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 structres: 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 desriptors 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).

3.1 Directory Operations

- *E. open, read, write, close*
  - System calls: opendir(), readdir(), closdir()
  - 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

- *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

- *What information is in inodes?*

  * 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 Data Structure for File

- *The data-structure for file should support*
  - read(), write(), bulk read
  - at random location, e.g. head, tail, lseek
- *Choices: data-structure from 1902/3321*
  - Linear arrays or lists
  - trees - (binary or nry), balanced?, fixed depth?
- *Unix Data-structure to search file blocks*
  - Unbalanced tree of depth 3
  - Trade-of between small and large files
  - Interesting exercise 3.3 (pp. 87)
- *Q? How will one get first byte? last byte? Nth byte?*
- *Compare this data-structure to balanced trees of arbitrary depth.*
  - Maximum file sizes
  - Complexity of adding information at end/start

3.2 Disk structures: directory entries

- *3.2.1 Directory = list of directory entries*

  * Directory entry = <filename, inode number>
  - has variable size due to filenames
  - Stored in a special file

- *Compare and contrast inode and directory entries.*

  * Content

  * fixed or variable lengths

  * their storge containers

- *Q? Why separate filenames from inodes?*

  * Can a file have multiple names?

  * many dirctory entries? many inode numbers?

3.2 Disk structures: hard links

- *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: symbolic links

- *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 data structres for open files

- *3 Unix tables for managing files: (Fig. 3.11, pp. 100)*

  * OS kernel: (1) In-memory Inode table
  - Caches inode information from disk structures

  * OS kernel: (2) System open file table (SOFT),
  - <file status flag, current offset, ptr to Inode entry>
  - status flags = read, write, append, sync, nonblocking etc.

  * Per process - (3) File descriptor table (FDT)
  - <file descriptor flags, pointer to a SOFT entry>
  - descriptor flags (0/1): 0 => close fd on exec()

- *Why separate per process FDT from kernel SOFT?*

  * process specific I/O redirection

- *Why separate SOFT from Inode table?*

  * Allow 2 processes to share a file and its buffer (e.g. pipe)
  - 2 entries in SOFT - e.g. independent reading
  - 1 entry in SOFT - share offset, e.g. DBMS logfile

3.3 Memory structres: Buffers

- *Why Buffer I/O ?*
  * Slow, high fixed overhead.
- *Analogy: Suppose you eat one candy every day*.
  * Buying your favourite candy in Mall take 30 minutes
  * Q? How often do we want to go to the Mall?
  * Not often! Buy candy for a week in each visit!
- *Buffer size , Buffering*
  * Buffer for disk I/O = a block, e.g. 4Kbyte
  * Buffer for Keyboard/screen = line (i.e. carriage return)
  * Process I/O request until buffer is full
  * stderr is not buffered!

3.3 Memory structres: file handles

- *File handles = logical names for device independent I/O*
  - * returned by open("filename", ...)
  - * used by read/write/close to identify a file
  - * Types of handles: (1) file decsriptor, (2) file pointer
- *3.3.1 File Descriptor = an index into FDT*
  - * POSIX Include file: unistd.h
  - * Symbolic names: STDIN_FILENO, STDOUT_FILENO, ...
  - * System calls: open, close, read, write, ioctl
- *System call open() (specs on pp. 97)*
  - * Usage Example 3.10 (pp. 98)
  - * Returns file descriptor
  - * Argument 1 : filename (string)
  - * Argument 2: oflag - permissions for user
    - bit constants: O_RDONLY, OWRONLY, O_RDWR, O_APPEND, O_NONBLOCK, ...
  - * Argument 3: fd_mode - permissions for group, other
    - bit constants: Table 3.1 (pp. 99)

3.3 Memory structres: file handles

- *3.3.2 File Pointer = <file descriptor, memory buffer>*
  * Fig. 3.12 (pp. 102), Example 3.11 (pp. 101)
  * ANSI C Include file: stdio.h
  * Symbolic names: stdin, stdout, stderr
  * library routine: fopen, fclose, fread, fwrite, fscanf, fprintf
  * These call read()/write() in turn!
- *Should each fread/fwrite lead to system call read/write ?*
  * Additional Buffering is used to reduce system calls.
- *Note 2 kinds of buffers*
  * (A) Used by device (e.g. disk controller)
  * (B) Used by ANSI C to reduce calls to read/write
- *Avoid additional buffering by ANSI C runtime*
  * fputs()
  * stderr

3.3 Memory structres - Exercises

- *Ex.: Predict output of Examples 3.12, 3.13 (pp. 102-103)*
- *Q?Which table (Inode/SOFT/FDT) entries has :*

  * process access permissions for a file
  * memory buffer and next byte to be read/written
  * owning user, pointers to disk blocks

- *What are the disadvantages of buffering?*

  * Revisit test for last bullet (Lab. 1, Section 2.12, pp. 70)
  * lose data if system crashed before buffer is full
  - System call fflush() to force I/O after write()
  * Real-time I/O is harder

3.3.3 Memory structres and fork()

- *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 opens a file for reading and then forks.

  * How do reads and writes by two process interact?

- *Q? Are file pointers inherited?*

  * Are buffer contents inherited?

  * Are buffer for files opened before fork shared?

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 ( fnctl(fd, F_GETFL, 0) == -1)
  perror("Could not get flags for fd");
else{ fd_flags |= O_NONBLOCK;
      if ( fnctl(fd, F_SETFL, fd_flags) == -1)
          perror("Could not set flags for fd");
      }
```

- *Alterntive system call - select()*