Aalto University
School of Electrical
Engineering

ELEC-C7310 Sovellusohjelmointi
Lecture 3: Filesystem

Risto Järvinen

September 21, 2015

Lecture contents

- Filesystem concept.
- System call API.
- Buffered I/O API.
- Filesystem conventions.
- Additional stuff.
- Terminals.

- Stevens: ch3-5 and ch18(=terminals).
- Kerrisk: parts of ch4-5, ch13-14 and ch62(=terminals).

Filesystem concept

- Abstraction for storage devices.
- Mounted on any block device, a storage device that stores an array of data blocks.
- Also network filesystems (NFS, CIFS/SMB, various).
- Filesystems are connected in a hierarchy.
- "/" root, start of the hierarchy.
- "/abc/def/ghi" pathname.
- Files, directories, devices, are all stored as names in the hierarcy.

Types of files

- Regular files.
- Pipes, named.
- Directories.
- Device files.
- Symbolic links.
- Sockets.

File API

- Basic operations:
  - Open, initiates access to a file.
  - Read, copies data from file to process memory.
  - Write, copies data from process memory to file.
  - Seek, changes location where file is read/written.
  - Close, ends access to file.
- Files are opened via pathname, then accessed via file descriptor.
- File descriptor is an index in the descriptor table maintained by the OS for the process.
- Stdin/stdout/stderr are defined to be the first three file descriptors, with indexes 0, 1 and 2.

Opening files

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

- Primary flags: O_RDONLY, O_WRONLY, O_RDWR.
- Optional flags: O_APPEND, O_CREAT, O_EXCL, O_TRUNC, O_NOCTTY, O_NONBLOCK.
- Synchronous I/O: O_DSYNC, O_RSYNC, O_SYNC.
- creat() is like open() with O_WRONLY|O_CREAT|O_TRUNC.

Reading and writing

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

- When reading, read returns the number of bytes read. Amount may be less than the amount requested! Value -1 means error, and zero means end of file.
- Similarly, writing returns the number of bytes written, may be less than amount requested, even zero. Value -1 means error.
- Partial operations may happen for many reasons:
  - Trying to read past the end of the file.
  - O_NONBLOCK on some file types returns only the available data.
  - Signal may interrupt a read.

Seeking

```
off_t lseek(int fd, off_t offset, int whence);

#define _FILE_OFFSET_BITS 64
 or
#define _LARGEFILE64_SOURCE
off64_t lseek64(int fd, off64_t offset, int whence);
```

- Every open file descriptor has an offset that stores where next operation will occur.
- whence: SEEK_SET, SEEK_CUR, SEEK_END
- Some file descriptors are non-seekable. F.ex. FIFOs.
- 32-bit to 64-bit transition issues.

Closing files

```
int close(int fd);
```

- Closes the file descriptor.
- If file was unlink()ed, when last file descriptor referring to it is closed, then file is actually deleted.
- Return value also returns errors that happened after the last write.

File permissions

- File mode, 16-bit value, usually represented as six digit octal value.
- Three lowest digits read/write/execute permissions for owner/group/others.
- Fourth digit : setuid, setgid, sticky.
- Two highest digits: file type, FIFO / character device / directory / block device / regular file / symbolic link / socket.
- See man 2 stat.
- Process has umask value to set defaults to created files.

```
mode_t umask(mode_t mask);
```

Permissions API

```
int access(const char *pathname, int mode);
int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
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);
```

File info

```
#include <sys/stat.h>

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- lstat() is like stat() but if file is a symbolic link, it stats the link and not the linked file.

File info structure

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    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 ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B 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 status change */
};
```

Creating and deleting links

```
#include <unistd.h>

int unlink(const char *pathname);
int link(const char *oldpath, const char *newpath);
int symlink(const char *oldpath, const char *newpath);
ssize_t readlink(const char *path,
                 char *buf, size_t bufsiz);
```

- unlink() is used to delete files.

Directory operations 1/2

```
#include <unistd.h>

int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
int chdir(const char *path);
int fchdir(int fd);

char *getcwd(char *buf, size_t size);
```

Directory operations 2/2

```
#include <dirent.h>

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result));
void seekdir(DIR *dirp, long offset);
void rewinddir(DIR *dirp);
long telldir(DIR *dirp);
int closedir(DIR *dirp);

DIR *fdopendir(int fd);
```

Additional

```
int rename(const char *oldpath, const char *newpath);

int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);

int mknod(const char *path, mode_t mode, dev_t dev);

int utime(const char *path, const struct utimbuf *times);
int utimes(const char *path, const struct timeval times[2]);
```

- rename() is atomic on POSIX-compliant systems.

Duplicating file descriptors

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Creating unnamed pipes

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

- Both ends are in one process, unless passed via some mechanism (fork() or UNIX domain socket).

Buffered file I/O

- C library provides buffered I/O on top of system calls.
- Should be familiar from C programming, let's just run through.
- Three modes:
  - Fully buffered (_IOFBF)
  - Line buffered (_IOLBF)
  - Unbuffered (_IONBF)

Setting buffering

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode,
            size_t size);
```

- buf is user allocated buffer, if none given, system allocates it.
- setbuf() must have BUFSIZ size buffer. Just use setvbuf().

Opening and closing files

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
FILE *freopen(const char *path, const char *mode,
              FILE *stream);
int fclose(FILE *fp);
int fflush(FILE *stream);
```

Buffered input

```
#include <stdio.h>

int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
int getc(FILE *stream);
int getchar(void);
char *gets(char *s);  <-- DEPRECATED
int ungetc(int c, FILE *stream);
size_t fread(void *ptr, size_t size, size_t nmemb,
             FILE *stream);
```

Buffered output

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
int puts(const char *s);
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

Error handling

```
#include <stdio.h>

void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
int fileno(FILE *stream);
```

Seeking

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
void rewind(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, fpos_t *pos);
```

Formatted output

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);

#include <stdarg.h>

int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

Formatted input

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);

#include <stdarg.h>

int vscanf(const char *format, va_list ap);
int vsscanf(const char *str, const char *format, va_list ap);
int vfscanf(FILE *stream, const char *format, va_list ap);
```

Temporary files

```
#include <stdio.h>

FILE *tmpfile(void);
```

- Obsolete functions tmpnam() and tempnam(), mktemp()

```
int mkstemp(char *template);
int mkostemp (char *template, int flags);
int mkstemps(char *template, int suffixlen);
int mkostemps(char *template, int suffixlen, int flags);
```

Synchronizing data to storage

```
#include <unistd.h>

int fsync(int fd);
int fdatasync(int fd);
```

- fsync() flushes all changes related to the file to disk. (data+metadata)
- fdatasync() flushes all data changes related to the file to disk. (data only)

Filesystem tricks: How to keep a file consistent

- Avoid losing files, either keep new or old.
- Procedure:
  1. Write new data into a temporary file
  2. Fsync() the temporary file
  3. Rename() it to the target name
- Use on important files, the ones that must not be corrupt.

Tale of fsync() and ext3

- Example of how synchronizing data to disk may have really adverse effect on system performance.
- Hard disks are slow. 6-9ms seek times, transfer speed in tens of MB/s. (SSDs help some)
- Ext3 is a common *journaling* Linux filesystem, meaning it keeps a separate record of disk actions to maintain consistency even in case of sudden power losses.
- Does this by keeping a log in the order in which events occured and plays them out in that order.
- When you fsync() any file, all operations that preceded it must be also flushed.
- Firefox 3 introduced storage backend that fsync()ed data to disk..

Filesystem tricks: Sparse files

- UNIX filesystems can allocate discontinuous blocks for a file.
- Only parts of a file actually written to are stored.
- Every unwritten part reads as zeroes.
- Procedure:
  1. Create file.
  2. Seek to points where you want data.
  3. Write data.
  4. Repeat 2-3.
- Use cases include databases and filesystem images.

For those occasions.. ioctls

- How to cover unanticipated operations in filesystem?

```
#include <sys/ioctl.h>

int ioctl(int fd, int request, ...);
```

- Basically a open interface to OS kernel, tied to filesystem.
- Calls are varied and there are few standards.
- Example: Linux CDROM ioctl calls.
  http://lxr.free-electrons.com/source/Documentation/ioctl/cdrom.txt

Memory mapping 1/2

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
int msync(void *addr, size_t length, int flags);
```

- Uses virtual memory system to map a file into memory.
- On modern systems, demand-paged.

Memory mapping 2/2

- Flags:
  - MAP_SHARED, mapping is allowed to be shared.
  - MAP_PRIVATE, copy-on-write mapping, based on file, but creates a private copy if data is modified. No changes to files.
  - MAP_ANONYMOUS, mapping is not backed by a file, rather initialized to zero.
- Very useful, can simplify many problems, used for shared memory IPC (more later).
- Though, error handling is dangerous, and makes performance analysis complex.

Terminals

- Abstraction for a text-based display device
- Serial port, monitor+keyboard console, terminal programs, ssh terminal session, etc.
- Loads and loads of history, nowadays mostly worked out.

- Two-ended; software<->device or software<->software (pty)
- Two modes : raw and cooked, alias non-canonical and canonical.
- OS keeps terminal line discipline for handling cooked mode
- POSIX API is called termios

- A nice write-up on terminals: http://www.linusakesson.net/programming/tty/index.php

Terminal attributes

```
#include <termios.h>
#include <unistd.h>

int tcgetattr(int fd, struct termios *termios_p);
int tcsetattr(int fd, int optional_actions,
              const struct termios *termios_p);
```

- Struct termios has members c_iflag, c_oflag, c_cflag, c_lflag and c_cc.
- Dozens of attributes controlled by the members. See man tcgetattr.

Buffer control

```
int tcsendbreak(int fd, int duration);
int tcdrain(int fd);
int tcflush(int fd, int queue_selector);
int tcflow(int fd, int action);
```

Terminal line speed

```
speed_t cfgetispeed(const struct termios *termios_p);
speed_t cfgetospeed(const struct termios *termios_p);

int cfsetispeed(struct termios *termios_p,
                speed_t speed);
int cfsetospeed(struct termios *termios_p,
                speed_t speed);
```

- Note: speed_t is not a bps value, but a type tag. See man page.

Non-POSIX but useful

```
void cfmakeraw(struct termios *termios_p);
int cfsetspeed(struct termios *termios_p,
               speed_t speed);
```

- cfmakeraw() does most things you want to do to use a raw terminal.
- cfsetspeed() sets both input and output speed at once.

Why use such monstrosity?

- Legacy.
- Asking password on console.
- Serial port programming.
- Helper libraries for actual console use: ncurses, S-Lang.

Did you learn this?

- Filesystem is a straightforward tool.
- Filesystem semantics are important.
- Terminals are painful but necessary.

- Next time: Processes and signals.