Practical Filesystems (A Systems Perspective)
Section I

Introduction to Filesystems
Virtual Filesystem

- serves as a layer between the real filesystem and the userspace applications
- BSD implemented VFS for NFS
- VMS had elaborate filesystem
- NT/Win95 have VFS type interfaces too
- Newer systems integrate VM with buffer cache.
Linux-based Filesystem

★ Media based
  – ext2 - Linux native
  – ufs - BSD
  – fat - DOS FS FAT16
  – vfat - win 95 FAT32
  – hpfs - OS/2
  – minix - well
  – Isofs - CDROM
  – sysv - Sysv Unix
  – hfs - Macintosh
  – affs - Amiga Fast FS
  – NTFS - NTs FS
– adfs - Acorn-strongarm

★ Media based - Journalled
– ext3
– reiserfs
– jfs - from IBM
– xfs - from SGI
– clusterfs - from Oracle

★ Network
– nfs
– Coda
– AFS - Andrew FS
– smbfs - LanManager
– ncpfs - Novell
– CIFS - samba
– Appleshare

★ Special ones

– procfs -/proc
– umsdos - Unix in DOS
– userfs - redirector to user space application
– devfs
– DFS - DCE distributed filesystem
Section II

VFS Concepts
Linux VFS Subsystem

★ Manages kernel level file abstractions in one format for all file systems
★ Receives system call requests from user level (e.g. write, open, stat, link)
★ Interacts with a specific file system based on mount point traversal
★ Receives requests from other parts of the kernel, mostly from memory management
Linux VFS Subsystem

- Multiple interfaces build up VFS:
  - files
  - dentries - directory entries
  - inodes
  - superblock
  - quota

- VFS can do all caching and provides utility fcnns to FS

- FS provides methods to VFS; many are optional
Data Structures

VFS data structures for:

★ VFS handle to the file: inode (BSD: vnode)

★ User instantiated file handle: file (BSD: file)

★ The whole filesystem: superblock (BSD: vfs)

★ A name to inode translation: dentry
Superblock

- Handle metadata only (attributes etc)
- Responsible for retrieving and storing metadata from the FS media or peers
- Struct superblocks hold things like:
  - device, blocksize, dirty flags, list of dirty inodes
  - super operations
  - wait queue
  - pointer to the root inode of this FS
Inode

- Inodes are VFS abstraction for the file
- Inode has operations
- VFS maintains an inode cache, NOT the individual FSs (compare NT, BSD etc)
- Inodes contain an FS specific area where:
  - ext2 stores disk block numbers etc
  - AFS would store the FID
- Extraordinary inode ops are good for dealing with stale NFS file handles etc.
Dentry

★ Dentry is a name to inode translation structure
★ Cached aggressively by VFS
★ Eliminates lookups by FS & private caches
  - timing on Coda FS: ls -IR 1000 files after priming cache
    ★ linux 2.0.32: 7.2secs
    ★ linux 2.1.92: 0.6secs
  - disk fs: less benefit, NFS even more
★ Negative entries!
★ Namei is dramatically simplified
Anatomy of the some system calls

- to better understand the inner workings of a VFS system
- to determine the relationships between the filesystem items
- VFS layer overview
Anatomy of the stat() system call

```c
sys_stat(path, buf) {
    /* obtain dentry from VFS */
dentry = namei(path);
    if ( dentry == NULL ) return -ENOENT;

    /* call into inode layer of FS */
    inode = dentry->d_inode;
    rc = inode->i_op->i_permission(inode);
    if ( rc ) return -EPERM;

    /* call into inode layer of FS */
    rc = inode->i_op->i_getattr(inode, buf);
    dput(dentry);
    return rc;
}
```
Anatomy of the fstatfs() system call

/* for things like df */
sys_fstatfs(fd, buf) {
    /* translate fd to VFS structure */
    file = fget(fd);
    if ( file == NULL ) return -EBADF;

    /* call superblock layer for FS */
    superb = file->f_dentry->d_inode->i_super;

    /* get information from superblock */
    rc = superb->sb_op->sb_statfs(sb, buf);
    return rc;
}
Section III

Filesystem I/O
Typical userlevel file access

★ pathnames: /myfile
★ file descriptors: fd = open(/myfile)
★ attributes in struct stat: stat(/myfile, &mybuf), chmod, chown...
★ offsets: write, read, lseek
★ directory handles: DIR *dh = opendir(/mydir)
★ directory entries: struct dirent *ent = readdir(dh)
Unbuffered I/O

- kernel open files and assigns a *File Descriptor* to each file
- the operating system provides system calls in order to manipulate these *file descriptors*
- the syntax of these function calls are defined by the IEEE Portable Operating System Interface for Computing Environments (POSIX 1003.1) standard
- a POSIX operating system provides three basic file descriptors:
  - `STDIN_FILENO 0`
  - `STDOUT_FILENO 1`
  - `STDERR_FILENO 2`
- most of these operations are atomic operations (considered as a single execution event and not a combination of different events)
Unbuffered I/O: Open and Close

* header files for these functions are:
  - sys/types.h
  - sys/stat.h
  - fcntl.h
  - unistd.h
* int open (const char *pathname, int oflag);
* int open (const char *pathname, int oflag, mode_t mode);
  - returns the file descriptor of the opened file and returns -1 if an error has occurred
  - oflag is a combination of macros that define the properties of an open file
* O_RDONLY - open file for read only
* `O_WRONLY` - open file for write only
* `O_RDWR` - open file for read and write access
* `O_APPEND` - open file and move file pointer to the end of file
* `O_CREAT` - open file and truncate size to zero. require the version of open with the `mode` parameter which determines the file's access permissions
* `O_EXCL` - used in conjunction with the `O_CREAT` and returns an error if file exists
* `O_TRUNC` - truncates file size to zero if file is opened for write-only or read-write
* `O_SYNC` - forces each write on this file descriptor to wait for each physical write operation
  - `mode` determines the file permissions of the created file
* `S_IRWXU` - 00700 user (file owner) has read, write and execute permission
* `S_IRUSR` (S_IREAD) - 00400 user has read permission
∗ S_IWUSR (S_IWRITE) - 00200 user has write permission
∗ S_IXUSR (S_IEXEC) - 00100 user has execute permission
∗ S_IRWXG - 00070 group has read, write and execute permission
∗ S_IRGRP - 00040 group has read permission
∗ S_IWGRP - 00020 group has write permission
∗ S_IXGRP - 00010 group has execute permission
∗ S_IRWXO - 00007 others have read, write and execute permission
∗ S_IROTH - 00004 others have read permission
∗ S_IWOTH - 00002 others have write permission
∗ S_IXOTH - 00001 others have execute permission

∗ int creat(const char *pathname, mode_t mode);
   – is equivalent to open with oflags equal to
     O_CREAT | O_WRONLY | O_TRUNC

∗ int close(int fd);
   – closes a file descriptor and returns a -1 if an error occurred
Unbuffered I/O: File Access

* `off_t lseek(int fildes, off_t offset, int whence);`
  - moves the file pointer of the file defined by `fildes` to a value `offset` according to the directive `whence`
  - returns a -1 if an error occurred
  - values for `whence` are typically:
    * `SEEK_SET` - the offset is set to `offset` bytes from the beginning of the file
    * `SEEK_CUR` - the offset is set to its current location plus `offset` bytes.
    * `SEEK_END` - the offset is set to the size of the file plus `offset` bytes.

* `ssize_t read(int fd, void *buf, size_t count);`
* `ssize_t write(int fd, const void *buf, size_t count);`
count);

- basic unbuffered input and output functions respectively
- returns the number of bytes actually read or written
- returns -1 if an error occurred
- read takes count bytes of data from the fd and write the input into buf
- write places count bytes of data to the fd from buf

* int dup(int oldfd);

* int dup2(int oldfd, int newfd);

- create a copy of the file descriptor oldfd
- returns the file descriptor of the copied file descriptor
- returns -1 if an error occurred
- for newfd is closed first if already open
Unbuffered I/O: Manipulating File Descriptors

⋆ int fcntl(int fd, int cmd);
⋆ int fcntl(int fd, int cmd, long arg);
⋆ int fcntl(int fd, int cmd, struct flock *lock);

– general purpose function for manipulating file descriptors
– returns the intended file descriptors or flag values for set functions
– get functions accept an argument
– returns -1 if an error occurred
– performs commands based on the value of cmd argument:
  ⋆ F_DUPFD - accepts a single argument which is the file descriptor to be duplicated
  ⋆ F_GETFD/F_SETFD - get or set the FD_CLOEXEC flag (this flag determines if a descriptor is closed after an exec)
  ⋆ F_GETFL/F_SETFL - get or set the descriptor’s flags (similar to the
flags passed during an open
* F_GETLK/F_SETLK/F_SETLKW - are used to manage file locks.
  There is a third argument called lock of struct flock. The
  F_GETLK unsets the lock while F_SETLK sets it. F_GETLKW is
  similar to F_GETLK, however it waits for the lock to be release instead
  of returning the flock structure.
* F_GETOWN/F_SETOWN - gets and set the process ID or the process
  group that owns the file descriptor. Process are positive value while
  process groups are negative values

* int ioctl(int d, int request, ...);
  – defined in the sys/ioctl.h header file
  – the ducktape of system programming
  – catch all function for input/output
  – requests for this function can be divided into these groups:
<table>
<thead>
<tr>
<th>Category</th>
<th>Constant Names</th>
<th>Header</th>
<th>Number of ioctl operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>disk labels</td>
<td>DIOxxx</td>
<td>disklabel.h</td>
<td>10</td>
</tr>
<tr>
<td>file I/O</td>
<td>FIOxxx</td>
<td>ioctl.h</td>
<td>7</td>
</tr>
<tr>
<td>socket I/O</td>
<td>SIOxxx</td>
<td>ioctl.h</td>
<td>25</td>
</tr>
<tr>
<td>terminal I/O</td>
<td>TIOxxx</td>
<td>ioctl.h</td>
<td>35</td>
</tr>
</tbody>
</table>

Figure 1: ioctl operations
Unbuffered I/O: File Properties

- int stat(const char *file_name, struct stat *buf);
- int fstat(int filedes, struct stat *buf);
- int lstat(const char *file_name, struct stat *buf);

- defined in the stat.h header file
- returns file properties in buf defined as a struct stat
- returns a -1 if an error occurred
- fstat is similar to stat except that it operates on file descriptors
- lstat is similar to stat except that it returns the properties of the symbolic link instead of the file pointed to by the symbolic link
the `struct stat` is defined as:

```c
struct stat {
  dev_t st_dev; /* device */
  ino_t st_ino; /* inode */
  mode_t st_mode; /* protection */
  nlink_t st_nlink; /* number of hard links */
  uid_t st_uid; /* user ID of owner */
  gid_t st_gid; /* group ID of owner */
  dev_t st_rdev; /* device type (if inode device) */
  off_t st_size; /* total size, in bytes */
  unsigned long st_blksize; /* blocksize for filesystem I/O */
  unsigned long st_blocks; /* number of blocks allocated */
  time_t st_atime; /* time of last access */
  time_t st_mtime; /* time of last modification */
  time_t st_ctime; /* time of last change */
};
```

macros provided for checking filetype:

- `S_ISREG(m)` is it a regular file?
- `S_ISDIR(m)` directory?
- `S_ISCHR(m)` character device?
- `S_ISBLK(m)` block device?
- `S_ISFIFO(m)` fifo?
- S_ISLNK (m) symbolic link?
- S_ISSOCK (m) socket?

<table>
<thead>
<tr>
<th>Mask</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IFMT</td>
<td>0170000</td>
<td>bitmask for the file type bitfields</td>
</tr>
<tr>
<td>S_IFSOCK</td>
<td>0140000</td>
<td>socket</td>
</tr>
<tr>
<td>S_IFLNK</td>
<td>0120000</td>
<td>symbolic link</td>
</tr>
<tr>
<td>S_IFREG</td>
<td>0100000</td>
<td>regular file</td>
</tr>
<tr>
<td>S_IFBLK</td>
<td>0060000</td>
<td>block device</td>
</tr>
<tr>
<td>S_IFDIR</td>
<td>0040000</td>
<td>directory</td>
</tr>
<tr>
<td>S_IFCHR</td>
<td>0020000</td>
<td>character device</td>
</tr>
<tr>
<td>S_IFIFO</td>
<td>0010000</td>
<td>fifo</td>
</tr>
</tbody>
</table>

Figure 2: Flags defined for the st_mode field
<table>
<thead>
<tr>
<th>Mask</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_ISUID</td>
<td>0004000</td>
<td>set UID bit</td>
</tr>
<tr>
<td>S_ISGID</td>
<td>0002000</td>
<td>set GID bit</td>
</tr>
<tr>
<td>S_ISVTX</td>
<td>0001000</td>
<td>sticky bit</td>
</tr>
<tr>
<td>S_IRWXU</td>
<td>00700</td>
<td>mask for file owner permissions</td>
</tr>
<tr>
<td>S_IRUSR</td>
<td>00400</td>
<td>owner has read permission</td>
</tr>
<tr>
<td>S_IWUSR</td>
<td>00200</td>
<td>owner has write permission</td>
</tr>
<tr>
<td>S_IXUSR</td>
<td>00100</td>
<td>owner has execute permission</td>
</tr>
<tr>
<td>S_IRWXG</td>
<td>00070</td>
<td>mask for group permissions</td>
</tr>
<tr>
<td>S_IRGRP</td>
<td>00040</td>
<td>group has read permission</td>
</tr>
<tr>
<td>S_IWGRP</td>
<td>00020</td>
<td>group has write permission</td>
</tr>
</tbody>
</table>

Figure 3: Flags defined for the `st_mode` field
<table>
<thead>
<tr>
<th>Mask</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IIXGRP</td>
<td>00010</td>
<td>group has execute permission</td>
</tr>
<tr>
<td>S_IRWXO</td>
<td>00007</td>
<td>mask for permissions for others</td>
</tr>
<tr>
<td>S_IROTH</td>
<td>00004</td>
<td>others have read permission</td>
</tr>
<tr>
<td>S_IWOTH</td>
<td>00002</td>
<td>others have write permission</td>
</tr>
<tr>
<td>S_IIXOTH</td>
<td>00001</td>
<td>others have execute permission</td>
</tr>
</tbody>
</table>

Figure 4: Flags defined for the `st_mode` field
Unbuffered I/O: File Properties

* int access(const char *pathname, int mode);
  - function provided to check users permissions and test for the existence of a file
  - returns a -1 if an error occurred
  - mode can either be either or more of the following:
    * F_OK - checks for the existence of a file
    * R_OK - read access permitted
    * W_OK - write access permitted
    * X_OK - execute access permitted

* mode_t umask(mode_t mask);
  - set the file creation mask
  - the values of the mask can be found in Figures 2, 3 and 4
– returns -1 if an error occurs and returns the current mode if successful

* mode_t chmod(const char *path, mode_t mask);
* int fchmod(int fildes, mode_t mode);
  – change permissions of a file
  – the values of the mask can be found in Figures 2, 3 and 4
  – returns -1 if an error occurred

* int chown(const char *path, uid_t owner, gid_t group);
* int fchown(int fd, uid_t owner, gid_t group);
* int lchown(const char *path, uid_t owner, gid_t group);
  – change ownership of a file
– returns -1 if an error occurred

* int truncate(const char *path, off_t length);
* int ftruncate(int fd, off_t length);
  – truncates the size of the file to length bytes
  – returns -1 if an error occurred

* int link(const char *oldpath, const char *newpath);
* int symlink(const char *oldpath, const char *newpath);
  – creates a hard and soft link respectively
  – returns -1 if an error occurred
int unlink(const char *pathname);

- deletes a file from the filesystem
- if a symlink is encountered that link is delete and not the file
- returns -1 if an error occurred

int utime(const char *filename, struct utimbuf *buf);

- changes the access an modification times of a file
- the times are determined by the value buf
- returns -1 if an error occurred
Unbuffered I/O: Temporary Files

⋆ char *mktemp(char *template);
– generates a unique filename based on the template
– template must be a character array that contains a file name whose last letter must be XXXXXX
– the XXXXXX will be replaced with symbols that shall guarantee its uniqueness
– new string shall serve as the filename of the temporary file
– the value of template shall be modified with the filename of the actual temporary file
– NULL shall be returned on error and template shall be an empty string

⋆ int mkstemp(char *template);
– similar to mktemp
– the file descriptor of the opened file shall be returned
– returns -1 if an error occurred
Section IV

Unix Directories
Directory MANIPULATION

⋆ int mkdir(const char *pathname, mode_t mode);
  - attempts to create a directory named pathname
  - sets the permission to mode similar to those in chmod
  - returns a -1 if an error occurred

⋆ int rmdir(const char *pathname);
  - deletes a directory named pathname
  - returns a -1 if an error occurred

⋆ int chdir(const char *path);
⋆ int fchdir(int fd);
  - changes the path of the current working directory
- returns a -1 if an error occurred

* char *getcwd(char *buf, size_t size);
* char *get_current_dir_name(void);
* char *getwd(char *buf);
  - returns the path of the current working directory to the variable buf
  - returns NULL if an error occurred

* DIR *opendir(const char *name);
* struct dirent *readdir(DIR *dir);
* void rewinddir(DIR *dir);
* int closedir(DIR *dir);
  - defined in the header file dirent.h
- entries are returned as a pointer to struct dirent
- readdir returns NULL if there is an error
- returns a -1 if an error occurred