

Files and Directories

- *Administrative*
 - * HW# 1 Due this week
- *Goals: Understand the file system concepts*
 - * files, links, and directories
 - * device independent interface
- *Topics:*
 - * 3.0 Device independence
 - * 3.1 Directory operations, Paths
 - * 3.2 Disk structures: inodes, links, directories
 - * 3.3 Memory structures: descriptors, file pointers
 - * 3.4-5,3.9 Filters, Redirection, Pipes
 - * 3.6-8 File operations: blocking/non-blocking
- *Readings: Chapter 3 (Robbins, pp.76-137)*
- *Recommended Exercises: 3.1 - 12*

3.0 Device independence

- *Q? Which devices are of interest?*
 - * terminal, disk, tapes, audio, network, ...
 - * special files located in /dev/
 - * Q? Name 3 other device controlled by OS.

- *Why device independence ?*
 - * Ex. restoring files from tape backup to disk
 - * text/images from internet -> disk -> printer
 - * audio: microphone -> disk/CD -> speakers
 - * How many interfaces do you want to learn?

- *What is Device independence ?*
 - * uniform interface to all devices!
 - * Operations: open, close, read, write, ioctl
 - * File descriptors are used for all devices
 - * Device driver hides device specific things

3.0 Device independence

- *Advantage: simplifies systems programming*
 - * Ex. I/O redirection from terminal/keybard to files
 - * Ex. Pipes to link filter processes
 - * postscript files
 - * tar files (interchanging tapes, disk)
 - * audio files: (Sec. 3.11, Program 3.4)
- *Q? What are the disadvantages of device independence?*
 - * Which applications need device-specific operations?
- *Types of files:*
 - * Regular data files, directory files,
 - * Block special files - e.g. disk
 - * Character special files - e.g. keyboard
 - * Others, e.g. socket, ...
- *How are collections of files organized?*

3.1 Directory operations, Paths

- *Why directories?*
 - * Allows symbolic naming of files
 - * EE/CS Bldg. instead of
 - 200 Union St. SE, Minneapolis
- *Directories: filenames --> physical properties*
 - * Disk addresses - start, end, ...
 - * Type, size, date of creation/update
 - * owner, permission, ...
- *Directory Structures*
 - * Linear tables
 - * Fixed depth tree, e.g. one linear table per user
 - * General Tree structures (Fig. 3.1, pp. 79)

3.1 Directory Operations

- *Operation on Tree Structured directory*
 - * A. Where am I?
 - * B. Take me home (or to another node)
 - * C. Where is an interesting file?
 - * D. Default search paths for popular executables
 - * E. open, read, write, close
- *A. Current working directory*
 - * Command: pwd
/dirA/dirB
 - * System calls - Examples 3.2, 3.3 (pp. 80-81)
extern char *getcwd(char *buf, size_t size);
long pathconf(const char *path, int name);
- *Naming files- fullname or nicknames*
 - * Absolute: /dirA/my1.dat, /dirA/dirB/my1.dat
- path(root, file)
 - * Relative: my1.dat, ../my2.dat
- path(current working directory, file)
 - Special directories: . and ..

3.1 Directory operations

- *B. Take me home (or to another node)*
 - * command: cd [<directoryname>]
cd /dirA ; pwd
cd ../dirC ; pwd
cd ; pwd
 - * Q? Identify system call from Table 5.3 (pp. 191).
- *C. Where is an interesting file? [Appendix A.1.3]*
 - * Command: find pathname(s) operands
find / -name "cc" -print
find . -name "*.c" -size +10 -print
- *D. Default search paths for popular executables*
 - * 3.1.2 Search Paths = collection of directories
 - * Shell looks in these for commands typed in!
printenv | grep PATH
PATH=/usr/bin:/etc:/usr/local/bin:.
 - * Interesting Exercise 3.1 (pp. 85)
 - * Q? Recall system call to extract PATH (Sec. 2.9).

3.2 Disk structures: inodes, links, directories

- *E. open, read, write, close*
 - * System calls: `opendir()`, `readdir()`, `closedir()`
 - * Ex. specs (pp. 82), Program 3.1 (pp. 83)
 - * Note: struct `dirent`
 - * Q? Is `opendir()` signal safe?
- *3.1.3 Unix File Systems (Fig. 3.2, pp. 86)*
 - * disk drive --> partition(s), p1, p2, ...
 - * each partition has a directory
 - * `directory(p1)` mounted on `directory(p2)`
- *Q? What is kept under the following?*
 - * `/dev`, `/etc`, `/home`, `/opt`, `/usr`, `/var`

3.2 Disk structures: inodes, links, directories

- *inode = structure to store a file descriptor*
 - * Figure 3.3 (pp. 87)
 - * Fixed size (Does not contain filenames)
 - * Stored in inode-list array at disk start

- *Pointers to data block*
 - * Direct, or indirect (1-3 levels)
 - * Trade-of between small and large files
 - * Interesting exercise 3.3 (pp. 87)

- *Getting file information from inode:*
 - * Has file size, location, owner, c/a/m time, permission,
 - pointers to data blocks, hard link count
 - * System call: `stat()`, spec. pp. 88
 - * Program Example 3.6 (pp. 89)

3.2 Disk structures: inodes, links, directories

- *3.2.1 Directory Representation*
 - * UNIX directory is a file
 - * Directory entry = <filename, inode number>
 - has variable size due to filenames
 - Stored in a directory file
 - is a hard links
- *Compare and contrast inode and directory entries.*
 - * Content
 - * fixed or variable lengths
 - * their storage containers
- *Q? Why separate filenames from inodes?*
 - * Can a file have multiple names?
 - * many directory entries? many inode numbers?

3.2 Disk structures: inodes, links, directories

- *Q? Why links?*
 - * Alias, i.e. multiple names for a file
 - * Exercise 3.6 (pp. 95)
 - Programs assume /usr/include/X11 for X header files
 - but Solaris 2 uses /usr/openwin/share/include/X11
 - Q? How can we port C programs using X to Solaris 2?
- *Q? What is a simple implementation?*
 - * two directory entries sharing a inode
 - * Called Hard links!
 - * Example 3.7, Fig. 3.5 (pp. 91-92)
 - * Problem: inodes number - not unique across partitions
- *Q? what a is unique name across entire file system?*

3.2 Disk structures: inodes, links, directories

- *Symbolic links*
 - * content of file = pathname of real file
 - * Fig./Example 3.8 (pp. 94)
- *Commands: ln , ln -s*
 - ln file1 anotherLink
 - ln -s sLink file1
- *Commands: rm (system call unlink())*
 - * Remove a hard link,
 - reduce hardlink reference count!
 - remove file if count = 0.
 - * Example:
 - rm /dirA/file1
 - rm sLink
 - rm anotherLink

3.3 Memory structures: file descriptors/pointers

- *File handles = logical names for device independent I/O*
 - * e.g. file descriptor, file pointer
- *3.3.1 File Descriptor = an index into a file descriptor table*
 - * Ex. STDIN_FILENO in unistd.h
 - * System calls: open, close, read, write, ioctl
myfd = open("/dirA/file1.dat", O_RDONLY);
- *3.3.2 File Pointer = pointer to a file structure,*
 - which refers to a buffer and a file descriptor value
 - * Is a handle to a handle!
 - * Fig. 3.12 (pp. 102), Example 3.11 (pp. 101)
 - * Ex. Stdin, stdout in stdio.h
 - * System calls: fopen, fclose, fread, fwrite, fscanf, fprintf
- *Buffers*
 - * Delay write I/O to device until buffer is full
 - * Buffer size = 1 block for disk, 1 line for terminal
 - * lose data if system crashed before buffer is full
 - * System call fflush() to force I/O after write()

- *Ex.: Predict output of Examples 3.12, 3.13 (pp. 102-103)*
 - * Hint: stderr is not buffered, stdout is.

3.3 Memory structures: file descriptors/pointers

- *Unix tables for managing files: (Fig. 3.11, pp. 100)*
 - * Per process - (a) File descriptor table (FDT)
 - * Kernel: (b) System open file table (SOFT), (c) Inode table
 - * Offset (current location to read/write) in SOFT
 - * Allows two processes to share a file
 - 2 entries in SOFT - e.g. independent reading
 - 1 entry in SOFT - share offset, e.g. DBMS logfile

- *3.3.3 Inheritance of File Descriptors in fork()*
 - * Child FDT is a copy of parent process FDT
 - * Share SOFT entries, i.e. file-offsets
 - for files open at fork() time
 - not for files opened after fork()
 - * Fig. 3.13 and 3.14 (pp. 105-6)

- *Exercise 3.11 (pp. 101)*
 - A process files a file for reading and then forks.
 - How do reads and writes by two process interact?

- *Q? Are file pointers inherited? (Hint: Are buffers inherited?)*

3.4-5 Filters, Redirection, Pipes

- *Benefits of device independent*
- *3.4 Filter = program uses standard I/O for read/write,*
 - all parameters passed via command line args,
 - Requires no user interaction,
 - input data has no headers or trailers
 - * e.g. head, tail, more, sort, grep, awk
- *I/O Redirection*
 - * Shell symbols: >, <, >>, ...
 - * System call: dup2()
 - * Effect on per process FDT:
 - FDT Index 0, 1, and 2 are for standard I/O
 - These default to keyboard, terminal, terminal
 - Redirection changes these entries to disk files
- *Examples*
 - * Figure 3.15 (pp. 107) - FDT for 'cat > my.file'
 - * Example 3.17 (pp. 108) - use of dup2()

3.4-5 Filters, Redirection, Pipes

- *Pipe: A special type of file*
 - * A communication buffer w/ file descriptors: fd0, fd1
 - * Unidirectional: Data written on fd1 is read from fd0
 - * first-in-first-out property
 - * Has no permanent name (Named pipes = FIFOs (sec. 3.9))
- *Use: let filters work together in a single command*
 - * Command line: `ls -l | sort -n +4`
 - * 'ls' and 'sort' share a pipe, say <fd0, fd1>
 - * 'ls' redirect its stdout to 'fd1'
 - * 'sort' redirects its stdin to 'sort'
- *System call: pipe()*
 - * Example 3.20 (pp. 110-1) : Code showing use of
 - `pipe()`, `fork()`, STDI/O redirections via `dup2()`
 - * Fig. 3.18-20 (pp. 111-2) show effects on FDTs

3.4-5 Filters, Redirection, Pipes

- *Generalization of Pipes*
 - * Pipes are very successful, i.e. widely used
 - * Named pipes
 - * Bidirectional pipes
 - * Communication across a network of machines
- *3.9 Named pipes, i.e. FIFOs*
 - * first-in first-out files
 - * Create a fifo with a filename and permissions
 - * Persists after creator process exits
 - * Command/system call - mkfifo: Example 3.25, pp. 120
 - * Q? Name an advantage of FIFOs over pipes.
 - * Unrelated processes (non parent-child) can share it!
- *Bidirectional: Data written on fd1 is read from fd0*
 - and data written on fd0 can be read from fd1
 - * See STREAMS in chapter 12.
- *Network Communication*
 - * sockets() are generalization of pipes

* Chapter 12 (Client-Server Communications)

3.6-8 File operations: blocking/non-blocking

- *Blocking read/write is default, i.e.*
 - read() waits until input is available
 - * Not suitable for server processes (e.g. mail)
 - which read from a ready file-descriptor among many
- *System calls read() and write()*

```
while ((br = read(from_fd, buffer, BLKSIZE) > 0)
    if ( write(to_fd, buf, bytesread) <= 0)
        break;
```
- *Non-blocking I/O*
 - * Allow read() to return immediately
 - if no input is available in buffer
 - * System calls fcntl() - Ex. 3.22 (pp. 116)

```
if ( fcntl(fd, F_GETFL, 0) == -1)
    perror("Could not get flags for fd");
else{ fd_flags |= O_NONBLOCK;
      if ( fcntl(fd, F_SETFL, fd_flags) == -1)
          perror("Could not set flags for fd");
      }
```
- *Alternative system call - select()*