Standard I/O Library Implementation

COS 217

Goals of Today's Lecture

- Challenge: generic I/O support for C programs
  - Provide C programs with functions for input and output
    - Stream concept, line-by-line input, formatted output, ...
  - Implement across a variety of host OSes
- Solution: abstraction, and division of functionality
  - Standard I/O
    - ANSI C standard model and I/O functions for files
      - Specific C implementation for each different system
    - Additional features (e.g., buffered I/O and safe writing)
  - Low-level I/O
    - System calls that invoke OS services
      - UNIX examples: open, close, read, write, seek

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

Example Stdio Functions on Streams

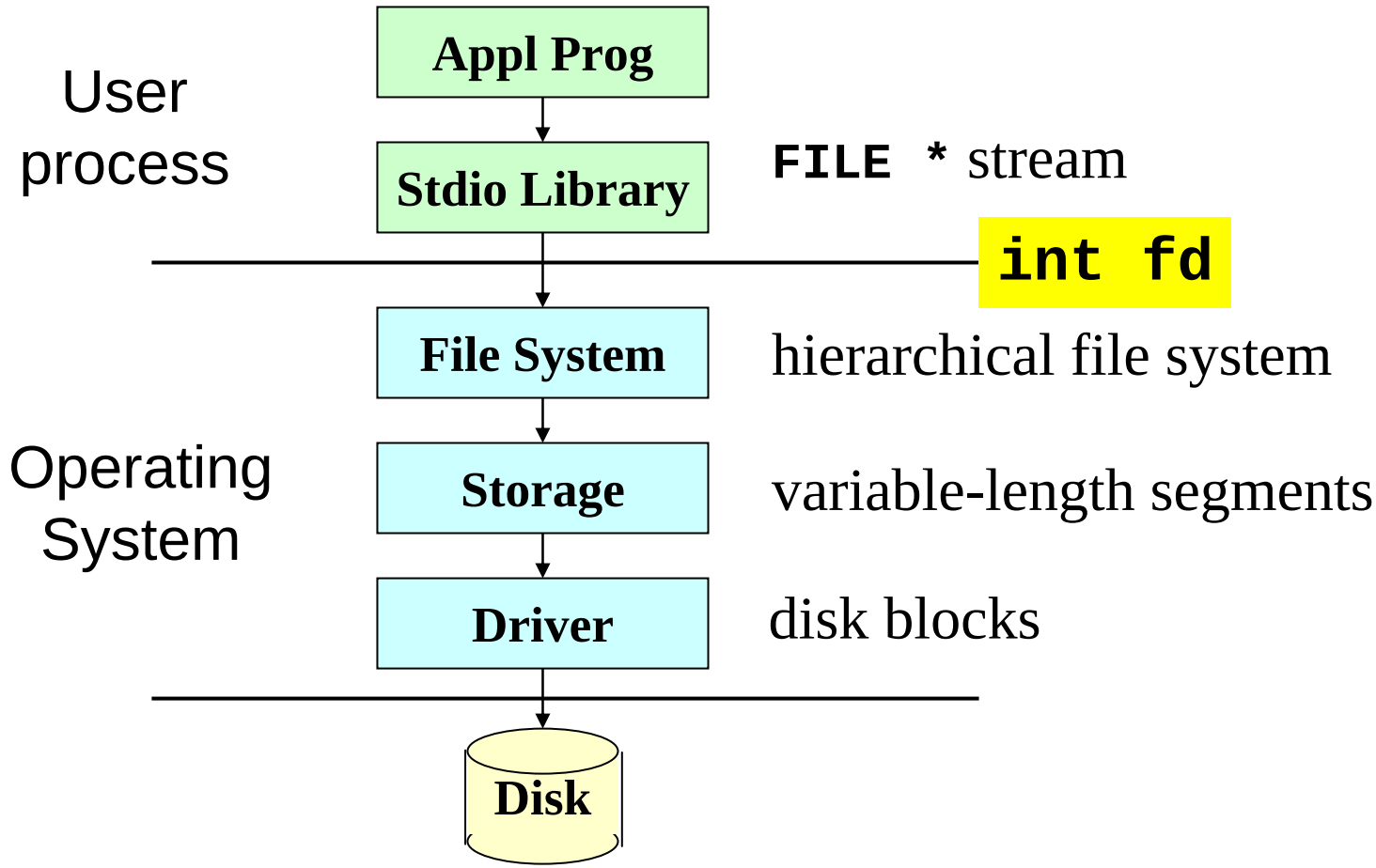
- **`FILE *fopen("myfile.txt", "r")`**
  - Open the named file and return a stream
  - Includes a mode, such as "r" for read or "w" for write
- **`int fclose(fp1)`**
  - Close the stream
  - Flush any unwritten data, and discard any unread input
- **`int fprintf(fp1, "Number: %d\n", i)`**
  - Convert and write output to stream in specified format
  - Note: **`printf(…)`** is just **`fprintf(stdout, …)`**
- **`int fscanf(fp1, "FooBar: %d", &i)`**
  - Read from stream in format and assign converted values
  - Note: **`scanf(…)`** is just **`fscanf(stdint, …)`**

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)

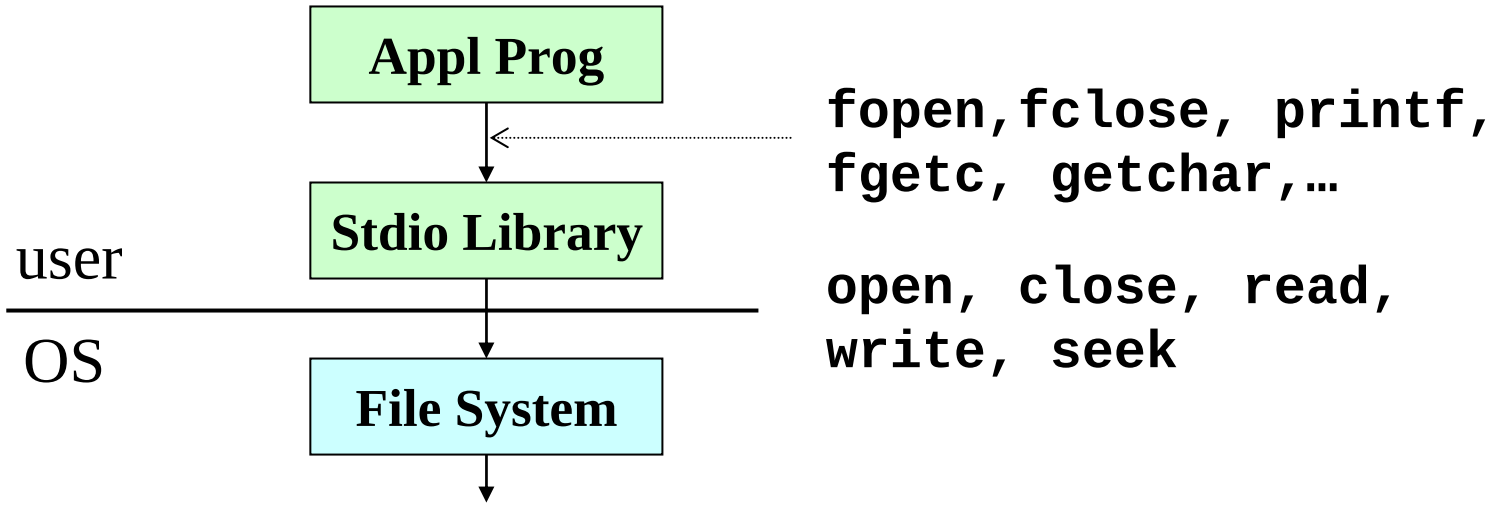
- Read/write operations advance the file position
  - Allows sequencing through the file in sequential manner
- Support for random access to the stream
  - Functions to learn current position and seek to new one

Layers of Abstraction



System Calls

- Method by which user processes invoke operating system services: “protected” function call



- Unix has ~150 system calls; see
  - man 2 intro
  - /usr/include/syscall.h

System Calls

- Processor modes
  - User mode: can execute normal instructions and access only user memory
  - Supervisor mode: can also execute privileged instructions and access all of memory (e.g., devices)
- System calls
  - User cannot execute privileged instructions
    - Users must ask OS to execute them
  - System calls are often implemented using traps
    - OS gains control through trap, switches to supervisor model, performs service, switches back to user mode, and gives control back to user

System-call Interface = ADTs

ADT

operations

- File input/output
  - open, close, read, write, lseek, dup
- Process control
  - fork, exit, wait, kill, exec, ...
- Interprocess communication
  - pipe, socket ...

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])
```

Main UNIX System Calls for Files

- Open: **`int open(char *pathname, int flags, mode_t mode);`**
  - Open a the file **`pathname`** and return a file descriptor
- Creat: **`int creat(char *pathname, mode_t mode);`**
  - Create a new file and assign a file descriptor
- Close: **`int close(int fd);`**
  - Close a file descriptor **`fd`**
- Read: **`int read(int fd, void *buf, int count);`**
  - Read up to **`count`** bytes from **`fd`**, into the buffer at **`buf`**
- Write: **`int write(int fd, void *buf, int count);`**
  - Writes up to **`count`** bytes into **`fd`**, from the buffer at **`buf`**

Example: UNIX open() System Call

- Converts a path name into a file descriptor
  - **`int open(const char *pathname, int flags, mode_t mode);`**
- Similar to **`fopen()`** in stdio
  - Uses a pathname to identify the file
  - Allows opening for reading, writing, etc
- Different from **`fopen()`** in stdio
  - Returns an integer descriptor rather than a **`FILE`** pointer
  - Specifies reading, writing, etc. through bit flags
    - E.g., **`O_RDONLY, O_WRONLY, O_RDWR`**
  - Specifies permissions to set if the file must be created
    - No need to worry about this (see K&R 8.3 for details)

Implementing `fopen()` in stdio

- If **`mode`** is invalid, return **`NULL`**
  - E.g,. mode of access needs to be ‘r’, ‘w’, or ‘a’
- Search for an available slot in the IOB array
  - Stop when unused slot is found, or return NULL if none
- Open or create 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
- Assign fields in IOB structure, and return pointer
  - Cnt of zero, base of **`NULL`**, flags based on mode, etc.

See K&R Section 8.5 for the full details

Simple Implementation of getchar()

```
int getchar(void) {
   static 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

- Problem: poor performance reading byte at a time
  - Read system call is accessing the device (e.g., a disk)
  - Reading a single byte from a disk is very time consuming
  - Insight: better to read and write in larger chunks
- Buffered I/O
  - Read a larger chunk of data from disk into a buffer
    - And dole individual bytes to user process as needed
    - Discard the 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

- Solution: read a chunk and dole out as needed

```
int getchar(void) {
    static char buf[1024];
    static char *p;
    static int n = 0;

    if (n--) return *p++;

    n = read(0, buf, sizeof(buf));
    if (n <= 0) return EOF;
    p = buf;
    return getchar();
}
```

Funny Thing About Buffered I/O

```
int main() {
   printf("Step 1\n");
   sleep(10);
   printf("Step2\n");
   return(0);
}
```

- Try running "a.out > out.txt &" and then "more 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
  - Could add a **`fflush(stdout)`** to get the output flushed

Implementing `getc()` in stdio

```
#define getc(p) \
    (--(p)->_cnt >= 0 ? \
    (int)(*(unsigned char *)(p)->_ptr++) : \
    _filbuf(p))

#define getchar() getc(stdin)
```

- Decrement the count (cnt) of remaining characters
- If any characters are left in the buffer
  - Return the character, and increment the pointer to the next character
- Else if no characters are left
  - Replenish the buffer, re-initialize the structure, and return character

So, Why is getc() a Macro?

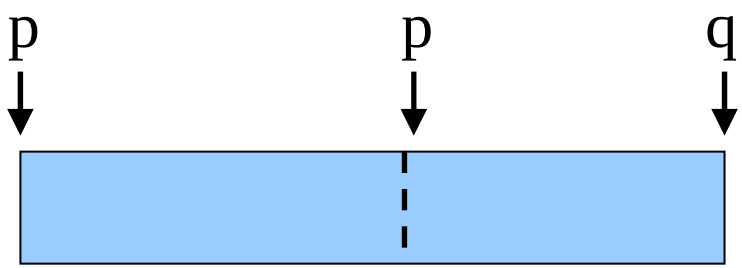
- Invented in ~1975, when
  - Computers had slow function-call instructions
  - Compilers couldn’t inline-expand very well
- It’s not 1975 any more
  - Moral: don’t invent new macros, use functions

Challenges of Writing

- Write system call
  - **`int write(int fd, void *buf, int count);`**
  - Writes *up to* ***`count`*** *bytes* into **`fd`**, from the buffer at **`buf`**
- Problem: might not write everything
  - Can return a number less than count
  - E.g., if the file system ran out of space
- Solution: safe_write
  - Try again to write the remaining bytes
  - Produce an error if it impossible to write more

Safe-Write Code

```
int safe_write(int fd, char *buf, int nbytes)
{
   int n;
   char *p = buf;
   char *q = buf + nbytes;
   while (p < q) {
      if ((n = write(fd, p, (q-p)*sizeof(char))) > 0)
         p += n/sizeof(char);
      else
         perror("safe_write:");
   }
   return nbytes;
}
```



Conclusion

- Standard I/O library provides simple abstractions
  - Stream as a source or destination of data
  - Functions for manipulating files and strings
- Standard I/O library builds on the OS services
  - Calls OS-specific system calls for low-level I/O
  - Adds features such as buffered I/O and safe writing
- Powerful examples of abstraction
  - User programs can interact with streams at a high level
  - Standard I/O library deals with some more gory details
  - Only the OS deals with the device-specific details