I/O Management

Goals of this Lecture

- Help you to learn about:
  - The Unix **stream** concept
  - Standard C I/O functions
  - Unix system-level functions for I/O
  - How the standard C I/O functions use the Unix system-level functions
  - Additional abstractions provided by the standard C I/O functions

Streams are a beautiful Unix abstraction

Stream Abstraction

- Any source of input or destination for output
  - E.g., keyboard as input, and screen as output
  - E.g., files on disk or CD, network ports, printer port, …
- Accessed in C programs through file pointers
  - E.g., `FILE *fp1, *fp2;`
  - E.g., `fp1 = fopen("myfile.txt", "r");`
- Three streams provided by stdio.h
  - Streams **stdin**, **stdout**, and **stderr**
    - Typically map to keyboard, screen, and screen
  - Can redirect to correspond to other streams
    - E.g., stdin can be the output of another program
    - E.g., stdout can be the input to another program

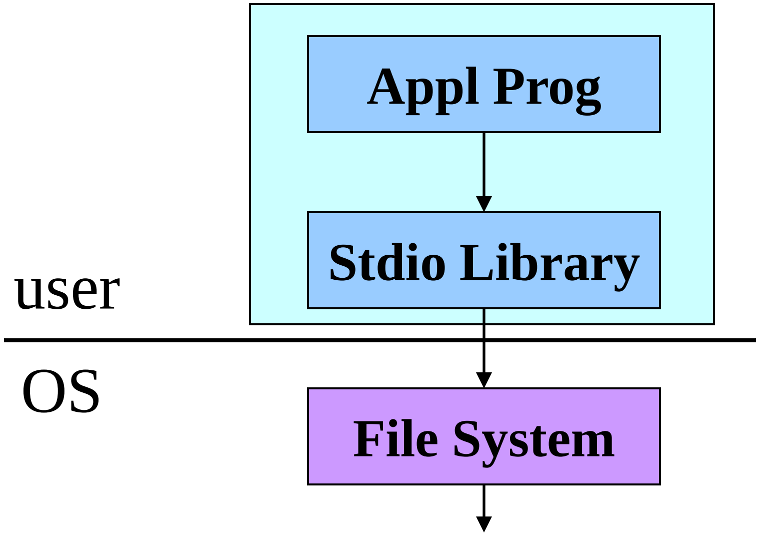
Sequential Access to a Stream

- Each stream has an associated file position
  - Starting at beginning of file (if opened to read or write)
  - Or, starting at end of file (if opened to append)

- Read/write operations advance the file position
  - Allows sequencing through the file in sequential manner
- Support for random access to the stream
  - Functions to learn current position and seek to new one

Standard I/O Functions

- Portability
  - Generic I/O support for C programs
  - Specific implementations for various host OSes
  - Invokes the OS-specific system calls for I/O
- Abstractions for C programs
  - Streams
  - Line-by-line input
  - Formatted output
- Additional optimizations
  - Buffered I/O
  - Safe writing



Example: Opening a File

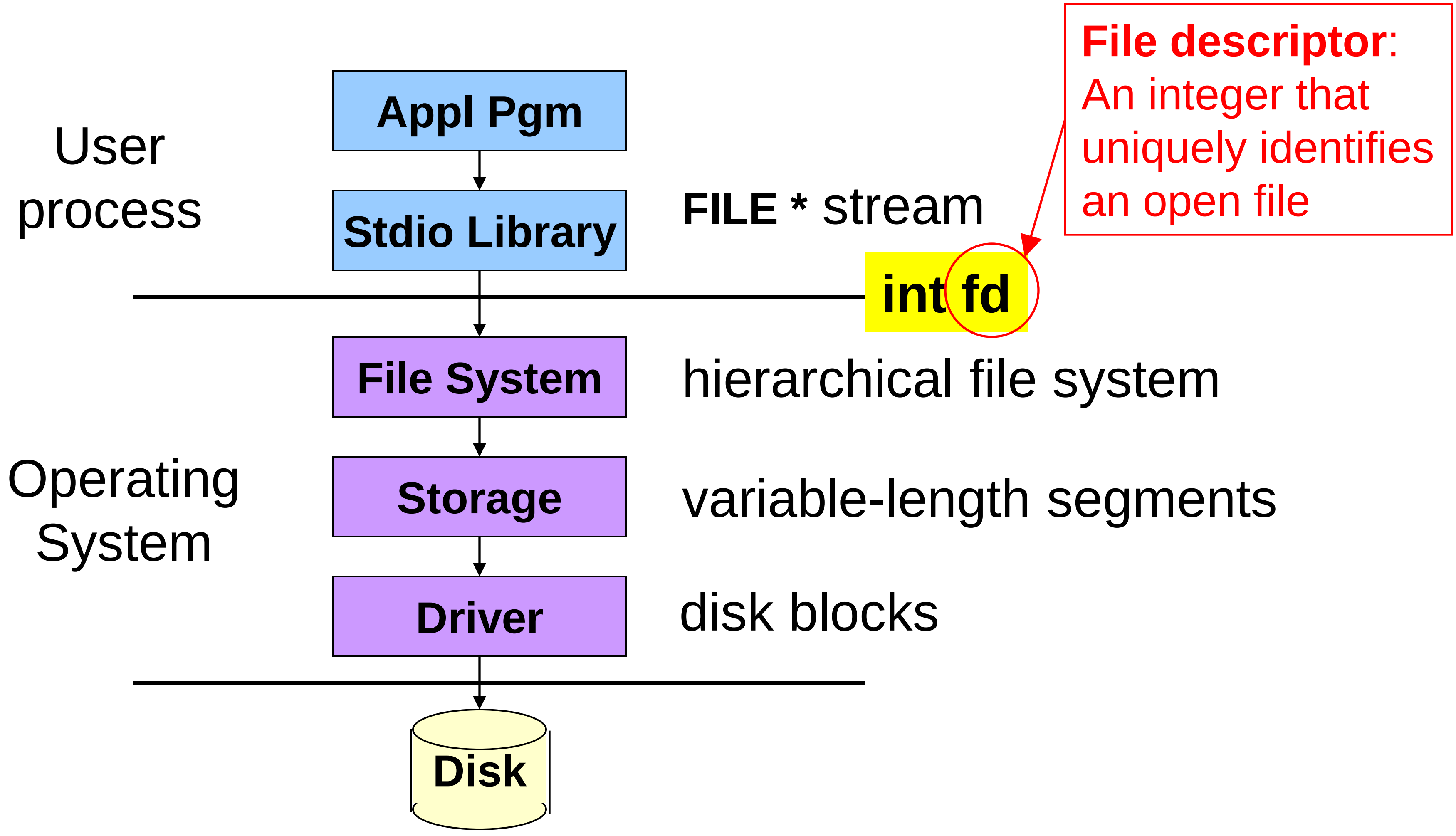
- **`FILE *fopen("myfile.txt", "r")`**
  - Open the named file and return a stream
  - Includes a mode, such as “r” for read or “w” for write
- Creates a FILE data structure for the file
  - Mode, status, buffer, …
  - Assigns fields and returns a pointer
- Opens or creates the file, based on the mode
  - Write (‘w’): create file with default permissions
  - Read (‘r’): open the file as read-only
  - Append (‘a’): open or create file, and seek to the end

Example: Formatted I/O

- **`int fprintf(fp1, "Number: %d\n", i)`**
  - Convert and write output to stream in specified format
- **`int fscanf(fp1, "FooBar: %d", &i)`**
  - Read from stream in format and assign converted values

- Specialized versions
  - **`printf(…)`** is just **`fprintf(stdout, …)`**
  - **`scanf(…)`** is just **`fscanf(stdin, …)`**

Layers of Abstraction



System-Level Functions for I/O

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

- Create a new file named **`pathname`**, and return a file descriptor

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

- Open the file **`pathname`** and return a file descriptor

`int close(int fd);`

- Close **`fd`**

`int read(int fd, void *buf, int count);`

- Read up to **`count`** bytes from **`fd`** into the buffer at **`buf`**

`int write(int fd, void *buf, int count);`

- Writes up to **`count`** bytes into **`fd`** from the buffer at **`buf`**

`int lseek(int fd, int offset, int whence);`

- Assigns the file pointer of **`fd`** to a new value by applying an **`offset`**

Example: `open()`

- Converts a path name into a file descriptor
  - **`int open(const char *pathname, int flags, mode_t mode);`**
- Arguments
  - Pathname: name of the file
  - Flags: bit flags for **`O_RDONLY`**, **`O_WRONLY`**, **`O_RDWR`**
  - Mode: permissions to set if file must be created
- Returns
  - File descriptor (or a -1 if an error)
- Performs a variety of checks
  - E.g., whether the process is entitled to access the file
- Underlies **`fopen()`**

Example: `read()`

- Reads bytes from a file descriptor
  - **`int read(int fd, void *buf, int count);`**
- Arguments
  - File descriptor: integer descriptor returned by **`open()`**
  - Buffer: pointer to memory to store the bytes it reads
  - Count: maximum number of bytes to read
- Returns
  - Number of bytes read
    - Value of 0 if nothing more to read
    - Value of -1 if an error
- Performs a variety of checks
  - Whether file has been opened, whether reading is okay
- Underlies **`getchar()`** , **`fgets()`**, **`scanf()`** , etc.

Example: A Simple `getchar()`

```
int getchar(void) {
    char c;
    if (read(0, &c, 1) == 1)
        return c;
    else return EOF;
}
```

- Read one character from **`stdin`**
  - File descriptor **`0`** is **`stdin`**
  - **`&c`** points to the buffer
  - **`1`** is the number of bytes to read
- Read returns the number of bytes read
  - In this case, **`1`** byte means success

Making `getchar()` More Efficient

- Poor performance reading one byte at a time
  - Read system call is accessing the device (e.g., a disk)
  - Reading one byte from disk is very time consuming
  - Better to read and write in *larger chunks*
- Buffered I/O
  - Read a large chunk from disk into a buffer
    - Dole out bytes to the user process as needed
    - Discard buffer contents when the stream is closed
  - Similarly, for writing, write individual bytes to a buffer
    - And write to disk when full, or when stream is closed
    - Known as "flushing" the buffer

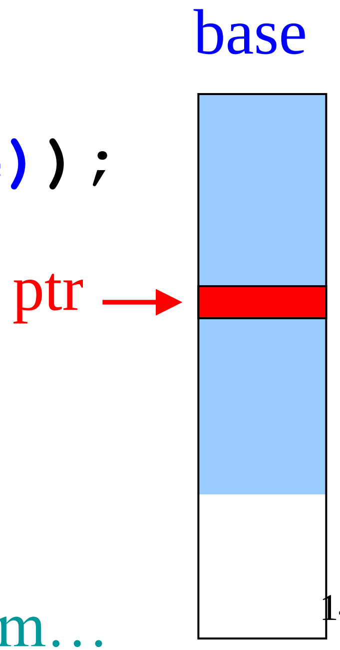
Better `getchar()` with Buffered I/O

```
int getchar(void) {
    static char base[1024];
    static char *ptr;
    static int cnt = 0;

    if (cnt--) return *ptr++;

    cnt = read(0, base, sizeof(base));
    if (cnt <= 0) return EOF;
    ptr = base;
    return getchar();
}
```

persistent variables



But, many functions may read (or write) the stream…

Details of FILE in stdio.h (K&R 8.5)

```
#define OPEN_MAX 20  /* max files open at once */

typedef struct _iobuf {
   int  cnt;      /* num chars left in buffer */
   char *ptr;     /* ptr to next char in buffer */
   char *base;    /* beginning of buffer */
   int  flag;     /* open mode flags, etc. */
   char fd;       /* file descriptor */
} FILE;
extern FILE _iob[OPEN_MAX];

#define stdin  (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
```

A Funny Thing About Buffered I/O

- The standard library also buffers **output**; example:

```
int main(void) {
   printf("Step 1\n");
   sleep(10);
   printf("Step 2\n");
   return 0;
}
```

- Run "a.out > out.txt &" and then "tail -f out.txt"
  - To run **`a.out`** in the background, outputting to **`out.txt`**
  - And then to see the contents on **`out.txt`**
- Neither line appears till ten seconds have elapsed
  - Because the output is being buffered
  - Add **`fflush(stdout)`** to flush the output buffer
  - **`fclose()`** also flushes the buffer before closing

Summary

- System-level I/O functions provide simple abstractions
  - Stream as a source or destination of data
  - Functions for manipulating streams
- Standard I/O library builds on system-level functions
  - Calls system-level functions for low-level I/O
  - Adds buffering
- Powerful examples of abstraction
  - Application pgms interact with streams at a high level
  - Standard I/O library interact with streams at lower level
  - Only the OS deals with the device-specific details