Announcements

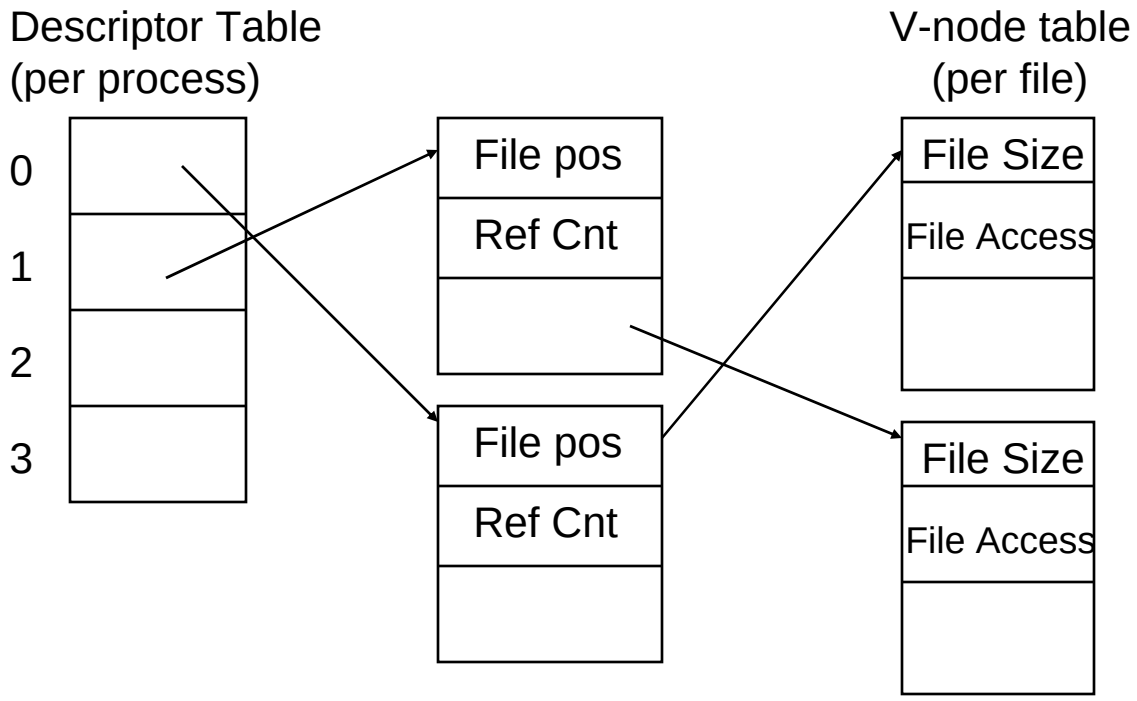
- Program #4 is on the web
  - Change is submit setup
    - 2 tokens/24 hours
    - Release tests will show output
- Midterm #2
  - April 10 (6:00-7:30 in ARM 0126)
  - No class on April 10th
- Reading
  - Bryant & O'Hallaron 8.5, 11.{6,7,8} (Today)
  - Bryant & O'Hallaron Chapter 12 (Thursday)

File Descriptor Vs. FILE*

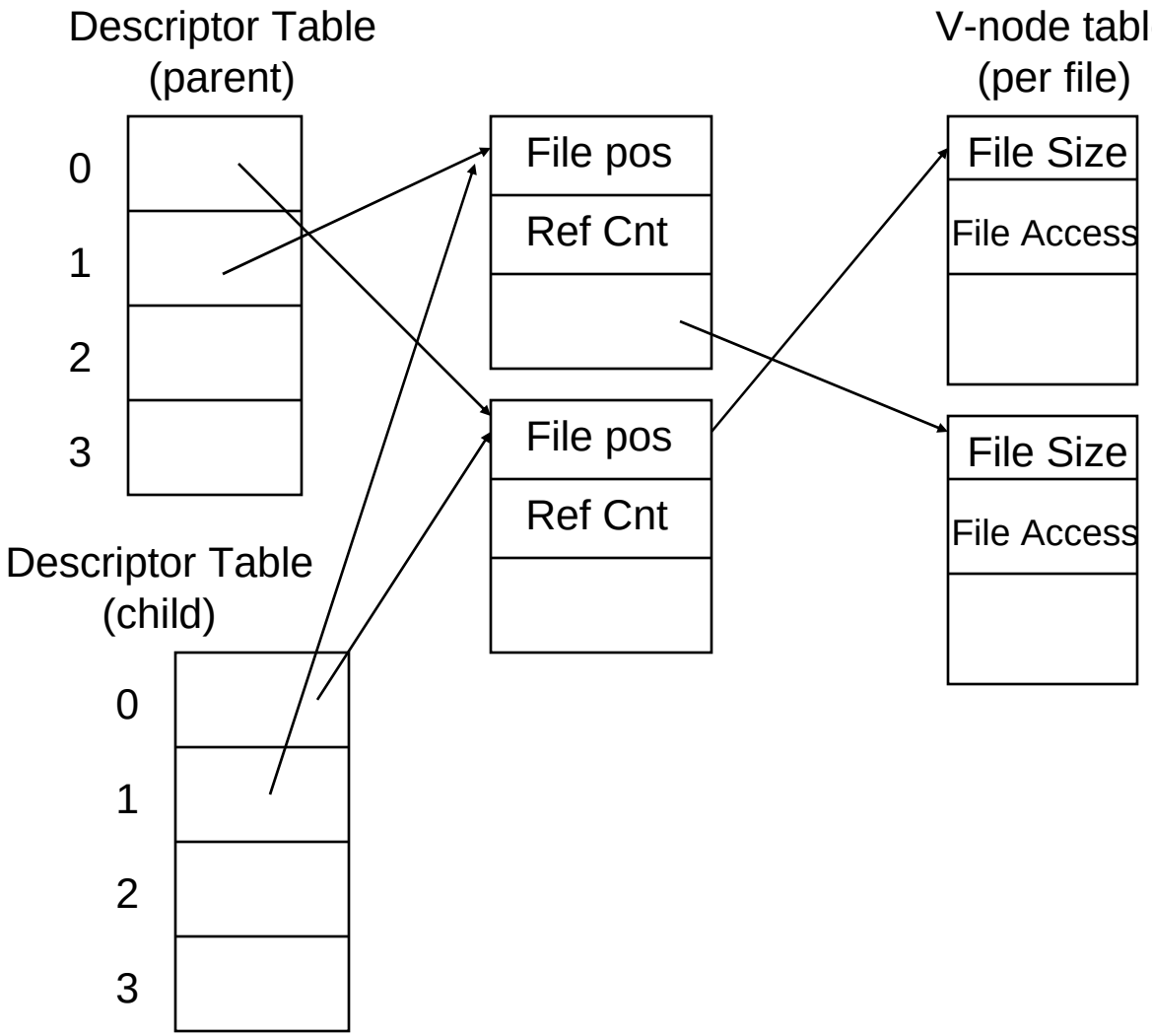
- File Descriptors
  - Permit unbuffered I/O
  - Are integers
  - Used or returned by open, read, write
- FILE*
  - Are used for buffered I/O
  - Are pointers to a struct
    - Struct contains a file descriptor and buffers
  - Are used or returned by fopen, fwrite, fprintf, ….

File Descriptors

- All UNIX I/O (at the OS) is done via file descriptors
  - They are integer numbers
  - Represent an abstract data type used by the operating system
  - open system call returns a new file descriptor
  - Read and write system call use file descriptors

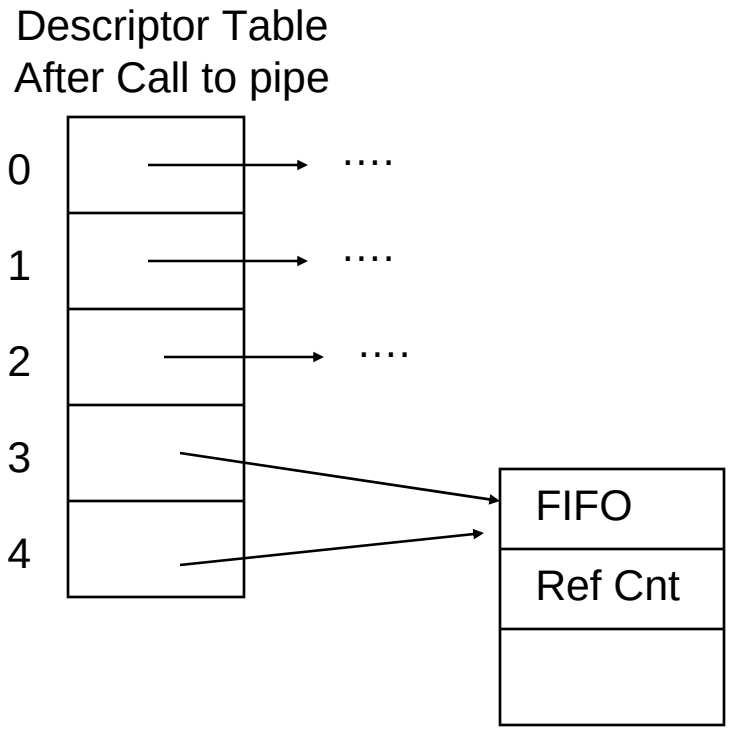


File Table After Forking a Process



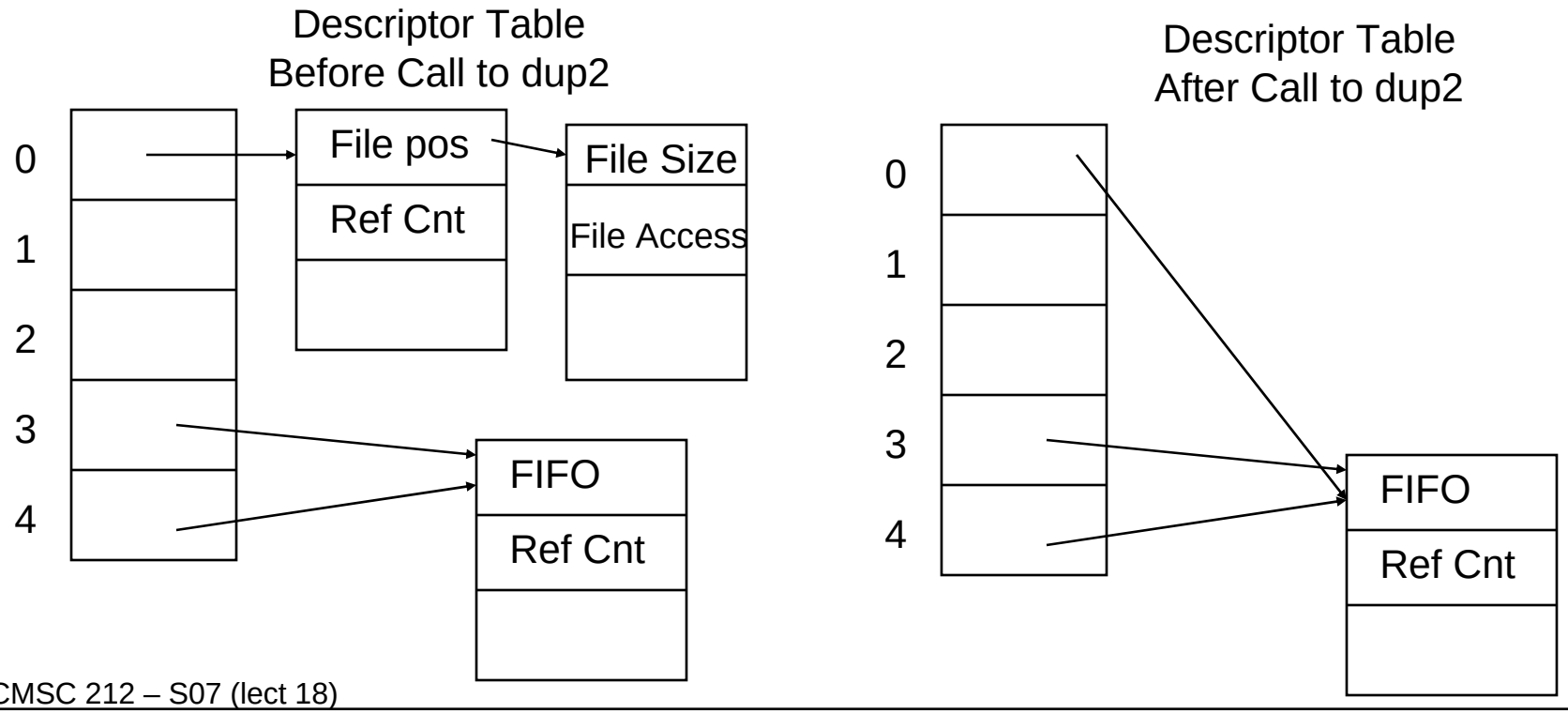
Pipe System Call

- int pipe(int fd[2]);
  - Creates a communication channel between two file descriptors
  - Data written to fd[1] can be read from fd[0]
  - Data written to fd[0] can be read from fd[1]



dup2 System Call

- int dup2(int fds1, int fds2);
  - Closes file descriptor fds1
  - Copies fds2 to fds2
- Example
  - dup2(0, 4);



Changing Standard Input/Output

- Useful to combine pipe, dup2, and exec

```
....
pipe(pipefds);
pid = fork();
if (pid == 0) {
      dup2(0, pipefd[0]);
      close(pipefd[1]);
      ret = execvp("ls", args);
      ....
else if (pid > 0) {
      dup2(1, pipefd[1]);
      close(pipefd[0]);
} else {
   ....
}
```

A Simple Shell Program

```
while (1)  {
        ret = fgets(line, sizeof(line), stdin);
        …
        /* convert line into array of arguments */
        pid = fork();
        if (pid == 0) {
                ret = execvp(args[0], args);
            if (ret) {
                printf("Command not found\n");
                exit(-1);
            }
        } else if (pid > 0) {
            ret = waitpid(pid, &status, NULL);
        } else { … }
}
```

Adding Pipes to the shell

- Declare a struct to hold each command

```
struct command {
        int pid;
        char *args[MAX];
};
#define MAX_PROGS 50
struct command cmd[MAX_PROGS];
```

- Read line into array cmd
  - a | b –foo | c –help | d stuff here
  - This line becomes:

```
cmd[0].args = {  “a” };
cmd[1].args = { “b”, “-foo” };
cmd[2].args = { “c”, “-help” };
cmd[3].args = { “d”, “stuff”, “here” };
```

Adding a Pipe Command to the Shell

```
/* skipped: split line into an array of commands to run */
int infds[2], outfds[2];
for (i=0; i < num; i++) {
        if (i < num -1) pipe(outfds);
        cmd[i].pid = fork();
        if (pid == 0) {
            if (i > 0) dup2(0, infds[0]);
            if (i < num-1) dup2(1, outfds[1]);
            close(infds[1]);
            close(outfds[0]);
            ret = execvp(cmd[i].args[0], cmd[i].args);
            if (ret) {
                printf("Command not found\n");
                exit(-1);
            }
               } else if (pid > 0) {
             memcpy(infds, outfds, sizeof(int)*2));
        }
}
/* wait for all forked commands to exit */
```

Signals

- A way for processes to communicate exceptional events
- Can originate from:
  - Hardware events:
    - SIGFPE – floating point exception
    - SIGILL – illegal instruction
    - SIGSEGV – segmentation violation
  - User interaction:
    - SIGKILL – kill the program
    - SIGINT – suspend process (control-z)
    - SIGSCHILD – child process stopped or terminated

Process Groups

- Every process is the member of one group
  - A small integer of type pid_t
- A child process inherits its parent's group
- A process can change its group
  - Setpgid(pid_t pid, pitd_t pgid)
    - Setpgid(0,0) – create a new group for the current proccess

Sending Signals

- Kill command sends a signal
  - From C: int kill(pid_t pid, int sig);
  - From command line: kill –signal <pid>
- A negative pid
  - Sends a signal to all processes in group abs(pid)
- When a process receives a signal
  - Might terminate
  - Might ignore signal
  - Might run a function