp error_code

ENTRY(double_fault)
pushl $ SYMBOL_NAME(do_double_fault)
jmp error_code

ENTRY(invalid_TSS)
pushl $ SYMBOL_NAME(do_invalid_TSS)
jmp error_code

ENTRY(segment_not_present)
pushl $ SYMBOL_NAME(do_segment_not_present)
jmp error_code

ENTRY(stack_segment)
pushl $ SYMBOL_NAME(do_stack_segment)
jmp error_code

ENTRY(general_protection)
pushl $ SYMBOL_NAME(do_general_protection)
jmp error_code

ENTRY(alignment_check)
pushl $ SYMBOL_NAME(do_alignment_check)
jmp error_code

ENTRY(page_fault)
pushl $ SYMBOL_NAME(do_page_fault)
jmp error_code

ENTRY(machine_check)
pushl $0
pushl $ SYMBOL_NAME(do_machine_check)
jmp error_code

ENTRY(spurious_interrupt_bug)
pushl $0
pushl $ SYMBOL_NAME(do_spurious_interrupt_bug)
jmp error_code

.data
ENTRY(sys_call_table)
.long SYMBOL_NAME(sys_ni_syscall) /* 0  -  old "setup()" system call*/
.long SYMBOL_NAME(sys_exit)
```

```
.long SYMBOL_NAME(sys_fork)
.long SYMBOL_NAME(sys_read)
.long SYMBOL_NAME(sys_write)
.long SYMBOL_NAME(sys_open) /* 5 */
.long SYMBOL_NAME(sys_close)
.long SYMBOL_NAME(sys_waitpid)
.long SYMBOL_NAME(sys_creat)
.long SYMBOL_NAME(sys_link)
.long SYMBOL_NAME(sys_unlink) /* 10 */
.long SYMBOL_NAME(sys_execve)
.long SYMBOL_NAME(sys_chdir)
.long SYMBOL_NAME(sys_time)
.long SYMBOL_NAME(sys_mknod)
.long SYMBOL_NAME(sys_chmod) /* 15 */
.long SYMBOL_NAME(sys_lchown16)
.long SYMBOL_NAME(sys_ni_syscall) /* old break syscall holder */
.long SYMBOL_NAME(sys_stat)
.long SYMBOL_NAME(sys_lseek)
.long SYMBOL_NAME(sys_getpid) /* 20 */
.long SYMBOL_NAME(sys_mount)
.long SYMBOL_NAME(sys_oldumount)
.long SYMBOL_NAME(sys_setuid16)
.long SYMBOL_NAME(sys_getuid16)
.long SYMBOL_NAME(sys_stime) /* 25 */
.long SYMBOL_NAME(sys_ptrace)
.long SYMBOL_NAME(sys_alarm)
.long SYMBOL_NAME(sys_fstat)
.long SYMBOL_NAME(sys_pause)
.long SYMBOL_NAME(sys_utime) /* 30 */
.long SYMBOL_NAME(sys_ni_syscall) /* old stty syscall holder */
.long SYMBOL_NAME(sys_ni_syscall) /* old gtty syscall holder */
.long SYMBOL_NAME(sys_access)
.long SYMBOL_NAME(sys_nice)
.long SYMBOL_NAME(sys_ni_syscall) /* 35 */ /* old ftime syscall holder */
.long SYMBOL_NAME(sys_sync)
.long SYMBOL_NAME(sys_kill)
.long SYMBOL_NAME(sys_rename)
.long SYMBOL_NAME(sys_mkdir)
.long SYMBOL_NAME(sys_rmdir) /* 40 */
.long SYMBOL_NAME(sys_dup)
.long SYMBOL_NAME(sys_pipe)
.long SYMBOL_NAME(sys_times)
.long SYMBOL_NAME(sys_ni_syscall) /* old prof syscall holder */
.long SYMBOL_NAME(sys_brk) /* 45 */
.long SYMBOL_NAME(sys_setgid16)
.long SYMBOL_NAME(sys_getgid16)
```

```
.long SYMBOL_NAME(sys_signal)
.long SYMBOL_NAME(sys_geteuid16)
.long SYMBOL_NAME(sys_getegid16) /* 50 */
.long SYMBOL_NAME(sys_acct)
.long SYMBOL_NAME(sys_umount) /* recycled never used phys() */
.long SYMBOL_NAME(sys_ni_syscall) /* old lock syscall holder */
.long SYMBOL_NAME(sys_ioctl)
.long SYMBOL_NAME(sys_fcntl) /* 55 */
.long SYMBOL_NAME(sys_ni_syscall) /* old mpx syscall holder */
.long SYMBOL_NAME(sys_setpgid)
.long SYMBOL_NAME(sys_ni_syscall) /* old ulimit syscall holder */
.long SYMBOL_NAME(sys_olduname)
.long SYMBOL_NAME(sys_umask) /* 60 */
.long SYMBOL_NAME(sys_chroot)
.long SYMBOL_NAME(sys_ustat)
.long SYMBOL_NAME(sys_dup2)
.long SYMBOL_NAME(sys_getppid)
.long SYMBOL_NAME(sys_getpgrp) /* 65 */
.long SYMBOL_NAME(sys_setsid)
.long SYMBOL_NAME(sys_sigaction)
.long SYMBOL_NAME(sys_sgetmask)
.long SYMBOL_NAME(sys_ssetmask)
.long SYMBOL_NAME(sys_setreuid16) /* 70 */
.long SYMBOL_NAME(sys_setregid16)
.long SYMBOL_NAME(sys_sigsuspend)
.long SYMBOL_NAME(sys_sigpending)
.long SYMBOL_NAME(sys_sethostname)
.long SYMBOL_NAME(sys_setrlimit) /* 75 */
.long SYMBOL_NAME(sys_old_getrlimit)
.long SYMBOL_NAME(sys_getrusage)
.long SYMBOL_NAME(sys_gettimeofday)
.long SYMBOL_NAME(sys_settimeofday)
.long SYMBOL_NAME(sys_getgroups16) /* 80 */
.long SYMBOL_NAME(sys_setgroups16)
.long SYMBOL_NAME(old_select)
.long SYMBOL_NAME(sys_symlink)
.long SYMBOL_NAME(sys_lstat)
.long SYMBOL_NAME(sys_readlink) /* 85 */
.long SYMBOL_NAME(sys_uselib)
.long SYMBOL_NAME(sys_swapon)
.long SYMBOL_NAME(sys_reboot)
.long SYMBOL_NAME(old_readdir)
.long SYMBOL_NAME(old_mmap) /* 90 */
.long SYMBOL_NAME(sys_munmap)
.long SYMBOL_NAME(sys_truncate)
.long SYMBOL_NAME(sys_ftruncate)
```

```
.long SYMBOL_NAME(sys_fchmod)
.long SYMBOL_NAME(sys_fchown16) /* 95 */
.long SYMBOL_NAME(sys_getpriority)
.long SYMBOL_NAME(sys_setpriority)
.long SYMBOL_NAME(sys_ni_syscall) /* old profil syscall holder */
.long SYMBOL_NAME(sys_statfs)
.long SYMBOL_NAME(sys_fstatfs) /* 100 */
.long SYMBOL_NAME(sys_ioperm)
.long SYMBOL_NAME(sys_socketcall)
.long SYMBOL_NAME(sys_syslog)
.long SYMBOL_NAME(sys_setitimer)
.long SYMBOL_NAME(sys_getitimer) /* 105 */
.long SYMBOL_NAME(sys_newstat)
.long SYMBOL_NAME(sys_newlstat)
.long SYMBOL_NAME(sys_newfstat)
.long SYMBOL_NAME(sys_uname)
.long SYMBOL_NAME(sys_iopl) /* 110 */
.long SYMBOL_NAME(sys_vhangup)
.long SYMBOL_NAME(sys_ni_syscall) /* old "idle" system call */
.long SYMBOL_NAME(sys_vm86old)
.long SYMBOL_NAME(sys_wait4)
.long SYMBOL_NAME(sys_swapoff) /* 115 */
.long SYMBOL_NAME(sys_sysinfo)
.long SYMBOL_NAME(sys_ipc)
.long SYMBOL_NAME(sys_fsync)
.long SYMBOL_NAME(sys_sigreturn)
.long SYMBOL_NAME(sys_clone) /* 120 */
.long SYMBOL_NAME(sys_setdomainname)
.long SYMBOL_NAME(sys_newuname)
.long SYMBOL_NAME(sys_modify_ldt)
.long SYMBOL_NAME(sys_adjtimex)
.long SYMBOL_NAME(sys_mprotect) /* 125 */
.long SYMBOL_NAME(sys_sigprocmask)
.long SYMBOL_NAME(sys_create_module)
.long SYMBOL_NAME(sys_init_module)
.long SYMBOL_NAME(sys_delete_module)
.long SYMBOL_NAME(sys_get_kernel_syms) /* 130 */
.long SYMBOL_NAME(sys_quotactl)
.long SYMBOL_NAME(sys_getpgid)
.long SYMBOL_NAME(sys_fchdir)
.long SYMBOL_NAME(sys_bdflush)
.long SYMBOL_NAME(sys_sysfs) /* 135 */
.long SYMBOL_NAME(sys_personality)
.long SYMBOL_NAME(sys_ni_syscall) /* for afs_syscall */
.long SYMBOL_NAME(sys_setfsuid16)
.long SYMBOL_NAME(sys_setfsgid16)
```

```
.long SYMBOL_NAME(sys_llseek) /* 140 */
.long SYMBOL_NAME(sys_getdents)
.long SYMBOL_NAME(sys_select)
.long SYMBOL_NAME(sys_flock)
.long SYMBOL_NAME(sys_msync)
.long SYMBOL_NAME(sys_readv) /* 145 */
.long SYMBOL_NAME(sys_writev)
.long SYMBOL_NAME(sys_getsid)
.long SYMBOL_NAME(sys_fdatasync)
.long SYMBOL_NAME(sys_sysctl)
.long SYMBOL_NAME(sys_mlock) /* 150 */
.long SYMBOL_NAME(sys_munlock)
.long SYMBOL_NAME(sys_mlockall)
.long SYMBOL_NAME(sys_munlockall)
.long SYMBOL_NAME(sys_sched_setparam)
.long SYMBOL_NAME(sys_sched_getparam)   /* 155 */
.long SYMBOL_NAME(sys_sched_setscheduler)
.long SYMBOL_NAME(sys_sched_getscheduler)
.long SYMBOL_NAME(sys_sched_yield)
.long SYMBOL_NAME(sys_sched_get_priority_max)
.long SYMBOL_NAME(sys_sched_get_priority_min)  /* 160 */
.long SYMBOL_NAME(sys_sched_rr_get_interval)
.long SYMBOL_NAME(sys_nanosleep)
.long SYMBOL_NAME(sys_mremap)
.long SYMBOL_NAME(sys_setresuid16)
.long SYMBOL_NAME(sys_getresuid16) /* 165 */
.long SYMBOL_NAME(sys_vm86)
.long SYMBOL_NAME(sys_query_module)
.long SYMBOL_NAME(sys_poll)
.long SYMBOL_NAME(sys_nfsservctl)
.long SYMBOL_NAME(sys_setresgid16) /* 170 */
.long SYMBOL_NAME(sys_getresgid16)
.long SYMBOL_NAME(sys_prctl)
.long SYMBOL_NAME(sys_rt_sigreturn)
.long SYMBOL_NAME(sys_rt_sigaction)
.long SYMBOL_NAME(sys_rt_sigprocmask) /* 175 */
.long SYMBOL_NAME(sys_rt_sigpending)
.long SYMBOL_NAME(sys_rt_sigtimedwait)
.long SYMBOL_NAME(sys_rt_sigqueueinfo)
.long SYMBOL_NAME(sys_rt_sigsuspend)
.long SYMBOL_NAME(sys_pread) /* 180 */
.long SYMBOL_NAME(sys_pwrite)
.long SYMBOL_NAME(sys_chown16)
.long SYMBOL_NAME(sys_getcwd)
.long SYMBOL_NAME(sys_capget)
.long SYMBOL_NAME(sys_capset)           /* 185 */
```

```
.long SYMBOL_NAME(sys_sigaltstack)
.long SYMBOL_NAME(sys_sendfile)
.long SYMBOL_NAME(sys_ni_syscall) /* streams1 */
.long SYMBOL_NAME(sys_ni_syscall) /* streams2 */
.long SYMBOL_NAME(sys_vfork)            /* 190 */
.long SYMBOL_NAME(sys_getrlimit)
.long SYMBOL_NAME(sys_mmap2)
.long SYMBOL_NAME(sys_truncate64)
.long SYMBOL_NAME(sys_ftruncate64)
.long SYMBOL_NAME(sys_stat64) /* 195 */
.long SYMBOL_NAME(sys_lstat64)
.long SYMBOL_NAME(sys_fstat64)
.long SYMBOL_NAME(sys_lchown)
.long SYMBOL_NAME(sys_getuid)
.long SYMBOL_NAME(sys_getgid) /* 200 */
.long SYMBOL_NAME(sys_geteuid)
.long SYMBOL_NAME(sys_getegid)
.long SYMBOL_NAME(sys_setreuid)
.long SYMBOL_NAME(sys_setregid)
.long SYMBOL_NAME(sys_getgroups) /* 205 */
.long SYMBOL_NAME(sys_setgroups)
.long SYMBOL_NAME(sys_fchown)
.long SYMBOL_NAME(sys_setresuid)
.long SYMBOL_NAME(sys_getresuid)
.long SYMBOL_NAME(sys_setresgid) /* 210 */
.long SYMBOL_NAME(sys_getresgid)
.long SYMBOL_NAME(sys_chown)
.long SYMBOL_NAME(sys_setuid)
.long SYMBOL_NAME(sys_setgid)
.long SYMBOL_NAME(sys_setfsuid) /* 215 */
.long SYMBOL_NAME(sys_setfsgid)
.long SYMBOL_NAME(sys_pivot_root)
.long SYMBOL_NAME(sys_mincore)
.long SYMBOL_NAME(sys_madvise)
.long SYMBOL_NAME(sys_getdents64) /* 220 */
.long SYMBOL_NAME(sys_fcntl64)
#ifdef CONFIG_TUX
.long SYMBOL_NAME(__sys_tux)
#else
# ifdef CONFIG_TUX_MODULE
.long SYMBOL_NAME(sys_tux)
# else
.long SYMBOL_NAME(sys_ni_syscall)
# endif
#endif
.long SYMBOL_NAME(sys_ni_syscall) /* Reserved for Security */
```

```
.long SYMBOL_NAME(sys_gettid)
.long SYMBOL_NAME(sys_readahead) /* 225 */
.long SYMBOL_NAME(sys_setxattr)
.long SYMBOL_NAME(sys_lsetxattr)
.long SYMBOL_NAME(sys_fsetxattr)
.long SYMBOL_NAME(sys_getxattr)
.long SYMBOL_NAME(sys_lgetxattr) /* 230 */
.long SYMBOL_NAME(sys_fgetxattr)
.long SYMBOL_NAME(sys_listxattr)
.long SYMBOL_NAME(sys_llistxattr)
.long SYMBOL_NAME(sys_flistxattr)
.long SYMBOL_NAME(sys_removexattr) /* 235 */
.long SYMBOL_NAME(sys_lremovexattr)
.long SYMBOL_NAME(sys_fremovexattr)
  .long SYMBOL_NAME(sys_tkill)
.long SYMBOL_NAME(sys_sendfile64) /* reserved for sendfile64 */
.long SYMBOL_NAME(sys_futex) /* 240 */
.long SYMBOL_NAME(sys_sched_setaffinity)
.long SYMBOL_NAME(sys_sched_getaffinity)
.long SYMBOL_NAME(sys_set_thread_area)
.long SYMBOL_NAME(sys_get_thread_area)
.long SYMBOL_NAME(sys_ni_syscall) /* 245 sys_io_setup */
.long SYMBOL_NAME(sys_ni_syscall) /* sys_io_destroy */
.long SYMBOL_NAME(sys_ni_syscall) /* sys_io_getevents */
.long SYMBOL_NAME(sys_ni_syscall) /* sys_io_submit */
.long SYMBOL_NAME(sys_ni_syscall) /* sys_io_cancel */
.long SYMBOL_NAME(sys_ni_syscall) /* 250 sys_alloc_hugepages */
.long SYMBOL_NAME(sys_ni_syscall) /* sys_free_hugepages */
.long SYMBOL_NAME(sys_exit_group)
.long SYMBOL_NAME(sys_lookup_dcookie)
.long SYMBOL_NAME(sys_ni_syscall)
.long SYMBOL_NAME(sys_ni_syscall) /* 255 sys_epoll_ctl */
.long SYMBOL_NAME(sys_ni_syscall) /* sys_epoll_wait */
.long SYMBOL_NAME(sys_ni_syscall) /* sys_remap_file_pages */
.long SYMBOL_NAME(sys_set_tid_address)


.rept NR_syscalls-(.-sys_call_table)/4
.long SYMBOL_NAME(sys_ni_syscall)
.endr
```

5.5 open.c [10]

```
#include <linux/string.h>
#include <linux/mm.h>
#include <linux/utime.h>
```

```
#include <linux/file.h>
#include <linux/smp_lock.h>
#include <linux/quotaops.h>
#include <linux/dnotify.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/tty.h>
#include <linux/iobuf.h>

#include <asm/uaccess.h>

#define special_file(m) (S_ISCHR(m)||S_ISBLK(m)||S_ISFIFO(m)||S_ISSOCK(m))

int vfs_statfs(struct super_block *sb, struct statfs *buf)
{
int retval = -ENODEV;

if (sb) {
retval = -ENOSYS;
if (sb->s_op && sb->s_op->statfs) {
memset(buf, 0, sizeof(struct statfs));
lock_kernel();
retval = sb->s_op->statfs(sb, buf);
unlock_kernel();
}
}
return retval;
}


asmlinkage long sys_statfs(const char * path, struct statfs * buf)
{
struct nameidata nd;
int error;

error = user_path_walk(path, &nd);
if (!error) {
struct statfs tmp;
error = vfs_statfs(nd.dentry->d_inode->i_sb, &tmp);
if (!error && copy_to_user(buf, &tmp, sizeof(struct statfs)))
error = -EFAULT;
path_release(&nd);
}
return error;
}
```

```
asmlinkage long sys_fstatfs(unsigned int fd, struct statfs * buf)
{
struct file * file;
struct statfs tmp;
int error;

error = -EBADF;
file = fget(fd);
if (!file)
goto out;
error = vfs_statfs(file->f_dentry->d_inode->i_sb, &tmp);
if (!error && copy_to_user(buf, &tmp, sizeof(struct statfs)))
error = -EFAULT;
fput(file);
out:
return error;
}

/*
 * Install a file pointer in the fd array.
 *
 * The VFS is full of places where we drop the files lock between
 * setting the open_fds bitmap and installing the file in the file
 * array.  At any such point, we are vulnerable to a dup2() race
 * installing a file in the array before us.  We need to detect this and
 * fput() the struct file we are about to overwrite in this case.
 *
 * It should never happen - if we allow dup2() do it, _really_ bad things
 * will follow.
 */

void fd_install(unsigned int fd, struct file * file)
{
struct files_struct *files = current->files;

write_lock(&files->file_lock);
if (files->fd[fd])
BUG();
files->fd[fd] = file;
write_unlock(&files->file_lock);
}

int do_truncate(struct dentry *dentry, loff_t length)
{
struct inode *inode = dentry->d_inode;
int error;
```

```
struct iattr newattrs;

/* Not pretty: "inode->i_size" shouldn't really be signed. But it is. */
if (length < 0)
return -EINVAL;

down(&inode->i_sem);
newattrs.ia_size = length;
newattrs.ia_valid = ATTR_SIZE | ATTR_CTIME;
error = notify_change(dentry, &newattrs);
up(&inode->i_sem);
return error;
}

static inline long do_sys_truncate(const char * path, loff_t length)
{
struct nameidata nd;
struct inode * inode;
int error;

error = -EINVAL;
if (length < 0) /* sorry, but loff_t says... */
goto out;

error = user_path_walk(path, &nd);
if (error)
goto out;
inode = nd.dentry->d_inode;

/* For directories it's -EISDIR, for other non-regulars - -EINVAL */
error = -EISDIR;
if (S_ISDIR(inode->i_mode))
goto dput_and_out;

error = -EINVAL;
if (!S_ISREG(inode->i_mode))
goto dput_and_out;

error = permission(inode,MAY_WRITE);
if (error)
goto dput_and_out;

error = -EROFS;
if (IS_RDONLY(inode))
goto dput_and_out;
```

```
error = -EPERM;
if (IS_IMMUTABLE(inode) || IS_APPEND(inode))
goto dput_and_out;

/*
 * Make sure that there are no leases.
 */
error = get_lease(inode, FMODE_WRITE);
if (error)
goto dput_and_out;

error = get_write_access(inode);
if (error)
goto dput_and_out;

error = locks_verify_truncate(inode, NULL, length);
if (!error) {
DQUOT_INIT(inode);
error = do_truncate(nd.dentry, length);
}
put_write_access(inode);

dput_and_out:
path_release(&nd);
out:
return error;
}

asmlinkage long sys_truncate(const char * path, unsigned long length)
{
/* on 32-bit boxen it will cut the range 2^31--2^32-1 off */
return do_sys_truncate(path, (long)length);
}

static inline long do_sys_ftruncate(unsigned int fd, loff_t length, int small)
{
struct inode * inode;
struct dentry *dentry;
struct file * file;
int error;

error = -EINVAL;
if (length < 0)
goto out;
error = -EBADF;
file = fget(fd);
```

```
if (!file)
goto out;

/* explicitly opened as large or we are on 64-bit box */
if (file->f_flags & O_LARGEFILE)
small = 0;

dentry = file->f_dentry;
inode = dentry->d_inode;
error = -EINVAL;
if (!S_ISREG(inode->i_mode) || !(file->f_mode & FMODE_WRITE))
goto out_putf;

error = -EINVAL;
/* Cannot ftruncate over 2^31 bytes without large file support */
if (small && length > MAX_NON_LFS)
goto out_putf;

error = -EPERM;
if (IS_APPEND(inode))
goto out_putf;

error = locks_verify_truncate(inode, file, length);
if (!error)
error = do_truncate(dentry, length);
out_putf:
fput(file);
out:
return error;
}

asmlinkage long sys_ftruncate(unsigned int fd, unsigned long length)
{
return do_sys_ftruncate(fd, length, 1);
}

/* LFS versions of truncate are only needed on 32 bit machines */
#if BITS_PER_LONG == 32
asmlinkage long sys_truncate64(const char * path, loff_t length)
{
return do_sys_truncate(path, length);
}

asmlinkage long sys_ftruncate64(unsigned int fd, loff_t length)
{
return do_sys_ftruncate(fd, length, 0);
```

```
}
#endif

#if !(defined(__alpha__) || defined(__ia64__))

/*
 * sys_utime() can be implemented in user-level using sys_utimes().
 * Is this for backwards compatibility?  If so, why not move it
 * into the appropriate arch directory (for those architectures that
 * need it).
 */

/* If times==NULL, set access and modification to current time,
 * must be owner or have write permission.
 * Else, update from *times, must be owner or super user.
 */
asmlinkage long sys_utime(char * filename, struct utimbuf * times)
{
int error;
struct nameidata nd;
struct inode * inode;
struct iattr newattrs;

error = user_path_walk(filename, &nd);
if (error)
goto out;
inode = nd.dentry->d_inode;

error = -EROFS;
if (IS_RDONLY(inode))
goto dput_and_out;

/* Don't worry, the checks are done in inode_change_ok() */
newattrs.ia_valid = ATTR_CTIME | ATTR_MTIME | ATTR_ATIME;
if (times) {
error = get_user(newattrs.ia_atime, &times->actime);
if (!error)
error = get_user(newattrs.ia_mtime, &times->modtime);
if (error)
goto dput_and_out;

newattrs.ia_valid |= ATTR_ATIME_SET | ATTR_MTIME_SET;
} else {
if (current->fsuid != inode->i_uid &&
    (error = permission(inode,MAY_WRITE)) != 0)
goto dput_and_out;
```

```
}
error = notify_change(nd.dentry, &newattrs);
dput_and_out:
path_release(&nd);
out:
return error;
}

#endif

/* If times==NULL, set access and modification to current time,
 * must be owner or have write permission.
 * Else, update from *times, must be owner or super user.
 */
asmlinkage long sys_utimes(char * filename, struct timeval * utimes)
{
int error;
struct nameidata nd;
struct inode * inode;
struct iattr newattrs;

error = user_path_walk(filename, &nd);

if (error)
goto out;
inode = nd.dentry->d_inode;

error = -EROFS;
if (IS_RDONLY(inode))
goto dput_and_out;

/* Don't worry, the checks are done in inode_change_ok() */
newattrs.ia_valid = ATTR_CTIME | ATTR_MTIME | ATTR_ATIME;
if (utimes) {
struct timeval times[2];
error = -EFAULT;
if (copy_from_user(&times, utimes, sizeof(times)))
goto dput_and_out;
newattrs.ia_atime = times[0].tv_sec;
newattrs.ia_mtime = times[1].tv_sec;
newattrs.ia_valid |= ATTR_ATIME_SET | ATTR_MTIME_SET;
} else {
if (current->fsuid != inode->i_uid &&
    (error = permission(inode,MAY_WRITE)) != 0)
goto dput_and_out;
}
```

```
error = notify_change(nd.dentry, &newattrs);
dput_and_out:
path_release(&nd);
out:
return error;
}

/*
 * access() needs to use the real uid/gid, not the effective uid/gid.
 * We do this by temporarily clearing all FS-related capabilities and
 * switching the fsuid/fsgid around to the real ones.
 */
asmlinkage long sys_access(const char * filename, int mode)
{
struct nameidata nd;
int old_fsuid, old_fsgid;
kernel_cap_t old_cap;
int res;

if (mode & ~S_IRWXO) /* where's F_OK, X_OK, W_OK, R_OK? */
return -EINVAL;

old_fsuid = current->fsuid;
old_fsgid = current->fsgid;
old_cap = current->cap_effective;

current->fsuid = current->uid;
current->fsgid = current->gid;

/* Clear the capabilities if we switch to a non-root user */
if (current->uid)
cap_clear(current->cap_effective);
else
current->cap_effective = current->cap_permitted;

res = user_path_walk(filename, &nd);
if (!res) {
res = permission(nd.dentry->d_inode, mode);
/* SuS v2 requires we report a read only fs too */
if(!res && (mode & S_IWOTH) && IS_RDONLY(nd.dentry->d_inode)
   && !special_file(nd.dentry->d_inode->i_mode))
res = -EROFS;
path_release(&nd);
}

current->fsuid = old_fsuid;
```

```
current->fsgid = old_fsgid;
current->cap_effective = old_cap;

return res;
}

asmlinkage long sys_chdir(const char * filename)
{
int error;
struct nameidata nd;

error = __user_walk(filename,LOOKUP_POSITIVE|LOOKUP_FOLLOW|LOOKUP_DIRECTORY,&nd);
if (error)
goto out;

error = permission(nd.dentry->d_inode,MAY_EXEC);
if (error)
goto dput_and_out;

set_fs_pwd(current->fs, nd.mnt, nd.dentry);

dput_and_out:
path_release(&nd);
out:
return error;
}

asmlinkage long sys_fchdir(unsigned int fd)
{
struct file *file;
struct dentry *dentry;
struct inode *inode;
struct vfsmount *mnt;
int error;

error = -EBADF;
file = fget(fd);
if (!file)
goto out;

dentry = file->f_dentry;
mnt = file->f_vfsmnt;
inode = dentry->d_inode;

error = -ENOTDIR;
if (!S_ISDIR(inode->i_mode))
```

```
goto out_putf;

error = permission(inode, MAY_EXEC);
if (!error)
set_fs_pwd(current->fs, mnt, dentry);
out_putf:
fput(file);
out:
return error;
}

asmlinkage long sys_chroot(const char * filename)
{
int error;
struct nameidata nd;

error = __user_walk(filename, LOOKUP_POSITIVE | LOOKUP_FOLLOW |
      LOOKUP_DIRECTORY | LOOKUP_NOALT, &nd);
if (error)
goto out;

error = permission(nd.dentry->d_inode,MAY_EXEC);
if (error)
goto dput_and_out;

error = -EPERM;
if (!capable(CAP_SYS_CHROOT))
goto dput_and_out;

set_fs_root(current->fs, nd.mnt, nd.dentry);
set_fs_altroot();
error = 0;
dput_and_out:
path_release(&nd);
out:
return error;
}

asmlinkage long sys_fchmod(unsigned int fd, mode_t mode)
{
struct inode * inode;
struct dentry * dentry;
struct file * file;
int err = -EBADF;
struct iattr newattrs;
```

```
file = fget(fd);
if (!file)
goto out;

dentry = file->f_dentry;
inode = dentry->d_inode;

err = -EROFS;
if (IS_RDONLY(inode))
goto out_putf;
err = -EPERM;
if (IS_IMMUTABLE(inode) || IS_APPEND(inode))
goto out_putf;
if (mode == (mode_t) -1)
mode = inode->i_mode;
newattrs.ia_mode = (mode & S_IALLUGO) | (inode->i_mode & ~S_IALLUGO);
newattrs.ia_valid = ATTR_MODE | ATTR_CTIME;
err = notify_change(dentry, &newattrs);

out_putf:
fput(file);
out:
return err;
}

asmlinkage long sys_chmod(const char * filename, mode_t mode)
{
struct nameidata nd;
struct inode * inode;
int error;
struct iattr newattrs;

error = user_path_walk(filename, &nd);
if (error)
goto out;
inode = nd.dentry->d_inode;

error = -EROFS;
if (IS_RDONLY(inode))
goto dput_and_out;

error = -EPERM;
if (IS_IMMUTABLE(inode) || IS_APPEND(inode))
goto dput_and_out;

if (mode == (mode_t) -1)
```

```
mode = inode->i_mode;
newattrs.ia_mode = (mode & S_IALLUGO) | (inode->i_mode & ~S_IALLUGO);
newattrs.ia_valid = ATTR_MODE | ATTR_CTIME;
error = notify_change(nd.dentry, &newattrs);

dput_and_out:
path_release(&nd);
out:
return error;
}

static int chown_common(struct dentry * dentry, uid_t user, gid_t group)
{
struct inode * inode;
int error;
struct iattr newattrs;

error = -ENOENT;
if (!(inode = dentry->d_inode)) {
printk(KERN_ERR "chown_common: NULL inode\n");
goto out;
}
error = -EROFS;
if (IS_RDONLY(inode))
goto out;
error = -EPERM;
if (IS_IMMUTABLE(inode) || IS_APPEND(inode))
goto out;
if (user == (uid_t) -1)
user = inode->i_uid;
if (group == (gid_t) -1)
group = inode->i_gid;
newattrs.ia_mode = inode->i_mode;
newattrs.ia_uid = user;
newattrs.ia_gid = group;
newattrs.ia_valid =  ATTR_UID | ATTR_GID | ATTR_CTIME;
/*
 * If the user or group of a non-directory has been changed by a
 * non-root user, remove the setuid bit.
 * 19981026 David C Niemi <niemi@tux.org>
 *
 * Changed this to apply to all users, including root, to avoid
 * some races. This is the behavior we had in 2.0. The check for
 * non-root was definitely wrong for 2.2 anyway, as it should
 * have been using CAP_FSETID rather than fsuid -- 19990830 SD.
 */
```

```
if ((inode->i_mode & S_ISUID) == S_ISUID &&
!S_ISDIR(inode->i_mode))
{
newattrs.ia_mode &= ~S_ISUID;
newattrs.ia_valid |= ATTR_MODE;
}
/*
 * Likewise, if the user or group of a non-directory has been changed
 * by a non-root user, remove the setgid bit UNLESS there is no group
 * execute bit (this would be a file marked for mandatory locking).
 * 19981026 David C Niemi <niemi@tux.org>
 *
 * Removed the fsuid check (see the comment above) -- 19990830 SD.
 */
if (((inode->i_mode & (S_ISGID | S_IXGRP)) == (S_ISGID | S_IXGRP))
&& !S_ISDIR(inode->i_mode))
{
newattrs.ia_mode &= ~S_ISGID;
newattrs.ia_valid |= ATTR_MODE;
}
error = notify_change(dentry, &newattrs);
out:
return error;
}

asmlinkage long sys_chown(const char * filename, uid_t user, gid_t group)
{
struct nameidata nd;
int error;

error = user_path_walk(filename, &nd);
if (!error) {
error = chown_common(nd.dentry, user, group);
path_release(&nd);
}
return error;
}

asmlinkage long sys_lchown(const char * filename, uid_t user, gid_t group)
{
struct nameidata nd;
int error;

error = user_path_walk_link(filename, &nd);
if (!error) {
error = chown_common(nd.dentry, user, group);
```

```
path_release(&nd);
}
return error;
}


asmlinkage long sys_fchown(unsigned int fd, uid_t user, gid_t group)
{
struct file * file;
int error = -EBADF;

file = fget(fd);
if (file) {
error = chown_common(file->f_dentry, user, group);
fput(file);
}
return error;
}

/*
 * Note that while the flag value (low two bits) for sys_open means:
 * 00 - read-only
 * 01 - write-only
 * 10 - read-write
 * 11 - special
 * it is changed into
 * 00 - no permissions needed
 * 01 - read-permission
 * 10 - write-permission
 * 11 - read-write
 * for the internal routines (ie open_namei()/follow_link() etc). 00 is
 * used by symlinks.
 */
struct file *filp_open(const char * filename, int flags, int mode)
{
int namei_flags, error;
struct nameidata nd;

flags &= ~O_DIRECT;

namei_flags = flags;
if ((namei_flags+1) & O_ACCMODE)
namei_flags++;
if (namei_flags & O_TRUNC)
namei_flags |= 2;
```

```
error = open_namei(filename, namei_flags, mode, &nd);
if (!error)
return dentry_open(nd.dentry, nd.mnt, flags);

return ERR_PTR(error);
}

struct file *dentry_open(struct dentry *dentry, struct vfsmount *mnt, int flags)
{
struct file * f;
struct inode *inode;
static LIST_HEAD(kill_list);
int error;

error = -ENFILE;
f = get_empty_filp();
if (!f)
goto cleanup_dentry;
f->f_flags = flags;
f->f_mode = (flags+1) & O_ACCMODE;
inode = dentry->d_inode;
if (f->f_mode & FMODE_WRITE) {
error = get_write_access(inode);
if (error)
goto cleanup_file;
}

f->f_dentry = dentry;
f->f_vfsmnt = mnt;
f->f_pos = 0;
f->f_reada = 0;
f->f_op = fops_get(inode->i_fop);
file_move(f, &inode->i_sb->s_files);

/* preallocate kiobuf for O_DIRECT */
f->f_iobuf = NULL;
f->f_iobuf_lock = 0;
if (f->f_flags & O_DIRECT) {
error = alloc_kiovec(1, &f->f_iobuf);
if (error)
goto cleanup_all;
}

if (f->f_op && f->f_op->open) {
error = f->f_op->open(inode,f);
if (error)
```

```
goto cleanup_all;
}
f->f_flags &= ~(O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);

return f;

cleanup_all:
if (f->f_iobuf)
free_kiovec(1, &f->f_iobuf);
fops_put(f->f_op);
if (f->f_mode & FMODE_WRITE)
put_write_access(inode);
file_move(f, &kill_list); /* out of the way.. */
f->f_dentry = NULL;
f->f_vfsmnt = NULL;
cleanup_file:
put_filp(f);
cleanup_dentry:
dput(dentry);
mntput(mnt);
return ERR_PTR(error);
}

/*
 * Find an empty file descriptor entry, and mark it busy.
 */
int get_unused_fd(void)
{
struct files_struct * files = current->files;
int fd, error;

   error = -EMFILE;
write_lock(&files->file_lock);

repeat:
  fd = find_next_zero_bit(files->open_fds,
files->max_fdset,
files->next_fd);

/*
 * N.B. For clone tasks sharing a files structure, this test
 * will limit the total number of files that can be opened.
 */
if (fd >= current->rlim[RLIMIT_NOFILE].rlim_cur)
goto out;
```

```
/* Do we need to expand the fdset array? */
if (fd >= files->max_fdset) {
error = expand_fdset(files, fd);
if (!error) {
error = -EMFILE;
goto repeat;
}
goto out;
}

/*
 * Check whether we need to expand the fd array.
 */
if (fd >= files->max_fds) {
error = expand_fd_array(files, fd);
if (!error) {
error = -EMFILE;
goto repeat;
}
goto out;
}

FD_SET(fd, files->open_fds);
FD_CLR(fd, files->close_on_exec);
files->next_fd = fd + 1;
#if 1
/* Sanity check */
if (files->fd[fd] != NULL) {
printk(KERN_WARNING "get_unused_fd: slot %d not NULL!\n", fd);
files->fd[fd] = NULL;
}
#endif
error = fd;

out:
write_unlock(&files->file_lock);
return error;
}

asmlinkage long sys_open(const char * filename, int flags, int mode)
{
char * tmp;
int fd, error;

#if BITS_PER_LONG != 32
flags |= O_LARGEFILE;
```

```
#endif
tmp = getname(filename);
fd = PTR_ERR(tmp);
if (!IS_ERR(tmp)) {
fd = get_unused_fd();
if (fd >= 0) {
struct file *f = filp_open(tmp, flags, mode);
error = PTR_ERR(f);
if (IS_ERR(f))
goto out_error;
fd_install(fd, f);
}
out:
putname(tmp);
}
return fd;

out_error:
put_unused_fd(fd);
fd = error;
goto out;
}

#ifndef __alpha__

/*
 * For backward compatibility?  Maybe this should be moved
 * into arch/i386 instead?
 */
asmlinkage long sys_creat(const char * pathname, int mode)
{
return sys_open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
}

#endif

/*
 * "id" is the POSIX thread ID. We use the
 * files pointer for this..
 */
int filp_close(struct file *filp, fl_owner_t id)
{
int retval;

if (!file_count(filp)) {
printk(KERN_ERR "VFS: Close: file count is 0\n");
```

```
return 0;
}
retval = 0;
if (filp->f_op && filp->f_op->flush) {
lock_kernel();
retval = filp->f_op->flush(filp);
unlock_kernel();
}
dnotify_flush(filp, id);
locks_remove_posix(filp, id);
fput(filp);
return retval;
}

/*
 * Careful here! We test whether the file pointer is NULL before
 * releasing the fd. This ensures that one clone task can't release
 * an fd while another clone is opening it.
 */
asmlinkage long sys_close(unsigned int fd)
{
struct file * filp;
struct files_struct *files = current->files;

write_lock(&files->file_lock);
if (fd >= files->max_fds)
goto out_unlock;
filp = files->fd[fd];
if (!filp)
goto out_unlock;
files->fd[fd] = NULL;
FD_CLR(fd, files->close_on_exec);
__put_unused_fd(files, fd);
write_unlock(&files->file_lock);
return filp_close(filp, files);

out_unlock:
write_unlock(&files->file_lock);
return -EBADF;
}

/*
 * This routine simulates a hangup on the tty, to arrange that users
 * are given clean terminals at login time.
 */
asmlinkage long sys_vhangup(void)
```

```
{
if (capable(CAP_SYS_TTY_CONFIG)) {
tty_vhangup(current->tty);
return 0;
}
return -EPERM;
}

/*
 * Called when an inode is about to be open.
 * We use this to disallow opening RW large files on 32bit systems if
 * the caller didn't specify O_LARGEFILE.  On 64bit systems we force
 * on this flag in sys_open.
 */
int generic_file_open(struct inode * inode, struct file * filp)
{
if (!(filp->f_flags & O_LARGEFILE) && inode->i_size > MAX_NON_LFS)
return -EFBIG;
return 0;
}

EXPORT_SYMBOL(generic_file_open);
```

6 Data Structures

6.1 The dentry struct[15]

```
struct dentry {
atomic_t d_count;
unsigned int d_flags;
struct inode  * d_inode; /* Where the name belongs to - NULL is negative */
struct dentry * d_parent; /* parent directory */
struct list_head d_hash; /* lookup hash list */
struct list_head d_lru; /* d_count = 0 LRU list */
struct list_head d_child; /* child of parent list */
struct list_head d_subdirs; /* our children */
struct list_head d_alias; /* inode alias list */
int d_mounted;
struct qstr d_name;
unsigned long d_time; /* used by d_revalidate */
struct dentry_operations  *d_op;
struct super_block * d_sb; /* The root of the dentry tree */
unsigned long d_vfs_flags;
void * d_fsdata; /* fs-specific data */
void * d_extra_attributes; /* TUX-specific data */
```

```
unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
struct dcookie_struct * d_cookie; /* cookie, if any */
};
```

6.2 The atomic_t struct[16]

Make sure gcc doesn't try to be clever and move things around on us. We need to use exactly the address the user gave us, not some alias that contains the same information.

```
typedef struct { volatile int counter; } atomic_t;
```

6.3 The inode struct[13]

```
struct inode {
struct list_head i_hash;
struct list_head i_list;
struct list_head i_dentry;

struct list_head i_dirty_buffers;
struct list_head i_dirty_data_buffers;

unsigned long i_ino;
atomic_t i_count;
kdev_t i_dev;
umode_t i_mode;
unsigned int i_nlink;
uid_t i_uid;
gid_t i_gid;
kdev_t i_rdev;
loff_t i_size;
time_t i_atime;
time_t i_mtime;
time_t i_ctime;
unsigned int i_blkbits;
unsigned long i_blksize;
unsigned long i_blocks;
unsigned long i_version;
unsigned short i_bytes;
struct semaphore i_sem;
struct semaphore i_zombie;
struct inode_operations *i_op;
struct file_operations *i_fop; /* former ->i_op->default_file_ops */
struct super_block *i_sb;
wait_queue_head_t i_wait;
struct file_lock *i_flock;
```

```
struct address_space *i_mapping;
struct address_space i_data;
struct dquot *i_dquot[MAXQUOTAS];
/* These three should probably be a union */
struct list_head i_devices;
struct pipe_inode_info *i_pipe;
struct block_device *i_bdev;
struct char_device *i_cdev;

unsigned long i_dnotify_mask; /* Directory notify events */
struct dnotify_struct *i_dnotify; /* for directory notifications */

unsigned long i_state;

unsigned int i_flags;
unsigned char i_sock;

atomic_t i_writecount;
unsigned int i_attr_flags;
__u32 i_generation;
union {
struct minix_inode_info minix_i;
struct ext2_inode_info ext2_i;
struct ext3_inode_info ext3_i;
struct hpfs_inode_info hpfs_i;
struct ntfs_inode_info ntfs_i;
struct msdos_inode_info msdos_i;
struct umsdos_inode_info umsdos_i;
struct iso_inode_info isofs_i;
struct nfs_inode_info nfs_i;
struct sysv_inode_info sysv_i;
struct affs_inode_info affs_i;
struct ufs_inode_info ufs_i;
struct efs_inode_info efs_i;
struct romfs_inode_info romfs_i;
struct shmem_inode_info shmem_i;
struct coda_inode_info coda_i;
struct smb_inode_info smbfs_i;
struct hfs_inode_info hfs_i;
struct adfs_inode_info adfs_i;
struct qnx4_inode_info qnx4_i;
struct reiserfs_inode_info reiserfs_i;
struct bfs_inode_info bfs_i;
struct udf_inode_info udf_i;
struct ncp_inode_info ncpfs_i;
struct proc_inode_info proc_i;
```

```
struct socket socket_i;
struct usbdev_inode_info        usbdev_i;
struct jffs2_inode_info jffs2_i;
void *generic_ip;
} u;
};
```

6.4 The list_head struct[17]

```
typedef struct list_head {
struct list_head *next, *prev;
} list_t;
```

6.5 The qstr struct[15]

The "quick string" struct eases parameter passing, but more importantly saves "metadata" about the string (ie length and the hash).

```
struct qstr {
const unsigned char * name;
unsigned int len;
unsigned int hash;
};
```

6.6 The super_block struct[13]

```
struct super_block {
struct list_head s_list; /* Keep this first */
kdev_t s_dev;
unsigned long s_blocksize;
unsigned char s_blocksize_bits;
unsigned char s_dirt;
unsigned long long s_maxbytes; /* Max file size */
struct file_system_type *s_type;
struct super_operations *s_op;
struct dquot_operations *dq_op;
struct quotactl_ops *s_qcop;
unsigned long s_flags;
unsigned long s_magic;
struct dentry *s_root;
struct rw_semaphore s_umount;
struct semaphore s_lock;
int s_count;
atomic_t s_active;

struct list_head s_dirty; /* dirty inodes */
```

```
struct list_head s_locked_inodes;/* inodes being synced */
struct list_head s_files;

struct block_device *s_bdev;
struct list_head s_instances;
  struct quota_info s_dquot; /* Diskquota specific options */

union {
struct minix_sb_info minix_sb;
struct ext2_sb_info ext2_sb;
struct ext3_sb_info ext3_sb;
struct hpfs_sb_info hpfs_sb;
struct ntfs_sb_info ntfs_sb;
struct msdos_sb_info msdos_sb;
struct isofs_sb_info isofs_sb;
struct nfs_sb_info nfs_sb;
struct sysv_sb_info sysv_sb;
struct affs_sb_info affs_sb;
struct ufs_sb_info ufs_sb;
struct efs_sb_info efs_sb;
struct shmem_sb_info shmem_sb;
struct romfs_sb_info romfs_sb;
struct smb_sb_info smbfs_sb;
struct hfs_sb_info hfs_sb;
struct adfs_sb_info adfs_sb;
struct qnx4_sb_info qnx4_sb;
struct reiserfs_sb_info reiserfs_sb;
struct bfs_sb_info bfs_sb;
struct udf_sb_info udf_sb;
struct ncp_sb_info ncpfs_sb;
struct usbdev_sb_info   usbdevfs_sb;
struct jffs2_sb_info jffs2_sb;
struct cramfs_sb_info cramfs_sb;
void *generic_sbp;
} u;
/*
 * The next field is for VFS *only*. No filesystems have any business
 * even looking at it. You had been warned.
 */
struct semaphore s_vfs_rename_sem; /* Kludge */

/* The next field is used by knfsd when converting a (inode number based)
 * file handle into a dentry. As it builds a path in the dcache tree from
 * the bottom up, there may for a time be a subpath of dentrys which is not
 * connected to the main tree.  This semaphore ensure that there is only ever
 * one such free path per filesystem.  Note that unconnected files (or other
```

```
 * non-directories) are allowed, but not unconnected diretories.
 */
struct semaphore s_nfsd_free_path_sem;
};
```

6.7 The dcookie_struct struct[15]

This is null in the RedHat 9 Linux distribution.

```
struct dcookie_struct;
```

6.8 The kdev_t typedef[19]

As a preparation for the introduction of larger device numbers, we introduce a type kdev_t to hold them. No information about this type is known outside of this include file.

Objects of type kdev_t designate a device. Outside of the kernel the corresponding things are objects of type dev_t - usually an integral type with the device major and minor in the high and low bits, respectively. Conversion is done by

```
extern kdev_t to_kdev_t(int);
```

It is up to the various file systems to decide how objects of type dev_t are stored on disk. The only other point of contact between kernel and outside world are the system calls stat and mknod, new versions of which will eventually have to be used in libc.

[Unfortunately, the floppy control ioctls fail to hide the internal kernel structures, and the fd_device field of a struct floppy_drive_struct is user-visible. So, it remains a dev_t for the moment, with some ugly conversions in floppy.c.]

Inside the kernel, we aim for a kdev_t type that is a pointer to a structure with information about the device (like major, minor, size, blocksize, sectorsize, name, read-only flag, struct file_operations etc.).

However, for the time being we let kdev_t be almost the same as dev_t:

```
typedef struct { unsigned short major, minor; } kdev_t;
```

Admissible operations on an object of type kdev_t:

- passing it along
- comparing it for equality with another such object
- storing it in ROOT_DEV, inode-¿i_dev, inode-¿i_rdev, sb-¿s_dev, bh-¿b_dev, req-¿rq_dev, de-¿dc_dev, tty-¿device
- using its bit pattern as argument in a hash function
- finding its major and minor

- complaining about it

An object of type kdev_t is created only by the function MKDEV(), with the single exception of the constant 0 (no device).

Right now the other information mentioned above is usually found in static arrays indexed by major or major,minor.

An obstacle to immediately using

```
typedef struct { ... (* lots of information *) } *kdev_t
```

is the case of mknod used to create a block device that the kernel doesn't know about at present (but first learns about when some module is inserted).

```
typedef unsigned short kdev_t;
```

6.9 The umode_t typedef[20]

```
typedef unsigned short umode_t;
```

6.10 The uid_t typedef[21]

```
typedef __kernel_uid32_t uid_t;
```

6.11 The gid_t typedef[21]

```
typedef __kernel_gid32_t gid_t;
```

6.12 The loff_t typedef[21]

```
typedef __kernel_loff_t loff_t;
```

6.13 The time_t typedef[21]

```
typedef __kernel_time_t time_t;
```

6.14 The semaphore struct[22]

```
struct semaphore {
atomic_t count;
int sleepers;
wait_queue_head_t wait;
#if WAITQUEUE_DEBUG
long __magic;
#endif
};
```

6.15 The wait_queue_head_t typedef[23]

```
typedef struct __wait_queue_head wait_queue_head_t;
```

6.16 The wait_queue_head struct[23]

```
struct __wait_queue_head {
wq_lock_t lock;
struct list_head task_list;
#if WAITQUEUE_DEBUG
long __magic;
long __creator;
#endif
};
```

6.17 The file_lock struct[13]

```
struct file_lock {
struct file_lock *fl_next; /* singly linked list for this inode  */
struct list_head fl_link; /* doubly linked list of all locks */
struct list_head fl_block; /* circular list of blocked processes */
fl_owner_t fl_owner;
unsigned int fl_pid;
wait_queue_head_t fl_wait;
struct file *fl_file;
unsigned char fl_flags;
unsigned char fl_type;
loff_t fl_start;
loff_t fl_end;

void (*fl_notify)(struct file_lock *); /* unblock callback */
void (*fl_insert)(struct file_lock *); /* lock insertion callback */
void (*fl_remove)(struct file_lock *); /* lock removal callback */

struct fasync_struct * fl_fasync; /* for lease break notifications */
unsigned long fl_break_time; /* for nonblocking lease breaks */

union {
struct nfs_lock_info nfs_fl;
} fl_u;
};
```

6.18 The address_space struct[13]

```
struct address_space {
struct list_head clean_pages; /* list of clean pages */
struct list_head dirty_pages; /* list of dirty pages */
struct list_head locked_pages; /* list of locked pages */
unsigned long nrpages; /* number of total pages */
struct address_space_operations *a_ops; /* methods */
```

```
struct inode *host; /* owner: inode, block_device */
struct vm_area_struct *i_mmap; /* list of private mappings */
struct vm_area_struct *i_mmap_shared; /* list of shared mappings */
spinlock_t i_shared_lock;  /* and spinlock protecting it */
int gfp_mask; /* how to allocate the pages */
};
```

6.19 The dquot struct[24]

```
struct dquot {
struct list_head dq_hash; /* Hash list in memory */
struct list_head dq_inuse; /* List of all quotas */
struct list_head dq_free; /* Free list element */
wait_queue_head_t dq_wait_lock; /* Pointer to waitqueue on dquot lock */
wait_queue_head_t dq_wait_free; /* Pointer to waitqueue for quota to be unused */
int dq_count; /* Use count */
int dq_dup_ref; /* Number of duplicated refences */

/* fields after this point are cleared when invalidating */
struct super_block *dq_sb; /* superblock this applies to */
unsigned int dq_id; /* ID this applies to (uid, gid) */
kdev_t dq_dev; /* Device this applies to */
loff_t dq_off; /* Offset of dquot on disk */
short dq_type; /* Type of quota */
short dq_flags; /* See DQ_* */
unsigned long dq_referenced; /* Number of times this dquot was
   referenced during its lifetime */
struct mem_dqblk dq_dqb; /* Diskquota usage */
};
```

6.20 The pipe_inode_info struct[25]

```
struct pipe_inode_info {
wait_queue_head_t wait;
char *base;
unsigned int len;
unsigned int start;
unsigned int readers;
unsigned int writers;
unsigned int waiting_readers;
unsigned int waiting_writers;
unsigned int r_counter;
unsigned int w_counter;
};
```

6.21 The block_device struct[13]

```
struct block_device {
struct list_head bd_hash;
atomic_t bd_count;
struct inode * bd_inode;
dev_t bd_dev;  /* not a kdev_t - it's a search key */
int bd_openers;
const struct block_device_operations *bd_op;
struct semaphore bd_sem; /* open/close mutex */
struct list_head bd_inodes;
};
```

6.22 The char_device struct[13]

```
struct char_device {
struct list_head hash;
atomic_t count;
dev_t dev;
atomic_t openers;
struct semaphore sem;
};
```

6.23 The dnotify_struct struct[26]

```
struct dnotify_struct {
struct dnotify_struct * dn_next;
unsigned long dn_mask; /* Events to be notified
   see linux/fcntl.h */
int dn_fd;
struct file * dn_filp;
fl_owner_t dn_owner;
};
```

6.24 The ext2_inode_info struct[27]

The second extended file system inode data in memory.

```
struct ext2_inode_info {
__u32 i_data[15];
__u32 i_flags;
__u32 i_faddr;
__u8 i_frag_no;
__u8 i_frag_size;
__u32 i_file_acl;
__u32 i_dir_acl;
__u32 i_dtime;
```

```
__u32 i_block_group;
__u32 i_next_alloc_block;
__u32 i_next_alloc_goal;
__u32 i_prealloc_block;
__u32 i_prealloc_count;
__u32 i_dir_start_lookup;
int i_new_inode:1; /* Is a freshly allocated inode */
};
```

6.25 The vfsmount struct[28]

```
struct vfsmount
{
struct list_head mnt_hash;
struct vfsmount *mnt_parent; /* fs we are mounted on */
struct dentry *mnt_mountpoint; /* dentry of mountpoint */
struct dentry *mnt_root; /* root of the mounted tree */
struct super_block *mnt_sb; /* pointer to superblock */
struct list_head mnt_mounts; /* list of children, anchored here */
struct list_head mnt_child; /* and going through their mnt_child */
atomic_t mnt_count;
int mnt_flags;
char *mnt_devname; /* Name of device e.g. /dev/dsk/hda1 */
struct list_head mnt_list;
};
```

References

[1] glibc-2.3.5/sysdeps/generic/chdir.c

[2] glibc-2.3.5/sysdeps/generic/fchdir.c

[3] intel.pamphlet

[4] /usr/src/linux-2.4.20-8/include/asm-i386/unistd.h

[5] /usr/share/man/man2/chdir.2.gz

[6] IA-32 Intel Architecture Software Developer's Manual Volume 2A: Instruction Set Reference, A-M Order Number 253666-016 June 2005,

[7] IA-32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z Order Number 253667-016 June 2005

[8] /usr/include/asm/errno.h

[9] /usr/src/linux-2.4/arch/i386/kernel/entry.S

[10] /usr/src/linux-2.4/fs/open.c

[11] /usr/src/linux-2.4/kernel/ksyms.c

[12] /usr/src/linux-2.4/fs/namei.c

[13] /usr/src/linux-2.4/include/linux/fs.h

[14] /usr/src/linux-2.4/mm/slab.c

[15] /usr/src/linux-2.4/include/linux/dcache.h

[16] /usr/src/linux-2.4/include/asm/atomic.h

[17] /usr/src/linux-2.4/include/linux/list.h

[18] /usr/src/linux-2.4/Documentation/filesystems/vfs.txt

[19] /usr/src/linux-2.4/include/linux/kdev_t.h

[20] /usr/src/linux-2.4/include/asm-i386/types.h

[21] /usr/src/linux-2.4/include/linux/types.h

[22] /usr/src/linux-2.4/include/asm-i386/semaphore.h

[23] /usr/src/linux-2.4/include/linux/wait.h

[24] /usr/src/linux-2.4/include/linux/quota.h

[25] /usr/src/linux-2.4/include/linux/pipe_fs_i.h

[26] /usr/src/linux-2.4/include/linux/dnotify.h

[27] /usr/src/linux-2.4/include/linux/ext2_fs_i.h

[28] /usr/src/linux-2.4/include/linux/mount.h

[29] /usr/src/linux-2.4/include/linux/fs_struct.h

[30] Bonwick, Jeff, "The Slab Allocator: An Object-Caching Kernel Memory Allocator" USENIX Summer 1994 Technical Conference

[31] Vahalia, Uresh, "UNIX Internals: The New Frontiers" Prentice-Hall ISBN 0-13-101908-2

[32] /usr/src/linux-2.4/include/asm-alpha/system.h