**APUE Chapter 3: File I/O**

**3.1) Introduction**

* functions to open, read, write, and close a file, etc.
* open, read, write, lseek, and close
* unbuffered read and write, invokes a system call in kernel, part of POSIX.1 and Single UNIX Specification
* Atomic operations; file sharing
* More functions: dup, fcntl, sync, fsync, ioctl

**3.2) File Descriptors**

* A non-negative integer
* Returned when file is opened or created, used as argument to read and write functions
* 0 = standard input, 1 = standard output, 2 = standard error
* 0, 1, 2 to be replaced by STDIN\_FILLNO, STDOUT\_FILLNO, STDERR\_FILENO (defined in the <unistd.h>)
* Ranges from 0 – OPEN\_MAX-1

**3.3) open and openat Functions**

* Call to open or create a file
* Int open(const char *\*path*, int *oflag*, …/\* mode\_t *mode* \*/);
* Int openat(int *fd*, const char *\*path*, int *oflag*, …/\* mode\_t *mode* \*/);
* Both above function calls return file descriptors if OK, otherwise -1
* Last argument used only when file is being created
* Path = name of the file to open/create
* oflag = specifies the options used by the functions; formed by ORing tgt one or more of the constants from <fcntl.h>
  + O\_RDONLY, O\_WRONLY, O\_RDWR, O\_EXEC, O\_SEARCH
* File descriptor returned by these two functions are guaranteed to be the lowest-numbered unused descriptor
* fd = distinguishes openat from open function
  + if path = absolute pathname, then fd is ignored and openat = open
  + path = relative pathname, fd = file descriptor specifying starting location in the file system where the path is to be evaluated; obtained by opening directory where the relative pathname is to be evaluated
  + path = relative pathname, fd = special value AT\_FDCWD; pathname evaluated at working directory and openat = open function
* openat function address two problems:
  + gives threads a way to use relative pathnames to open files in directories instead of current working directory
  + provides a way to avoid time-of-check-to-time-of-use (TOCTTOU) errors
* TOCTTOU errors; program is vulnerable when two file-based functions are called
  + Second call depends on results of first call
  + Both calls not atomic so the file can change between the calls
  + Invalidating the results of the first call 🡪 program error
  + Deals with attempts to subvert file system permissions by tricking privileged program into reducing permissions of a privileged file or modifying a privileged file to open up a security hole
* Filename and pathname truncation
  + Early releases of System V allowed silent truncation of filename beyond the max count of names
  + BSD-derived systems will return error status – errno = ENAMETOOLONG
  + The former results in the loss of the original file name
* POSIX.1 determines if name should be truncated or error is returned using constant \_POSIX\_NO\_TRUNC
* If \_POSIX\_NO\_TRUNC is in effect, then error is returned

**3.4) creat function**

* Create a new file
* Int creat(const char *\*path*, mode\_t *mode*);
* Returns file descriptor open for write-only is OK, otherwise -1
* Deficiency is that it is created for write only
* Equivalent to open(path, O\_WRONLY | O\_CREAT | O\_TRUNC, mode)
* Better form: open(path, O\_RDWR | O\_CREAT | O\_TRUNC, mode)

**3.5) close function**

* to close an open file
* int close(int *fd*)
* returns 0 if ok, -1 if error
* releases any record locks that the process may have on the file
* when process terminates, all open files are closed by kernel

**3.6) lseek function**

* current file offset – a non-negative integer that measures the number of bytes from the beginning file
* read and write functions usually start at the current file offset and increments by number of bytes being read/written
* default file offset = 0 when opened
* off\_t lseek(int *fd*, off\_t *offset*, int *whence*)
* returns new file offset if OK, -1 if error
* offset value depends on whence argument
  + SEEK\_SET – offset = offset bytes from beginning of file
  + SEEK\_CUR – offset = current value + offset argument
  + SEEK\_END – offset = size of file + offset argument
* Lseek returns -1 if file descriptor refers to pipe, FIFO or socket
* Usually file offset must be non-negative

**3.7) read function**

* To read data from an open file
* ssize\_t read(int *fd*, void *\*buf*, size\_t *nbytes*);
* returns number of bytes read, 0 if end of file (EOF), -1 if error
* cases where number of bytes read is less than amount requested:
  + if EOF is reached before the number of bytes requested is reached
  + when reading from terminal device – read one line at a time
  + when reading from a network – buffering may cause less than requested to be returned
  + when reading from Pipe or FIFO – if Pipe contains fewer bytes
  + when reading from a record-oriented device – returns a single record at a time
  + when interrupted by a signal and a partial amount has already been read
* starts at file’s current offset, increments by amount of bytes read before returning
* original function: int read(int *fd*, char *\*buf*, unsigned *nbytes*)
  + second argument changed to void \* to be consistent with ISO C
  + return values required to be signed integer
  + third argument changed to signed integer

**3.8) write function**

* writes data into an open file
* ssize\_t read(int *fd*, void *\*buf*, size\_t *nbytes*);
* returns number of bytes written if ok, -1 if error
* common errors: filling up a disk, or exceeding file size limit
* starts at current offset
* if O\_APPEND is specified when file was opened, then file offset = current EOF before each write operation

**3.9) I/O Efficiency**

* Figure 3.5 – copying a file using only read and write functions
  + Assumes already set up by shell before program is executed; prevents program from opening input and output files; allows user to take advantage of shells I/O’s redirection facilities
  + Doesn’t close file, uses the feature that kernel closes all files when process terminates
  + Example works for both text and binary; no difference to the UNIX kernel
* Increasing buffer size beyond 4096 bytes has little positive effects
* Read-ahead property – elapsed time for small buffer sizes (32 bytes, etc.) is as good as time elapsed for larger buffer sizes

**3.10) File Sharing**

* UNIX supports sharing of open file among different processes
* Data structures used by kernel for all I/O
  + Every process has an entry in the process table; each entry has file descriptors; which is associated with the file descriptor flag, and a pointer to a file table entry
  + Kernel maintains a file table for all open files; each entry contains the file status flags, current file offset, a pointer to the v-node table entry for the file
  + Each open file has a v-node structure that contains info on the type of file and pointers to functions that operate on the files, also contains i-node; information is read from disk when file is opened
* It is possible for more than one file descriptor entries to point at one file table entry
  + Can happen after a fork where the parent and child share the same file table entry for each open descriptor

**3.11) Atomic Operations**

* pread and pwrite to allow applications to seek and perform I/O atomically
* ssize\_t pread(int *fd*, void *\*buf*, size\_t *nbytes*, off\_t *offset*);
  + returns number of bytes read, 0 if EOF, -1 if error
* ssize\_t pwrite(int *fd*, const void *\*buf*, size\_t *nbytes*, off\_t *offset*);
  + returns number of bytes written, 0 if EOF, -1 if error
* calling pread = calling lseek + read except there is no way to interrupt the two functions when calling pread, and current file offset is not updated
* calling pwrite = calling lseek + write with the same exceptions
* Creating a file
  + open will fail if the file already exists, when O\_CREAT and O\_EXCL has been specified
  + If file is created by another process between open and creat then data is erased when creat is executed
* Atomic operations = operation composed of multiple steps
* Either all are performed (successfully) or none are performed (failed)

**3.12) dup and dup2 Functions**

* Duplicate an existing file descriptor
* int dup(int *fd*)
* int dup2(int *fd*, int *fd2*)
* both returns new file descriptors if ok, -1 if error
* dup returns guaranteed lowest file descriptor value
* dup2 allows user to specify what file descriptor value; it is also an atomic operation
* if *fd2* is already opened, it is first closed
* if *fd* == *fd2*, then dup2 returns *fd2* without closing it
* Otherwise, FD\_CLOEXEC file descriptor flag is cleared for *fd2*, so that *fd2* is left open if the process calls exec
* Return value will share same file table entry as *fd*
* Another way to duplicate file descriptors is to call fcntl
  + Dup(fd) == fcntl(fd, F\_DUPFD, 0);
  + Dup2(fd, fd2) == close(fd2); fcntl(fd, F\_DUPFD, fd2);
* There are also some errno differences between dup2 and fcntl