

# Process Management

1

---

---

---


---

---

---

---

---



## Goals of this Lecture

- Help you learn about:
  - Creating new processes
  - Programmatically redirecting stdin, stdout, and stderr
  - (Appendix) communication between processes via pipes
- Why?
  - Creating new processes and programmatic redirection are fundamental tasks of a Unix **shell** (see Assignment 7)
  - A power programmer knows about Unix shells, and thus about creating new processes and programmatic redirection

2

---

---

---


---

---

---

---

---



## Why Create a New Process?

- Run a new program
  - E.g., shell executing a program entered at command line
  - Or, even running an entire pipeline of commands
  - Such as `“wc -l * | sort | uniq -c | sort -nr”`
- Run a new thread of control for the same program
  - E.g., a Web server handling a new Web request
  - While continuing to allow more requests to arrive
  - Essentially time sharing the computer
- Underlying mechanism
  - A process executes `fork()` to create a child process
  - (Optionally) child process does `exec()` of a new program

3

---

---

---

---

---

---

---

---

## Creating a New Process



- Cloning an existing process
  - Parent process creates a new child process
  - The two processes then run concurrently
- Child process inherits state from parent
  - Identical (but separate) copy of virtual address space
  - Copy of the parent's open file descriptors
  - Parent and child share access to open files
- Child then runs independently
  - Executing independently, including invoking a new program
  - Reading and writing its own address space



4

---

---

---

---

---

---

---

---

## Fork System-Level Function



- `fork()` is called once
  - But returns twice, once in each process
- Telling which process is which
  - Parent: `fork()` returns the child's process ID
  - Child: `fork()` returns 0

```
pid = fork();
if (pid != 0) {
    /* in parent */
    ...
} else {
    /* in child */
    ...
}
```

5

---

---

---

---

---

---

---

---

## Fork and Process State



- |  |   |
|--|---|
| <ul style="list-style-type: none"><li>• Inherited<ul style="list-style-type: none"><li>• User and group IDs</li><li>• Signal handling settings</li><li>• Stdio</li><li>• File pointers</li><li>• Root directory</li><li>• File mode creation mask</li><li>• Resource limits</li><li>• Controlling terminal</li><li>• All machine register states</li><li>• Control register(s)</li><li>• ...</li></ul></li></ul> | <ul style="list-style-type: none"><li>• Separate in child<ul style="list-style-type: none"><li>• Process ID</li><li>• Address space (memory)</li><li>• File descriptors</li><li>• Parent process ID</li><li>• Pending signals</li><li>• Time signal reset times</li><li>• ...</li></ul></li></ul> |
|--|---|

6

---

---

---

---

---

---

---

---

## Example: What Output?



```
int main(void)
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid != 0) {
        printf("parent: x = %d\n", --x);
        exit(0);
    } else {
        printf("child: x = %d\n", ++x);
        exit(0);
    }
}
```

7

---

---

---

---

---

---

---

---

## Executing a New Program



- `fork()` copies the state of the parent process
  - Child continues running the parent program
  - ... with a copy of the process memory and registers
- Need a way to invoke a new program
  - In the context of the newly-created child process
- Example

program → `execvp("ls", argv);`  
NULL-terminated array  
Contains command-line arguments  
(to become "argv[]" of ls)

```
execvp("ls", argv);
fprintf(stderr, "exec failed\n");
exit(EXIT_FAILURE);
```

8

---

---

---

---

---

---

---

---

## Waiting for the Child to Finish



- Parent should wait for children to finish
  - Example: a shell waiting for operations to complete
- Waiting for a child to terminate: `wait()`
  - Blocks until some child terminates
  - Returns the process ID of the child process
  - Or returns -1 if no children exist (i.e., already exited)
- Waiting for specific child to terminate: `waitpid()`
  - Blocks till a child with particular process ID terminates

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

9

---

---

---

---

---

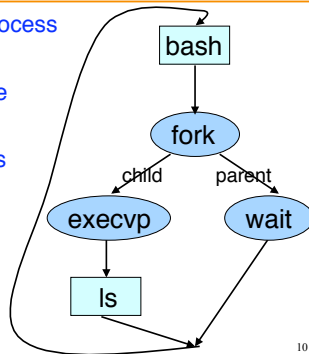
---

---

---

## Example: A Simple Shell

- Shell is the parent process
  - E.g., bash
- Parses command line
  - E.g., "ls -l"
- Invokes child process
  - `fork()`, `execvp()`
- Waits for child
  - `wait()`



---

---

---

---

---

---

---

---

## Simple Shell Code

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

---

---

---

---

---

---

---

---

## Simple Shell Trace (1)

### Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

Parent reads and parses command line  
Parent assigns values to `somepgm` and `someargv`

---

---

---

---

---

---

---

---

## Simple Shell Trace (2)



### Parent Process

```
Parse command line
Assign values to someppm, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execvp(someppm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

executing  
concurrently

### Child Process

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execvp(someppm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

`fork()` creates child process  
Which process gets the CPU first? Let's assume the parent...

13

---

---

---

---

---

---

---

---

## Simple Shell Trace (3)



### Parent Process

```
Parse command line
Assign values to someppm, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execvp(someppm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

child's pid

executing  
concurrently

### Child Process

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execvp(someppm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

In parent, `pid != 0`; parent waits; OS gives CPU to child

14

---

---

---

---

---

---

---

---

## Simple Shell Trace (4)



### Parent Process

```
Parse command line
Assign values to someppm, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execvp(someppm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

executing  
concurrently

### Child Process

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execvp(someppm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

In child, `pid == 0`; child calls `execvp()`

15

---

---

---

---

---

---

---

---

## Simple Shell Trace (5)



Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execlp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

executing  
concurrently

Child Process

```
somepgm
```

In child, somepgm overwrites shell program; `main ()` is called with `someargv` as argv parameter

16

---

---

---

---

---

---

---

---

## Simple Shell Trace (6)



Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execlp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

executing  
concurrently

Child Process

```
somepgm
```

Somepgm executes in child, and eventually exits

17

---

---

---

---

---

---

---

---

## Simple Shell Trace (7)



Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execlp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

Parent returns from `wait ()` and proceeds

18

---

---

---

---

---

---

---

---

## Combined Fork/Exec/Wait



- Common combination of operations
  - `fork()` to create a new child process
  - `exec()` to invoke new program in child process
  - `wait()` in the parent process for the child to complete
- Single call that combines all three
  - `int system(const char *cmd);`
- Example

```
int main(void) {
    system("echo Hello world");
    return 0;
}
```

19

---

---

---

---

---

---

---

---

## Fork and Virtual Memory



- Incidentally...
- Question:
  - `fork()` duplicates an entire process (text, bss, data, rodata, stack, heap sections)
  - Isn't that *very* inefficient???!?
- Answer:
  - Using virtual memory, not really!
  - Upon `fork()`, OS creates virtual pages for child process
  - Each child virtual page points to real page (in memory or on disk) of parent
  - OS duplicates real pages incrementally, and only if/when "write" occurs

20

---

---

---

---

---

---

---

---

## Redirection



- Unix allows programmatic redirection of stdin, stdout, or stderr
- How?
  - Use `open()`, `creat()`, and `close()` system calls
    - Described in **I/O Management** lecture
  - Use `dup()` system call...

```
int dup(int oldfd);
```

- Create a copy of the file descriptor `oldfd`. After a successful return from `dup()` or `dup2()`, the old and new file descriptors may be used interchangeably. They refer to the same open file description and thus share file offset and file status flags. Uses the lowest-numbered unused descriptor for the new descriptor. Return the new descriptor, or -1 if an error occurred.

21

---

---

---

---

---

---

---

---

## Redirection Example



How does shell implement "somepgm > somefile"?

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

---

---

---

---

---

---

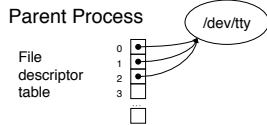
---

---

---

---

## Redirection Example Trace (1)



```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Parent has file descriptor table; first three point to "terminal"

---

---

---

---

---

---

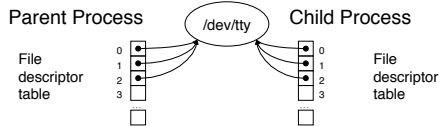
---

---

---

---

## Redirection Example Trace (2)



```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Parent forks child; child has identical file descriptor table

---

---

---

---

---

---

---

---

---

---



### Redirection Example Trace (3)

**Parent Process**

File descriptor table

0	•
1	•
2	•
3	□

/dev/tty

**Child Process**

File descriptor table

0	•
1	•
2	•
3	□

```

pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
  
```

```

pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
  
```

Let's say parent gets CPU first; parent waits

---

---

---

---

---

---

---

---

---

---

### Redirection Example Trace (4)

**Parent Process**

File descriptor table

0	•
1	•
2	•
3	□

/dev/tty

somefile

**Child Process**

File descriptor table

0	•
1	•
2	•
3	□

```

pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
  
```

```

pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
  
```

Child gets CPU; child creates somefile

---

---

---

---

---

---

---

---

---

---

### Redirection Example Trace (5)

**Parent Process**

File descriptor table

0	•
1	•
2	•
3	□

/dev/tty

somefile

**Child Process**

File descriptor table

0	•
1	•
2	•
3	□

```

pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
  
```

```

pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
  
```

Child closes file descriptor 1 (stdout)

---

---

---

---

---

---

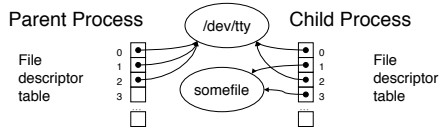
---

---

---

---

### Redirection Example Trace (6)



```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child duplicates file descriptor 3 into first unused spot

28

---

---

---

---

---

---

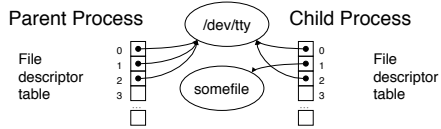
---

---

---

---

### Redirection Example Trace (7)



```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child closes file descriptor 3

29

---

---

---

---

---

---

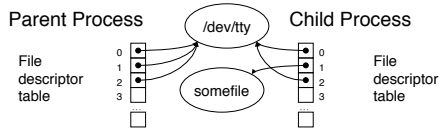
---

---

---

---

### Redirection Example Trace (8)



```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child calls execvp()

30

---

---

---

---

---

---

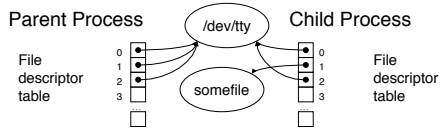
---

---

---

---

## Redirection Example Trace (9)



```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

```
somepgm
```

Somepgm executes with stdout redirected to somefile

31

---

---

---

---

---

---

---

---

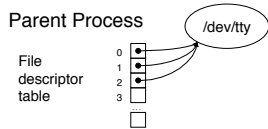
---

---

---

---

## Redirection Example Trace (10)



```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somefile, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Somepgm exits; parent returns from wait() and proceeds

32

---

---

---

---

---

---

---

---

---

---

---

---

## The Beginnings of a Unix Shell



- A shell is mostly a big loop
  - Parse command line from stdin
  - Expand wildcards (\*)
  - Interpret redirections (<, and >)
  - fork(), dup(), exec(), and wait(), as necessary
- Start from the code in earlier slides
  - And edit till it becomes a Unix shell
  - This is the heart of the last programming assignment

33

---

---

---

---

---

---

---

---

---

---

---

---

## Summary



- **System-level functions for creating processes**
  - `fork()` : process creates a new child process
  - `wait()` : parent waits for child process to complete
  - `exec()` : child starts running a new program
  - `system()` : combines fork, wait, and exec all in one
- **System-level functions for redirection**
  - `open()` / `creat()` : to open a file descriptor
  - `close()` : to close a file descriptor
  - `dup()` : to duplicate a file descriptor

34

---

---

---

---

---

---

---

---

## Appendix



### Inter-Process Communication (IPC)

35

---

---

---

---

---

---

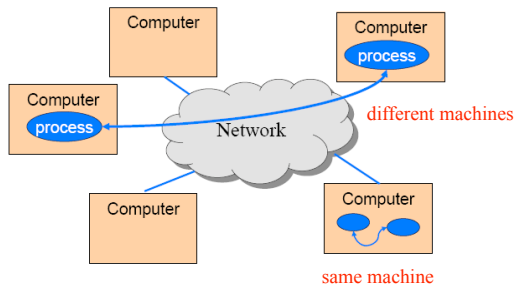
---

---

## IPC



- Mechanism by which two processes exchange information and coordinate activities



36

---

---

---

---

---

---

---

---

## IPC Mechanisms



- Pipes
  - Processes on the same machine
  - Allows parent process to communicate with child process
  - Allows two “sibling” processes to communicate
  - Used mostly for a pipeline of filters
- Sockets
  - Processes on any machines
  - Processes created independently
  - Used for client/server communication (e.g., Web)

Both provide abstraction of an “ordered stream of bytes”

37

---

---

---

---

---

---

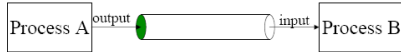
---

---

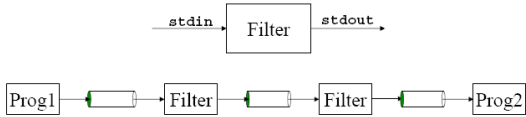
## Pipes



- Provides an interprocess communication channel



- A **filter** is a process that reads from `stdin` and writes to `stdout`



38

---

---

---

---

---

---

---

---

## Example Use of Pipes



- Compute a histogram of content types in my e-mail
  - Many e-mail messages, consisting of many lines
  - Lines like “Content-Type: image/jpeg” indicate the type
- Pipeline of Unix commands
  - Identifying content type: `grep -i Content-Type *`
  - Extracting just the type: `cut -d" " -f2`
  - Sorting the list of types: `sort`
  - Counting the unique types: `uniq -c`
  - Sorting the counts: `sort -nr`
- Simply running this at the shell prompt:
  - `grep -i Content-Type * | cut -d" " -f2 | sort | uniq -c | sort -nr`

39

---

---

---

---

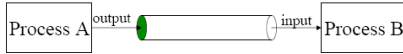
---

---

---

---

## Creating a Pipe



- Pipe is a communication channel abstraction
  - Process A can write to one end using "write" system call
  - Process B can read from the other end using "read" system call
- System call

```
int pipe( int fd[2] );
return 0 upon success -1 upon failure
fd[0] is open for reading
fd[1] is open for writing
```
- Two coordinated processes created by `fork` can pass data to each other using a pipe.

40

---

---

---

---

---

---

---

---

## Pipe Example



```
int pid, p[2];
...
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    ... read using p[0] as fd until EOF ...
}
else {
    close(p[0]);
    ... write using p[1] as fd ...
    close(p[1]); /* sends EOF to reader */
    wait(&status);
}
```



41

---

---

---

---

---

---

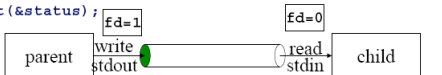
---

---

## Pipes and Stdio



```
int pid, p[2];
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    dup2(p[0], 0);
    close(p[0]);
    ... read from stdin ...
}
else {
    close(p[0]);
    dup2(p[1], 1);
    close(p[1]);
    ... write to stdout ...
    wait(&status);
}
```



42

---

---

---

---

---

---

---

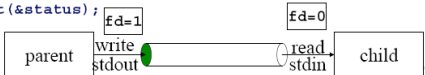
---

## Pipes and Exec



```
int pid, p[2];
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    dup2(p[0], 0);
    close(p[0]);
    execl(...);
}
else {
    close(p[0]);
    dup2(p[1], 1);
    close(p[1]);
    ... write to stdout ...
    wait(&status);
}
```

child process  
invokes a new program



43

---

---

---

---

---

---

---

---

## I/O Management



44

---

---

---

---

---

---

---

---

## 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

45

---

---

---

---

---

---

---

---

## 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

46

---

---

---

---

---

---

---

---

## 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

47

---

---

---

---

---

---

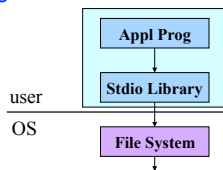
---

---

## 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



48

---

---

---

---

---

---

---

---



## 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

49

---

---

---

---

---

---

---

---

## 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, ...)`

50

---

---

---

---

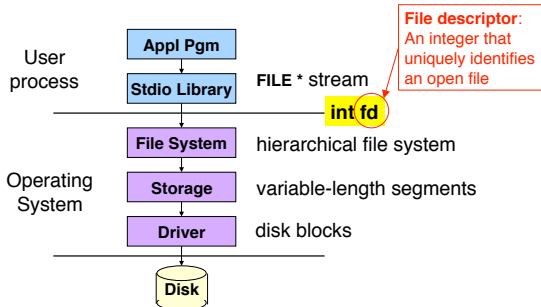
---

---

---

---

## Layers of Abstraction



51

---

---

---

---

---

---

---

---

## 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
```

52

---

---

---

---

---

---

---

---

## 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()`

53

---

---

---

---

---

---

---

---

## 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.

54

---

---

---

---

---

---

---

---

## 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

55

---

---

---

---

---

---

---

---

## 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

56

---

---

---

---

---

---

---

---

## Better getchar () with Buffered I/O



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

if (cnt-- > 0) 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...

57

---

---

---

---

---

---

---

---

## 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])
```

58

---

---

---

---

---

---

---

---

## 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

59

---

---

---

---

---

---

---

---

## 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

60

---

---

---

---

---

---

---

---