CSci 4061
Introduction to Operating Systems

Input/Output: Low-level
Chapter 4

Low-level I/O

- System call interface
- High-level I/O `<stdio.h>` calls low-level interface
- You can call the low-level interface also
- Abstraction: source/sink for data, file descriptor

- Key Idea
  - I/O devices and descriptors
  - No notion of streams (`FILE`)
  - No formatted I/O, just bytes
  - More control (i.e. read 1 char, don't buffer)

Opening a “File”

```
#include <sys/types.h>
#include <stat.h> // depends on gcc version
#include <unistd.h>
#include <fcntl.h>
```

mode => permissions

// create a file if not there

```
int creat (char *pname, mode_t mode);
```

// open a file

```
int open (char *pname, int flags, mode_t mode);
```

pname, e.g. “/usr/jon/file.txt”
returns a file descriptor
failure: return -1, sets errno

Opening a File (cont'd)

- Flags:
  - O_RDONLY, O_WRONLY, O_RDWR, O_APPEND,O_CREAT
  - O_CREAT: will create file if not already there
  - O_TRUNC: O_WRONLY => truncates length to 0
  - O_SYNC: flush writes
  - O_NONBLOCK: non-blocking I/O

- Returns a file descriptor or `fd`, an integer
  - Special descriptors are 0, 1, 2 and correspond to `stdin`, `stdout`, and `stderr` respectively (from `<stdio.h>`)

Close

```
int close (int fildes);
```

Important to `close`:

there are a limited number of allowable open files and `fd`'s

All open files are closed when process exits

* assuming only this process has file open!

File Descriptor Mapping Table

How do I get an fd from a FILE *

- Easy
  - `int fileno(FILE *stream)`
    returns the associated `fd`

- The other way:
  - `FILE *fdopen(int fd, const char *mode)`

Read

```
ssize_t read (int filedes,
              char *buffer,
              size_t n);
```

Returns # of bytes actually read up to EOF or n
Returns 0 if at EOF, -1 if an error
`buffer` must be allocated: output parameter!
Reads are sequential w/r to current file pointer

Reads block by default

How can `read` fail?

Read Example

File “foo”:
Hello John Hello Rich Hello Jay

```
char buf1[12], buf2[12];
int fd, n1, n2, n3;
fd = open (“foo”, O_RDONLY);
n1 = read (fd, buf1, 11); // n1 is 11
n2 = read (fd, buf2, 11); // n2 is 11
n3 = read (fd, buf1, 11); // n3 is 9
```

Is `buf1` or `buf2` a string?

What are the values of `buf1` and `buf2`?

What did I forget to do in the code?

Read Example

```
// Count characters in a file (i.e. file size)
// good idea to read large chunks
#define BUFSIZE 512
void main () {
    int fd;
    int total = 0;

    fd = open (“somefile”, O_RDONLY);

    printf (“Size = %d\n”, total);
}
```

Read Example

```
#define BUFSIZE 512
void main () {
      int fd;
      int total = 0;
      ssize_t nread;
      char buffer [BUFSIZE];

      fd = open ("somefile", O_RDONLY);
      // loop until EOF
      while (nread = read (fd, buffer, BUFSIZE)) > 0)
            total += nread;  // why not BUFSIZE?
      printf ("Size = %d\n", total);
}
```

smaller or larger?

What don't you like about this solution?

Common read error

```
#define MAX_SIZE 1024
char *buffer;
ssize_t amt;
…
amt = read (fd, buffer, MAX_SIZE);
```

What will happen?

Write

```
ssize_t write (
    int filedes,
    const char *buffer, size_t n);
```

Returns number of bytes actually written

If this is < n, usually a problem

As always, -1 is an error

Write (cont'd)

As with reads, writes are done sequentially w/r to current file offset/pointer

octal

6: user/owner can r/w
4, 4: group, others can r

```
#define PERM 0644
char header1[512]="aaa…", header2[512]="bbb…";
int fd;
ssize_t w1, w2;
…
fd = open ("newfile", O_WRONLY|O_CREAT, PERM);
w1 = write (fd, header1, 512);
w2 = write (fd, header2, 512);
```

LOGICAL OR

Will overwrite any data in the file if it exists (unless `O_APPEND` or `O_TRUNC`)

aaaaaaaaaaaaaaaaa….bbbbbbbbbbbbbbbbbbb…<old stuff stays here>

Example: File Copy

```
// All headers …
#define BUFIZE 512
#define PERM 0644  // user can read/write, group/other can only read

void copyfile (const char *name1, const char *name2) {
        int infile, outfile;
        ssize_t nread;
        char buffer [BUFSIZE];
        infile = open (name1, O_RDONLY);
        outfile = open (name2, O_WRONLY|O_TRUNC|O_CREAT, PERM);
        while (nread = read (infile, buffer, BUFSIZE)) > 0)
                write (outfile, buffer, nread);
        close (infile);
        close (outfile);
}
```

Why O_TRUNC?

Call it: copyfile (“square_peg”, “round_hole”);

Writes are binary by default

```
int i = 5;

//output: a3^34
write (1, (char *)&i, sizeof(int));
…
// assume: a3^34 is on stdin
read (0, (char *)&i, sizeof(int));
// i = 5
```

OS Buffers writes

- Use O_SYNC flag on `open` in write mode

```
outfile = open (name2,
                O_WRONLY|O_TRUNC|O_CREAT|O_SYNC, PERM);
```

or at a point in time

- `int fsync (int fd);`

Symbolic names

- Don’t like 0, 1, 2
- STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO
- Analagous to stdin, stdout, stderr (for FILE *)

What else?

`open, close, creat, read, write`

What seems to be missing?