Files and Directories

File Interfaces in Unix

- Unix has two main mechanisms for managing file access.
- file pointers: standard I/O library
  - You deal with a pointer to a FILE structure that contains a file descriptor and a buffer.
  - Use for regular files (more abstract and portable)
- file descriptors: low-level
  - Each open file is identified by a small integer.
  - Use for pipes, sockets.

stdin, stdout, stderr

- 3 files are automatically opened for any executing program:

	stdio name	File descriptor
Standard input	stdin	0
Standard output	stdout	1
Standard error	stderr	2

- Reading from stdin by default comes from the keyboard
- Writing to stdout or stderr by default goes to the screen.

Buffering

- un-buffered – output appears immediately
  - `stderr` is not buffered
- line buffered – output appears when a full line has been written.
  - `stdout` is line buffered when going to the screen
- block buffered – output appears when a buffer is filled.
  - normally output to a file is block buffered
  - `stdout` is block buffered when redirected to a file.

File Operations

- For regular files use: fopen, fread, fwrite, fprintf, fgets, fscanf, fclose.

```
FILE *fopen(const char *filename, const char *mode);


char *fgets(char *s, int size, FILE *stream);
```

- reads the next line from a file pointer
  - It reads at most size -1 characters
  - Reading stops after a newline or EOF
  - Appends a ‘\0’ character at the end of the string.

Reading from a file?

- If we want to read from somewhere other than `stdin`, we need to open a file.
- How should we specify the filename?
  - `argv[0]` == name of program
  - `argv[1]` == first argument

```
int main(int argc, char **argv) {
    if(argc != 2)
        fprintf(stderr, "Usage: %s <filename>\n",
                argv[0]);
        exit(1);
    }
}
```

stdio

- To open a file:

```
FILE *fopen(const char *filename,
            const char *mode);
```

  - `filename` identifies the file to open.
  - `mode` tells how to open the file:
    - "r" for reading, "w" for writing, "a" for appending
  - returns a pointer to a `FILE` struct which is the handle to the file. This pointer will be used in subsequent operations.
  - To close a file: `void fclose(FILE *stream);`

Example

```
int main(int argc, char **argv)
{
    char *sptr, name[MAX];
    FILE *fp;


    if(argc != 2) {
      fprintf(stderr, "Usage: %s <file>\n”, argv[0]);
      exit(1);
    }
    if((fp = fopen(argv[1], "r")) == NULL) {
      perror(argv[1]);
      exit(1);
    }
```

Example (cont'd)

```
    while((sptr = fgets(name, MAX, fp)) != NULL) {
        /* strip the newline */
        sptr = strchr(name, '\n');
        *sptr = '\0';
        printf("%s\n", reverse_name(name));
    }
    return 0;
}
```

Error Handling

- Most system calls return -1 if an error occurs. (A few return NULL.)
- **errno** – global variable that holds the numeric code of the last system call.
- Every process has **errno** assigned to zero at process creation time.
- When a system call error occurs, **errno** is set.
- A successful system call never affects the current value of **errno**.
- An unsuccessful system call always overwrites the current value of **errno**.
- **Always check the return value of system calls!**

`perror()`

- Library routine:
- `void perror( char *str )`
- perror displays `str`, then a colon(:), then an English description of the last system call error as defined in errno.h.
- Protocol
  - check system calls for a return value of -1
  - call perror() for an error description.

Binary I/O

- Recall that `fgets` reads characters.
- By contrast, `fread` and `fwrite` operate on bytes.

```
size_t  fread(void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

  - read `nmemb * size` bytes into memory at `ptr`

```
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

  - write `nmemb * size` bytes from `ptr` to the file pointer `stream`

Example

- It doesn't matter what the bytes contain!

```
/* write an integer to the file */
int num = 21;
n = fwrite(&num, sizeof(num), 1, fp);


/* write a struct to the file */
struct rec {
  char name[20];
  int num;
} r;
r.num = 42;
strncpy(r.name, "koala", 20);
n = fwrite(&r, sizeof(r), 1, fp);
```

Example

- We need to know how to interpret the bytes from a file when reading.

```
/* read an integer from the file */
int num;
n = fread(&num, sizeof(num), 1, fp);


/* read a struct from the file */
struct rec r;
n = fread(&r, sizeof(r), 1, fp);

/* display the contents of the variables */
printf("%d %s %d\n", num, r.name, r.num);
```

Moving around a file

- Most often, we read files from beginning to end and only write at the end of a file.

- If we are treating a file as a collection of records, it is often useful to be able to move to a particular position (byte) in the file.

Moving around a file

```
int fseek(FILE *stream, long
          offset, int whence);
```

- set the file position for `stream`
- add `offset` bytes to the position specified by `whence`
- `whence` can be
  - `SEEK_SET` - beginning of file
  - `SEEK_CUR` - current position in file
  - `SEEK_END` - end of file

File Position

```
long ftell(FILE *stream);
```

- Return the current position (in bytes) for the file pointed to by stream.

```
void rewind(FILE *stream);
```

- Set the file position to the beginning of the file.

stat()

- `int stat(const char *file_name,`
  `struct stat *buf);`
- need to allocate memory for the `stat` `struct` before passing it to `stat`
- `struct stat` contains many fields including `st_mode`
- Useful macros: `S_ISREG(modefield)`, `S_ISDIR(modefield)`

stat()

```
struct stat sbuf;
if(stat(pathname, &sbuf) == -1) {
  perror("stat");
}
if(S_ISREG(sbuf.st_mode)) {
  printf("Regular file\n");
}
```

- There are also defined variables for each of the permission sets. For example:

```
if(sbuf.st_mode & S_IRUSR) {
  printf("Owner can read file\n");
}
```

Directories

- We will get to the information on directories later, but it goes with the file operations, so I'll leave the slides here.

Directory Operations

- Recall that a directory is a special kind of file.
- We can read directory entries using similar functions.
- For directories use:

```
DIR *opendir(const char *filename);
struct dirent *readdir(DIR *dirp);
```

- `readdir` works like `fread` on directory files. Each time `readdir` is called it returns a directory entry.

Ch. 4.4