Chapter 4

Files

File Systems

Read … :

BLP: Chapter 3
- **Experiment!**

USP: Chapters 4 and 5
- **Experiment!**

Background on File Systems in Operating Systems
- Silberschatz: **11**, 12
- Tanenbaum: **6**

File Concept

A file is a named collection of related information that is recorded on secondary storage.

- smallest unit of secondary storage
- usually viewed as
  - sequence of bits
  - sequence of bytes
  - sequence of characters
  - sequence of lines
  - sequence of records
- Meaning of stored information is **defined by creator**.

File Attributes

- **Name**
- **Identifier**
- **Type**
- **Location**
- **Size**
- **Protection** — in particular, ownership
- **Access time, modification time**

Disk Organization

- One **boot control block** per disk — information for booting an OS from that disk.
- Several **partitions**:
  - **partition control block**: partition size, block size, block management data structures (free block count, free bolck pointer, free FCB count, free FCB pointer)
  - **directory structure**
  - **File Control Blocks (FCBs) — in UNIX: inode:**
    - Ownership, premission information
    - Location of data blocks

inodes in Detail

- mode (permissions)
- number of hard links
- owner, group
- timestamps: modification mtime, access atime, change ctime
- size (in bytes)
- number of blocks allocated
- Pointers to allocated blocks:
  - 12 direct block pointers to the first data blocks of the file
  - one **single indirect** pointer, points to a block containing pointers to the next (blocksize / pointersize) data blocks
  - one **double indirect** pointer
  - one **triple indirect** pointer

UNIX Permissions

"***Condensed access control lists***" with three user classifications:

- **Owner** (u — user): usually creator, identified by user ID
- **Group** (g): identified by a group ID
- **Universe** (o — other): all other users in the system

Three kinds of access:

	Files	**Directories**
r	read	list
w	write	create and change entries
x	execute	traverse

Special permissions: setUID, setGID (s), and "sticky bit" (t).

Directories

- A directory is a file with **contents interpreted by the operating system**
- It contains a list of **entries:**
  - **file name**
  - inode number

Links

“Hard” Links:

- Several directory entries point to the same inode
- By definition possible only **within a single file system**
- “Remove” `rm` is really `unlink`: the directory entry is deleted, and the inode’s link count decreased.
- Interactive creation: `ln` *existing_file new_name*
- Only the super-user can hard link directories

Symbolic Links

- Special type of file that contains only a file path (relative or absolute)
- Target need not exist
- Can cross file system boundaries
- Interactive creation: `ln` **`-s`** *existing_file new_name*

`ls -lai`

```
759 drwxr-x--x 2 kahl users 4096 Mar 23 19:47 .
729 drwxrwxrwt 6 root root  4096 Mar 23 19:45 ..
145 -rw------- 1 kahl users    0 Mar 23 19:52 .hiddden
143 -rw-r--r-- 1 kahl users   17 Mar 23 19:47 README
144 lrwxrwxrwx 1 kahl users    6 Mar 23 19:50 info -> README
140 -rw-r-Sr-- 1 kahl users   35 Mar 23 19:44 lockable
141 -rwx--x--x 2 kahl users   13 Mar 23 19:46 program
142 -rwsr-x--- 1 kahl good   105 Mar 23 19:46 restricted
141 -rwx--x--x 2 kahl users   13 Mar 23 19:46 test
```


Copying Directories

- `cp -R` on Linux nowadays reproduces symbolic links
- `cp -R` does not recreate hard links!
- Portable solution:
  `tar cf - -C` ***fromDir*** `. | tar xf - -C` ***toDir***
- `tar` — originally for creating **t**ape **ar**chives:
  - one of: `c`reate, e`x`tract, `t`able of contents
  - “`f` ***file***” from or to ***file*** (“`-`” for *stdin*/*ctdout*)
  - “`-C` ***directory***”: change directory (has to exist!)
  - creation needs file arguments
- Producing packages:
  `tar czf` ***package***`.tar.gz` ***directory***
  - “`z`” for *gzip*; “`j`” for *bzip2*

File System Mounting

- Individual hard disks (or partitions) contain individual file systems
- Many such file systems are integrated into a single “file structure”
- In MS-DOS and MS-Windows: *drive letters* as identifiers
- In UNIX: **mounting** of file systems onto existing (empty) directories.
  - /etc/fstab contains default mounting instructions for the system
  - mount -a mounts everything listed (without noauto) in /etc/fstab

UNIX File System Mounting Example

- /dev/hda5 contains /dir/file
- Running system contains (empty) directory /local/p5
- mount /dev/hda5 /local/p5
- Now available: /local/p5/dir/file
- umount /local/p5
- The original contents of directory /local/p5 is visible again
- umount fails if the mounted file system is in use, e.g., open files, process working directory

  Typical case: cd /media/cdrom in one window,
  umount /media/cdrom in another window

Mounting Hints

- If /etc/fstab contains a line for mounting /dev/hda5 to /local/p5, then the command mount /local/p5 is sufficient — typical application: mount /media/cdrom
- mount can be called directly, without a line in /etc/fstab
- Useful mounting options: ro, nosuid, noatime
- mount may need to be told the file system type: ufs, ext2, ext3, jfs, xfs, reiserfs, proc, nfs, fat32, ntfs, smbfs, iso9660
- Since Linux 2.4.0 it is possible to remount part of the file hierarchy somewhere else. The call is

  mount --bind olddir newdir

File Operations

- **Creation**
- **Writing**
- **Reading**
- **Repositioning**
- **Deleting**
- **Truncating**

I/O System Calls

- *open*() creates a file descriptor associated with a file
- *close*() disassociates a file descriptor from its file
- *read*() performs input via a file descriptor
- *write*() preforms output via a file descriptor
- *fcntl*() can e.g. change flags of a file descriptor
- *fsync*() commits output to disk
- *lseek*() repositions the offset of a file descriptor

read(2)

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

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

If *count* is zero, *read*() returns zero and has no other results. If *count* is greater than *SSIZE_MAX*, the result is unspecified.

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because *read*() was interrupted by a signal. On error, -1 is returned, and *errno* is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

Reading & Writing — Counterexample 1

(USP Example 4.5)

```
#define BLKSIZE 1024
char buf[BLKSIZE];

read(STDIN_FILENO, buf, BLKSIZE);
write(STDOUT_FILENO, buf, BLKSIZE);
```


USP Program 4.1: Reading Lines

```
int readline(int fd, char *buf, int nbytes) {
  int numread = 0, returnval;

  while (numread < nbytes – 1) {
    returnval = read(fd, buf + numread, 1);
    if ((returnval == –1) && (errno == EINTR)) continue;
    if ( (returnval == 0) && (numread == 0) )    return 0;
    if (returnval == 0)                          break;
    if (returnval == –1)                         return –1;
    numread++;
    if (buf[numread–1] == '\n')
     { buf[numread] = '\0';    return numread; }
  }
  errno = EINVAL; return –1;
}
```


write(2)

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

write() writes up to *count* bytes to the file referenced by the file descriptor `fd` from the buffer starting at *buf*. POSIX requires that a *read*() which can be proved to occur after a *write*() has returned returns the new data. Note that not all file systems are POSIX conforming.

— *This is the* "**UNIX file system semantics**"!

On success, the number of bytes written are returned (zero indicates nothing was written). On error, -1 is returned, and *errno* is set appropriately. If *count* is zero and the file descriptor refers to a regular file, 0 will be returned without causing any other effect. For a special file, the results are not portable.

Reading & Writing — Counterexample 2

(USP Exmp. 4.6)

```
#define BLKSIZE 1024
char buf[BLKSIZE];
ssize_t bytesread;

bytesread = read(STDIN_FILENO, buf, BLKSIZE);
if (bytesread > 0)
   write(STDOUT_FILENO, buf, bytesread);
```


Copying — USP Program 4.2

```
int copyfile(int fromfd, int tofd) {
  char *bp, buf[BLKSIZE]; int bytesread, byteswritten, totalbytes=0;
  for ( ; ; ) {
    while (((bytesread = read(fromfd, buf, BLKSIZE)) == -1) &&
          (errno == EINTR)) ;     /* handle interruption by signal */
    if (bytesread ≤ 0) break;      /* real error or EOF on fromfd */
    bp = buf;
    while (bytesread > 0) {
      while(((byteswritten = write(tofd, bp, bytesread)) == -1) &&
            (errno == EINTR)) ;    /* handle interruption by signal */
      if (byteswritten ≤ 0)   break;              /* real error on tofd */
      totalbytes += byteswritten; bytesread -= byteswritten;
      bp += byteswritten;
    }
    if (byteswritten == -1)  break;               /* real error on tofd */
  }
  return totalbytes; }    /* end of copyfile() */
```


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);
```

Flags are constructed using bitwise-*or* from e.g.:

- ***exactly one*** of *O_RDONLY*, *O_WRONLY*, or *O_RDWR*
- *O_CREAT* — If the file does not exist it will be created
- *O_TRUNC* — Truncate existing file if opened for writing
- *O_APPEND* — Each *write* will append at end
- *O_SYNC* — Each *write* will block until data is on disk
- *O_ASYNC* — Generate signal when I/O becomes possible

File Access Permissions

The **mode** is constructed using bitwise-*or* from:

- *S_IRUSR* — **user** has **read** permission
- *S_IWUSR* — **user** has **write** permission
- *S_IXUSR* — **user** has **execute** permission
- *S_IRGRP* — **group** has **read** permission
- *S_IWGRP* — **group** has **write** permission
- *S_IXGRP* — **group** has **execute** permission
- *S_IROTH* — **others** have **read** permission
- *S_IWOTH* — **others** have **write** permission
- *S_IXOTH* — **others** have **execute** permission

Abbreviations:

- *S_IRWXU* — user has read, write and execute permission
- *S_IRWXG* — group has read, write and execute perm.
- *S_IRWXO* — others have read, write and execute perm.

File Handles

A file **handle** is a logical name for referring to a particular file or device for I/O:

– ***file descriptor:*** int (for system calls)

– ***file pointer (stream):*** *FILE* * (library type)

File descriptors are indices into the file descriptor table of process.

File pointers point to a library data structure *FILE* containing

– A file descriptor

– A buffer (array of bytes)

Buffered I/O

Writing:

- data is written into the buffer
- when buffer full, or upon request *fflush*(), the buffer is *flushed* to its destination
- *stdout* is buffered, *stderr* is unbuffered
- If data is not in the file / on the screen after a program crash, this does **not** mean that the program never wrote the data!

Reading:

- When data is requested while buffer is empty, data is read from destination (file or e.g. *stdin*)
- While buffer is non-empty, read requests are satisfied from the buffer.
- (this makes type-ahead possible)

BLP page 116 — *copy_stdio.c*

```
#include <stdio.h>

int main()
{
   int c;
   FILE *in, *out;

   in = fopen("file.in","r");
   out = fopen("file.out","w");

   while((c = fgetc(in)) ≠ EOF)
      fputc(c,out);

   exit(0);
}
```


Directory Interaction

Library functions:

- *DIR* **opendir*(const char **name*);
- int *closedir*(*DIR* **dir*);
- struct *dirent* **readdir*(*DIR* **dir*); — try `man readdir!`
- *off_t telldir*(*DIR* **dir*);
- void *seekdir*(*DIR* **dir*, *off_t offset*);
- void *rewinddir*(*DIR* **dir*);
- int *scandir*(const char **dir*, struct *dirent* ***namelist*,
  int(**filter*)(const struct *dirent* *),
  int(**compar*)(const struct *dirent* **, const struct *dirent* **));

BLP page 122 — *printdir.c*

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>

void printdir(char *dir, int depth)
{
   DIR *dp;
   struct dirent *entry;
   struct stat statbuf;

   if((dp = opendir(dir)) == NULL)
     { fprintf(stderr,"cannot open directory: %s\n", dir); return; }
   chdir(dir);
```


BLP page 122 — *printdir.c* (ctd.)

```
while((entry = readdir(dp)) ≠ NULL) {
      lstat(entry→d_name,&statbuf);
      if(S_ISDIR(statbuf.st_mode)) {
         // Found a directory, but ignore . and ..
         if(strcmp(".",entry→d_name) == 0 ||
strcmp("..",entry→d_name) == 0)
            continue;
         printf("%*s%s/\n",depth,"",entry→d_name);
         printdir(entry→d_name,depth+4);   // Recurse at a new
indent level
      }
      else printf("%*s%s\n",depth,"",entry→d_name);
   }
   chdir(".."); closedir(dp);
}
```


System-Wide Open-File Table

Keeping track of **all** files currently open for **any** process.
Entry contents: *process-independent* information:

– Location of file of disk
– Location of cached file information
– Open count
– ***current position*** (in UNIX)

For each process, the OS keeps a **File Descriptor Table**.
Entry contents:

– Index (pointer) into system-wide open-file table

Process-specific information:

– (*current position — in some systems*)
– access rights, "close-on-exec flag"

File Tables in UNIX

In-memory Inode Table

- For each open file **a single copy** of its file control blocks
- Acts as **cache** for file control blocks

System File Table

- one entry per active *open*
- contains reference to in-memory inode
- current position, access rights, access mode

File Descriptor Table

- one per process — copied by *fork*
- contains reference to system file table entry
- In user space, references to file descriptors are int indexes into this table.
- file locks — not copied by *fork*

Redirection — *dup2* — Duplicate a File Descriptor

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

dup and *dup2* create a copy of the file descriptor *oldfd*.

After successful return of *dup* or *dup2*, the old and new descriptors may be used interchangeably.

This is used to implement **I/O redirection** in shells:

- Input redirection: "*stdin* comes from `somefile`":
  `somecommand < somefile`
- Output redirection: "*stdout* goes to `somefile`":
  `somecommand > somefile`
- Direct FD redirection: "*stderr* goes to `somefile, too`":
  `somecommand > somefile 2&>1`

Redirection — USP Program 4.18

```
#include <fcntl.h> <stdio.h> <sys/stat.h> <unistd.h> "restart.h"
#define FLAGS (O_WRONLY | O_CREAT | O_APPEND)
#define MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

int main(void) {
  int fd = open("my.file", FLAGS, MODE);
  if (fd == -1) { perror("Failed to open my.file"); return 1; }
  if (dup2(fd, STDOUT_FILENO) == -1) {
    perror("Failed to redirect standard output"); return 1; }
  if (r_close(fd) == -1) {
    perror("Failed to close the file"); return 1; }
  if (write(STDOUT_FILENO, "OK", 2) == -1) {
    perror("Failed in writing to file"); return 1; }
  return 0;
}
```


Temporary Files

- char * *tmpnam*(char * *s*)
  produces a file name that ***might*** be used for temporary files but creates a **security risk!**

  Use *mkstemp* **or** *tmpfile* **instead!**

- int *mkstemp*(char *template)
  creates and *open*s a new temporary file.

- *FILE *tmpfile*(void)
  creates and *fopen*s a new temporary file, which will be deleted when closed.

- char *mkdtemp*(char *template)
  creates a new temporary directory.

Read

BLP Chapter 4: The Linux Environment

- int *getopt*(int *argc*, char * const *argv*[], const char **opts*);
- Environment Variables — *getenv*, *putenv*
- Time — UNIX internally uses UTC (GMT)
- User information
- Host information — `uname -a`
- Logging — void *syslog*(int *priority*, const char **format*, ...);
- Resource limits

BLP Chapter 5: Terminals

- Canonical mode: terminal passes only complete lines to application.
- int *isatty*(int *fd*) checks whether *fd* is connected to a terminal
- struct *termios*
- `stty -a`
- `terminfo` capabilities
- **Virtual Consoles:** Crtl-Alt-F*k* switches to console number *k*

  Console 7: First X server (if running), DISPLAY `:0`

BLP Chapter 6: *curses*

- Arbitrary **cursor** movement and screen updates.
- *#include <curses.h>*
- Link with `-lcurses`
- Coordinates (0,0) are in left upper corner — as often
- "Windows" — devide terminal screen into rectangular areas
- New CD Collection Application

Read BLP Section 7.1: Memory Management

- Linux implements a demand paged virtual memory system
- (We will get back to memory management later)

BLP

- BLP Section 7.2: File Locking: Postponed
- BLP Section 7.3: dbm indexed file storage system
- BLP Chapter 8: mysql
  - — look at this for your database course
  - — consider also postgresql