CSci 4061
Introduction to Operating Systems

Module 2: Input/Ouput

Input/Output: High-level

I/O Topics

- First, cover high-level I/O

- Next, talk about low-level device I/O

- I/O not part of the C language!

I/0 Device Abstraction(s)

- Abstraction: source/sink for data (raw bytes)
- Operations:
    - Open/close device
    - Read from device
    - Write to device
    - Control device

High-level I/O

- Go further with the abstraction
- Hide “device” or low-level I/O
- Low-level I/O abstraction: source/sink for data
- High-level: more than raw bytes --
- Features
    - Stream abstraction
    - Formatted/typed I/O
    - String-based I/O
    - Line-oriented buffering

Streams and Files

- Unix `FILE` object
  - Delivers an ordered sequence of bytes
  - Defined in `<stdio.h>`
  - User-space library
  - Built on top of low-level file descriptors

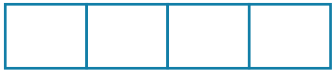
- Three default streams
  - `stdin`
  - `stdout`
  - `stderr`

Inside a `FILE`

- Points to actual file
- Current offset
- Mode : read, write, append, etc.
- Buffers

- Write Buffering (open for write)
  - Internal character buffer of size BUFSIZE
  - Writes are done to in-memory buffer
    - When buffer is full (BUFSIZE), `write` out buffer
    - Or if line-buffered stream (e.g. `stdout`) when ‘\n’ is written

Inside a `FILE` (cont'd)

- Read Buffering (open for read)
  - Reads are done from in-memory read buffer
  - When buffer is empty, we `read` a chunk of BUFSIZE 
  - Why is the use of an internal buffer for I/O advantageous?

Buffering: FFLUSH

- Sometime we want to “force” buffer to spill to the OS - why?
  - see data immediately (OS auto-flushes K sec)

```
int fflush (FILE *stream);


fprintf (F, “your output is %d”);
fflush (F); // force it
```

`stderr` always causes buffer to be flushed

`stdout` with ‘\n’ as well

OPEN/CLOSE

```
#include <stdio.h>
```

“w” will create file
if not there …

```
// NULL on failure, errno also set
FILE *fopen (const char *filename,
            const char *mode);
```

mode = “r”, “w”, “a” (others, r+, w+)

truncate to 0 length

```
// returns 0 on success, EOF otherwise (-1)
int fclose (FILE *stream);
```

I/O Access

- Assumed to be sequential

- The `FILE`* keeps track of the current file offset for read and write

Character-based I/O

```
#include <stdio.h>
// SUBSET of calls: return EOF on failure or end
int fgetc (FILE *stream);
int fputc (int char, FILE *stream);
int getc ();             // STDIN
int putc (int char); // STDOUT
```

Which header are things located?

```
    >man getc
```

String-based I/O

// reads `nsize-1` characters or up to a newline
// if newline, then '\n' goes into `buf`
// appends '\0' at the end of `buf`
// caller allocs `buf`

sets errno

// returns buf or NULL if at EOF or an error occurs

```
char *fgets (char *buf, int nsize, FILE *inf);
```

// outputs `buf` - better be '\0' terminated   what must be true of `buf`?
// returns last char written (+ #) or EOF if an error occurs

```
int fputs (const char *buf, FILE *outf);
```

sets errno

Also: `gets, puts`

Importance of newline ‘\n’

- `stdin` is a line-oriented device
  - ‘\n’ (EOL), input is read when ‘\n’ is seen or EOF
  - ^D is EOF

  - `fgets` on `stdin` - will include ‘\n’ in the string!
  - `gets` does not

String library functions

- Not system calls
- On your own: remember to allocate, ‘\0’
- `strcpy, strncpy, strlen, strtok, strcmp, …,` *`strdup`*

What about I/O for data types?

- Formatted I/O!
- Formatted output and “output” strings

```
int fprintf (FILE *outf,
             const char *fmt,
             args)

int sprintf (char *string, // alloc!
             const char *fmt,
             args)
```

Also: `printf`

Formatted Output (cont’d)

Formats: %d, %f %c, %x, others

```
int x=4;
char str [100];
FILE *f;
F = fopen (“myfile”, “w”);
fprintf (F, “%d”, x);
sprintf (str, “%d %d %s”, 12, x,
                “hello”);  => “12 4 hello”
```

FORMAT fields must match in type and #

Formatted Input

Formatted input and “input” strings

//Read from file

```
fscanf (FILE *in, const char *fmt,
<ptr args … allocated>);
```

// ”Read from” (i.e. parse) string

```
sscanf (const char*, const char *fmt,
        <ptr args … allocated >);
```

Also: `scanf`

Formatted Input (cont’d)

```
int i1, i2;
float flt;
char str1[10], str2[10];
char *str1, *str2; // ok?

sscanf (“11 12 34.07 keith ben”,
        “%d %d %f %s %s”,
        &i1, &i2, &flt, str1, str2);
```

?

Do not forget to allocate and use &!

Project #1

Binary I/O

- We like ascii char-based files: `cat`, `more`, `emacs`, …
- But they take up space
  - integer: 1234567890
- Use binary I/O to read/write fewer bytes which saves space and is faster
- Works for fixed-size data items and lots of ‘em
  - but must be memory-contiguous (i.e. an array)

```
fread/fwrite (void *buffer,
      size_t size, size_t nitems,
      FILE *f);
```

Binary I/O Example

```
typedef struct S{
        int ss;     // 8 digits
        int phone;  // 10 digits
} info_t;
info_t mine = {12345678, 5384937474};
// fopen F for write
fprintf (F, “%d %d”, mine.ss, mine.phone);
fwrite ((void *)&mine, sizeof (info_t), 1, F);
```

ascii file: 12345678 538493747 = (8+10)*1 bytes

binary file: ^a93e&^%8 = (4+4)*1

```
fread ((void *)&mine, sizeof (info_t), 1, F);
printf (“%d %d\n”, mine.ss, mine.phone);

suppose info_t mine [100000];
```

Random I/O

- Thusfar, all I/O was assumed to be sequential
  - `fgets, fgets,` … returns consecutive lines
  - current file ptr advances sequentially
- Sometimes we need random I/O
- Think of any examples of applications that access data in a non-sequential pattern?

Random I/O

- Random I/O
  - Advance file offset pointer w/o reading/writing

```
#include <stdio.h>

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

new offset in bytes

whence of one SEEK_SET, SEEK_CUR, SEEK_END

from the beginning

// other call: what is the current file offset?

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