System Calls and Standard I/O

Professor Jennifer Rexford

http://www.cs.princeton.edu/~jrex

1

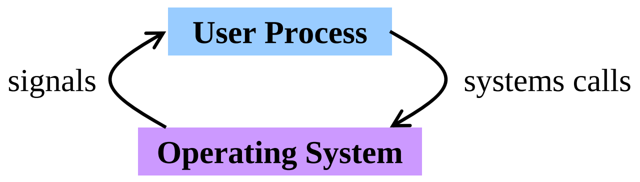
Goals of Today's Class

- System calls
  - How a user process contacts the Operating System
  - For advanced services that may require special privilege
- Standard I/O library
  - Generic I/O support for C programs
  - A smart wrapper around I/O-related system calls
  - Stream concept, line-by-line input, formatted output, ...

2

System Calls

Communicating With the OS



- System call
  - Request to the operating system to perform a task
  - … that the process does not have permission to perform
- Signal
  - Asynchronous notification sent to a process
  - … to notify the process of an event that has occurred

Processor Modes

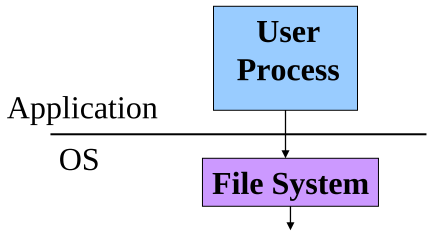
- The OS must restrict what a user process can do
  - What instructions can execute
  - What portions of the address space are accessible
- Supervisor mode (or kernel mode)
  - Can execute any instructions in the instruction set
    - Including halting the processor, changing mode bit, initiating I/O
  - Can access any memory location in the system
    - Including code and data in the OS address space
- User mode
  - Restricted capabilities
    - Cannot execute privileged instructions
    - Cannot directly reference code or data in OS address space
  - Any such attempt results in a fatal “protection fault”
    - Instead, access OS code and data indirectly via system calls

Main Categories of System Calls

- File system
  - Low-level file I/O
  - E.g., creat, open, read, write, lseek, close
- Multi-tasking mechanisms
  - Process control
  - E.g., fork, wait, exec, exit, signal, kill
- Inter-process communication
  - E.g., pipe, dup, dup2
- Unix has a few hundred system calls
  - See “man 2 intro” or /usr/include/syscall.h

System Calls

- Method for user process to invoke OS services



```
creat, open, close,
read, write, lseek
```

- Called just like a function
    - Essentially a “protected” function call
    - That transfers control to the OS and back

7

Implementing a System Call

- System calls are often implemented using traps
    - OS gains control through trap
    - Switches to supervisor mode
    - Performs the service
    - Switches back to user mode
    - Gives control back to user

Which call?
1: exit
2: fork
3: read
4: write
5: open
6: close
…

```
movl $1, %eax
int $0x80
```

Trap to the OS

System-call specific arguments are put in registers

8

Main UNIX System Calls for Files

- Creat: `int creat(char *pathname, mode_t mode);`
  - Create a new file and assign a file descriptor
- Open: `int open(char *pathname, int flags, mode_t mode);`
  - Open the file **`pathname`** and return 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`**
- Lseek: `int lseek(int fd, int offset, int whence);`
  - Assigns the file pointer to a new value by applying an **`offset`**

9

Example: UNIX open() System Call

- 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
  - Integer file descriptor (or a -1 if an error)
- Performs a variety of checks
  - E.g., whether the process is entitled to access the file

10

Example: UNIX read() System Call

- Converts a path name into 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

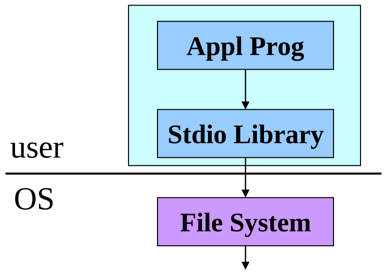
11

Standard I/O Library

12

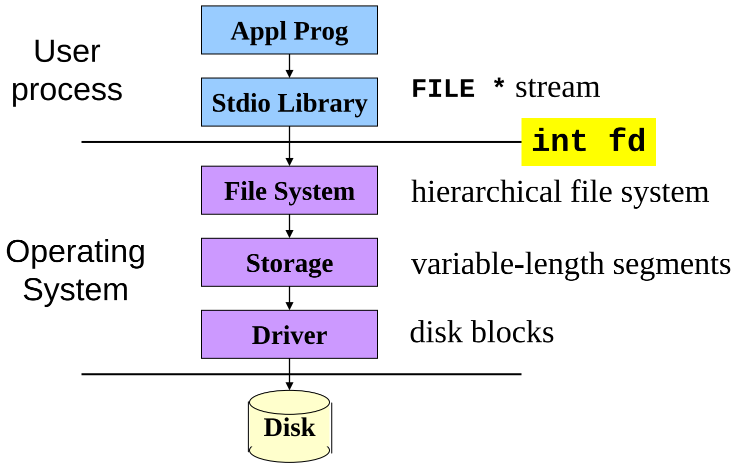
Standard I/O Library

- 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



13

Layers of Abstraction



14

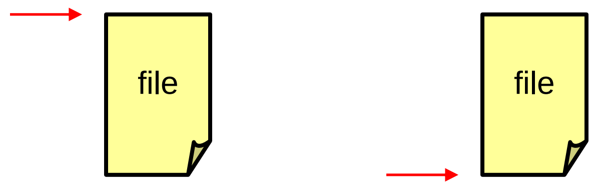
Stream Abstraction

- 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

15

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

16

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
  - File descriptor, 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

17

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, …)`**

18

Example: A Simple 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

- 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

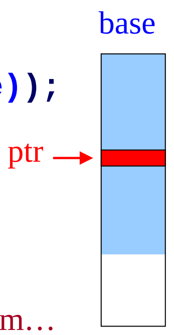
Better getchar() with Buffered I/O

```
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();
}
```

$\left.\begin{array}{l}\texttt{static char base[1024];}\\ \texttt{static char *ptr;}\\ \texttt{static int cnt = 0;}\end{array}\right\}$ 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

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

- 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

23

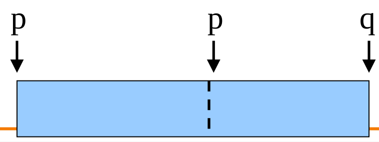
Challenges of Writing

- 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

24

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;
}
```



Summary of System Calls and Stdio

- 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