CMSC 216
Introduction to Computer Systems
Lecture 21
Process Control, cont.
& System-Level I/O

Larry Herman & Alan Sussman

Notes

- Project 6 posted, public tests on Grace soon
  - questions?
- Exam #2 Thursday
  - Make (lecture 12) through System-level I/O (today)
  - Practice exam posted
- Read Ch. 10 on System-level I/O

Sections 8.2-8.5, Bryant and O'Hallaron

PROCESS CONTROL

`execve()` example

```
/* #include statements omitted */
extern char **environ;
int main() {
  char *args[] = { "ls", "-l", NULL };
  pid_t child_pid;
  if ((child_pid = fork()) < 0)
    err(EX_OSERR, "fork error");
  if (child_pid) {  /* parent code */
    wait(NULL);
    printf("Parent pid = %d; my child had pid = %d\n",
           getpid(), child_pid);
  }
  else {  /* child code */
    printf("PID %d replacing myself\n", getpid());
    execve("/bin/ls", args, environ);
    err(EX_OSERR, "exec error");  /* why no if statement? */
  }
  return 0;
}
```

Other exec functions

- There are 5 other functions that perform exec operations
  - **l**: use an argument list, terminated by **NULL**
  - **v**: use an argument vector
  - **p**: search the **PATH** list of directories
  - **e**: can specify environment
- `execle(const char *filename, ..., NULL, char * const envp[]);`
- `execv(const char *filename, char * const argv[]);`
- `execl(const char *filename, ..., NULL);`
- `execvp(const char *progname, char * const argv[]);`
- `execlp(const char *progname, ..., NULL);`

How to use the other functions

```
char *filename = "/bin/ls";
char *args[] = { "ls", "-l", NULL };

execle("/bin/ls", "ls", "-l", NULL,
       environ);
execv(filename, args);
execl("/bin/ls", "ls", "-l", NULL);
execvp(args[0], args);
execlp("ls", "ls", "-l", NULL);
```


Why you specify the program twice

- The first time, you tell the system which program to run
- The second time, it's **argv[0]** for that program
  - used to by the program to know what its name is
  - sometimes the same program runs differently under different names
- Can be used for bizarre means...

An admittedly contrived example

```
$ cat sleeper.c
#include <stdio.h>
#include <unistd.h>
int main() {
  printf("PID = %d\n", getpid());
  execlp("sleep", "blargh", "60", NULL);
  return 1;
}
$ ./sleeper &
PID = 674
$ ps -fp 674
UID        PID  PPID  C STIME TTY          TIME CMD
bofh       674 26789  0 22:12 pts/6    00:00:00 blargh 60
```

A simple shell

- With fork, exec, and waiting, we have all the tools we need to build our own shell
- Basic idea: infinite loop with prompt
- Inside loop:
  - read in a command line
  - parse the command line
  - fork; parent waits, child execs command

A simple shell program

```
while (1) {
  /* prompt and read into buffer */
  /* parse buffer into string vector argv */
  switch (fork()) {
    case -1:
      perror("fork error");
      break;
    case 0:
      execvp(argv[0], argv);
      err(EX_OSERR, "exec error");
      break;
    default:
      wait(NULL);
      break;
  }
}
```


Chapter 10, Bryant and O'Hallaron

SYSTEM-LEVEL I/O

UNIX file management

- A file in UNIX is just a sequence of bytes
- All I/O devices are modeled as files in UNIX
  - disks, network sockets, etc.
- The kernel maintains a file descriptor table, holding information about all open files, for each process
- File descriptors are nonnegative integers used by processes to access files in a process' table

File operations

- A process requests access to a file via an **`open()`** system call; the kernel returns a file descriptor
- The process reads, writes, and moves around the file using the file descriptor and other system calls (**`read()`**, **`write()`**, **`lseek()`**)
- When finished with a file, the process closes the file via the **`close()`** system call
  - All open files are closed by the kernel when a process terminates

File descriptors

- In each process, there is a file descriptor associated with each open file
- Multiple processes can have the same file open; closing a file in one process will not close the file in other processes
- Standard input, output and error have file descriptors 0, 1, and 2, respectively
- Don't use those numbers in your programs: **`STDIN_FILENO`**, **`STDOUT_FILENO`**, and **`STDERR_FILENO`** are provided for you

The `open()` system call

- Two forms:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *filename, int flags);
int open(const char *filename, int flags, mode_t mode);
```

  - **`flags`** is the result of a bitwise OR of any of several options:
    - **`O_RDONLY`**: read only
    - **`O_WRONLY`**: write only
    - **`O_RDWR`**: read and write
    - **`O_CREAT`**: create file if it doesn't exist
    - **`O_EXCL`**: cause an error if file already exists
    - **`O_TRUNC`**: if file exists, truncate it to an empty file
    - **`O_APPEND`**: cause each write operation to write to end of file
  - **`mode`** specifies the permissions to be used if/when creating a new file
    - only needed/checked if **`O_CREAT`** is specified
    - we'll use **`0666`**; this allows all users to read and write the file
    - mode is made less permissive by the process' **`umask`**
  - returns -1 on error, or a file descriptor for the opened file

The `close()` system call

- When finished with a file, your programs should then release all memory associated with the use of that file using the **`close()`** system call

```
#include <unistd.h>
int close(int fd);
```

  - returns 0 if successful, -1 on error
  - errors can occur if you try to free an already freed descriptor, or if something else interrupts the closing operation

A simple example using `open()`

open.c:

```
/* #include statements omitted */

/* same as 0666, but a bit more symbolic */
#define DEF_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)

int main() {
  int fd;

  if ((fd = open("missing-file.txt", O_RDONLY)) < 0)
    perror("can't open missing-file.txt for read");
  else
    close(fd);

  fd = open("file.txt", O_WRONLY | O_TRUNC | O_CREAT, DEF_MODE);
  if (fd < 0)
    perror("can't open file.txt");
  else
    close(fd);

  fd = open("new-file.txt", O_WRONLY | O_APPEND | O_CREAT, DEF_MODE);
  if (fd < 0)
    perror("can't open new-file.txt for write");
  else
    close(fd);

  return 0;
}
```


Running the simple example

```
$ ls -l
total 8
-rw-rw----+ 1 bofh bofh   33 Apr 29 20:53 file.txt
-rwxrwx---  1 bofh bofh 5195 Apr 29 20:52 open
-rw-rw----+ 1 bofh bofh  696 Apr 29 20:52 open.c
$ cat file.txt
My, this is an interesting file.
$ ./open
can't open missing-file.txt for read: No such file or directory
$ ls -l
total 9
-rw-rw----+ 1 bofh bofh    0 Apr 29 20:55 file.txt
-rw-rw----+ 1 bofh bofh    0 Apr 29 20:55 new-file.txt
-rwxrwx---  1 bofh bofh 5195 Apr 29 20:52 open
-rw-rw----+ 1 bofh bofh  696 Apr 29 20:52 open.c
```


Performing I/O

- Typically, a program will use the **`read()`** and **`write()`** system calls to perform I/O operations on a file descriptor

```
#include <unistd.h>
ssize_t read(int fd,
             void *buffer, size_t n);
```

  - reads up to **`n`** bytes from **`fd`** into **`buffer`**
  - returns -1 on error, 0 on EOF, or number of bytes read (may be < **`n`**)

```
ssize_t write(int fd,
              const void *buffer, size_t n);
```

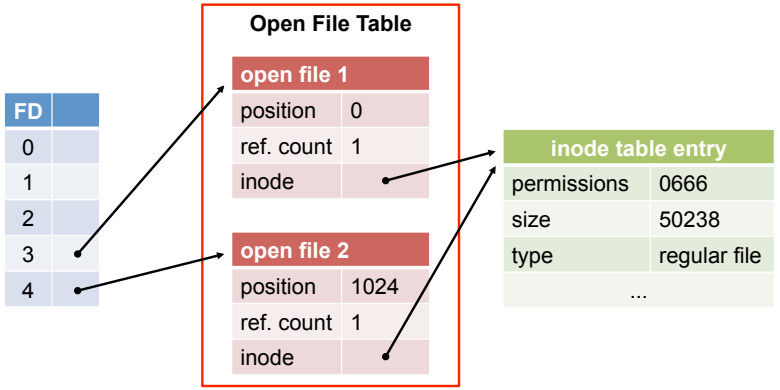
  - writes up to **`n`** bytes from **`buffer`** to **`fd`**
  - returns -1 on error, or number of bytes written (may be < **`n`**)

Kernel file data structures

- Each process has its own file descriptor table, mapping integers to open files
- The kernel keeps an open file table to keep track of all open files in the system
  - shared by all processes
  - the same file can be opened multiple times
  - contains current file position, reference count (how many descriptors point to the entry), and inode pointer (among other things)
  - entries are removed from here once the reference count is 0
- The kernel also has an inode table; each inode contains information about a file
  - e.g., mode (permissions + type of file), size
  - also shared by all processes

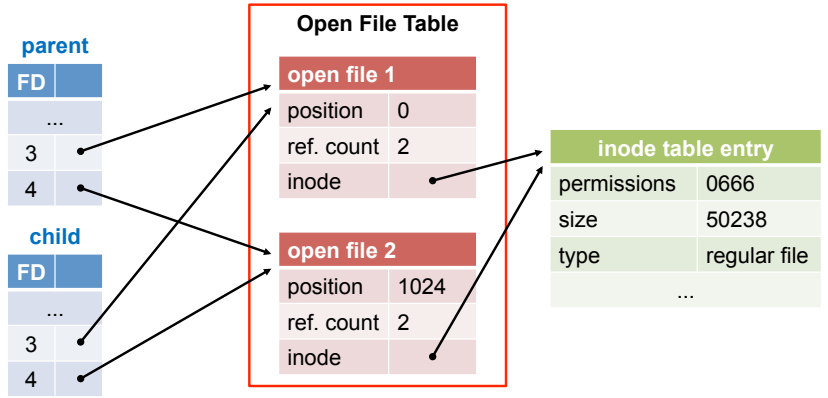
Opening the same file twice

```
fd1 = open("file.txt", O_RDONLY);
fd2 = open("file.txt", O_RDONLY);
read(fd2, buffer, 1024);
```



After a fork

```
fd1 = open("file.txt", O_RDONLY);
fd2 = open("file.txt", O_RDONLY);
read(fd2, buffer, 1024);
fork();
```



I/O redirection

- The **`dup2()`** system call copies an open file table entry from one file descriptor to another

  `#include <unistd.h>`

  `int dup2(int old_fd, int new_fd);`

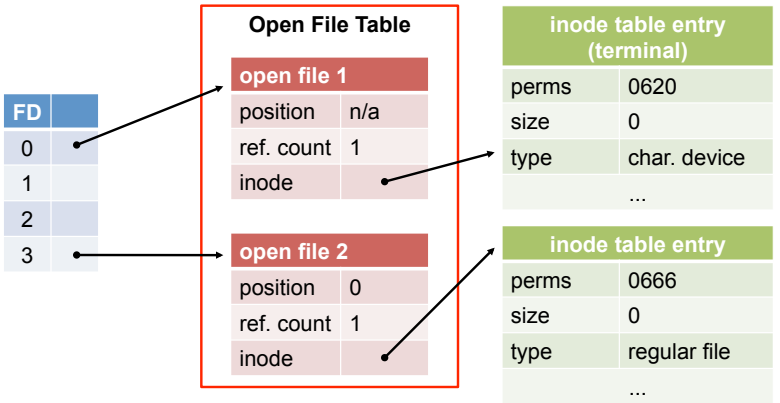
  – returns -1 on error, **`new_fd`** on success

  – **`new_fd`** is closed first if necessary (if it's already open)

- This can be used for things like I/O redirection, or to allow reading from a file named on the command line (instead of stdin)

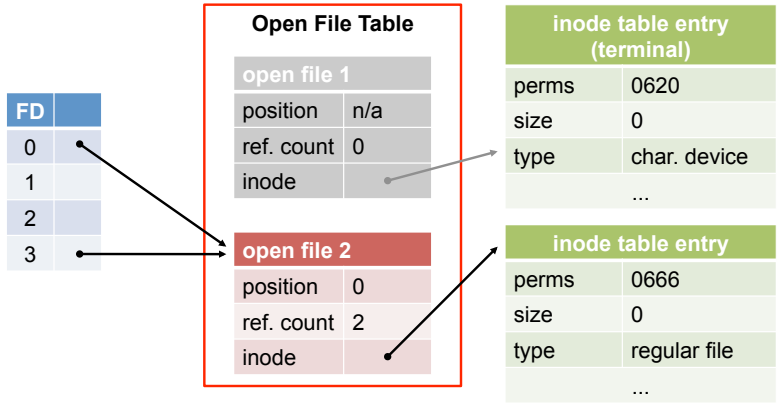
Before `dup2()`

```
int fd = open(filename, O_RDONLY);
```



After `dup2()`

```
int fd = open(filename, O_RDONLY);
dup2(fd, STDIN_FILENO);
```



Example using `dup2()`

- Allow reading from stdin or named file:

```
int main(int argc, char **argv) {
  ...
  if (argc > 1) {
    int fd = open(argv[1], O_RDONLY);
    dup2(fd, STDIN_FILENO);
    close(fd);
  }
  /* read from stdin from here on */
  ...
}
```


Interprocess Communication

- Processes can communicate with each other via various means, including signals and pipes
- Signals have a limited "vocabulary" - only 64 signals on some machines; other limitations as well
- Pipes allow arbitrary strings to be written from one process to another
- Two kinds of pipes: FIFOs (or "named pipes", created by **`mkfifo()`**), and anonymous pipes (often just called "pipes", created by **`pipe()`**)

Pipes

- Note: much of this information is from the pipe man page, in section 7 ("**`man 7 pipe`**")
  - the **`pipe()`** system call discussed below is in section 2
- Provide a unidirectional IPC channel, with a read end and a write end
- Created with the **`pipe()`** system call:

```
#include <unistd.h>
int pipe(int fd[2]);
```

  - **`fd`** needs to be an array of 2 elements
  - return value is 0 on success, -1 on error
  - On success, a new pipe is created, **`fd[0]`** will be the file descriptor for the read end, and **`fd[1]`** will be the file descriptor for the write end

Pipes, cont.

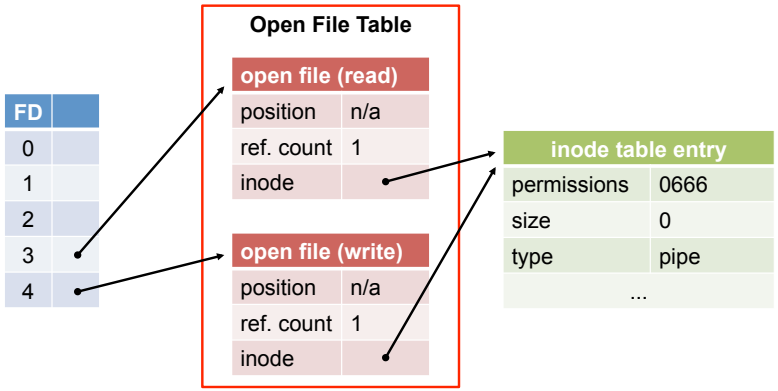
- An anonymous pipe's file descriptors are only visible in the process that created the pipe (and its children)
- I/O is blocking – read() from an empty pipe does not return until data is in the pipe, and write() to a full pipe does not return until sufficient space exists to complete the write
- Reads on the pipe will return EOF if no process has the write end open; writes to a pipe that has no readers cause an error
- Can't move the file position around (no seeking)

Pipes, cont.

- Generally, we create a pipe, then fork()
  - parent and child can then communicate via the pipe
- Because of the blocking I/O, we need to close ends of the pipe that we're not using, both immediately after the fork and once we're finished reading/writing
  - otherwise, read won't signal EOF appropriately or cause errors when we want to

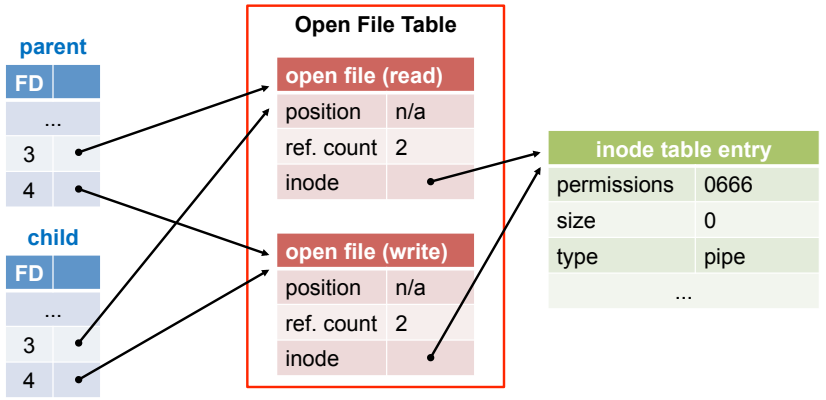
Opening a pipe

```
int fd[2];
pipe(fd);
```



After a fork

```
int fd[2];
pipe(fd);
fork();
```



An example with `pipe()`

```
/* #include statements and most error checking omitted */

int main() {
  int fd[2];
  char buf[BUFSIZ];  /* BUFSIZ is defined in stdio.h */

  if (pipe(fd) < 0)
    err(EX_OSERR, "pipe error");
  if (fork()) {  /* parent code */
    close(fd[1]);
    read(fd[0], buf, BUFSIZ);
    printf("Message from child: %s\n", buf);
  }
  else {  /* child code */
    char msg[] = "Hi there!";
    close(fd[0]);
    write(fd[1], msg, sizeof(msg));
  }
  return 0;
}
```


Standard I/O vs. Unix I/O

- Standard I/O (**`FILE *`**, **`fgets()`**, **`printf()`**, etc.) generally features buffers; Unix I/O does not
- Mixing the two can cause weird things to happen; what does this output?

```
printf("u");
write(STDOUT_FILENO, "m", 1);
printf("d\n");
```

- **`mud`**
- Even stranger things happen with reading...

Mixing buffered/unbuffered reads

`buf.c`:

```
#include <stdio.h>
#include <unistd.h>

#define BUFFER_SZ 100
static char buffer[BUFFER_SZ];

int main() {
  char line_s[11];
  char line_u[11] = {0};
  setbuffer(stdin, buffer, BUFFER_SZ);
  fgets(line_s, 11, stdin);
  read(STDIN_FILENO, line_u, 10);
  printf("L1: %s\n", line_s);
  printf("L2: %s\n", line_u);
  fgets(line_s, 11, stdin);
  read(STDIN_FILENO, line_u, 10);
  printf("L3: %s\n", line_s);
  printf("L4: %s\n", line_u);
  return 0;
}
```

`data.txt`:

```
aaaaaaaaaabbbbbbbbbbccccccccccc
ddddddddddeeeeeeeeeeffffffffff
gggggggggghhhhhhhhhhiiiiiiiiii
jjjjjjjjjjkkkkkkkkkklllllllll
mmmmmmmmmmnnnnnnnnnnoooooooooo
```

Execution:

```
$ ./buf < data.txt
L1: aaaaaaaaaa
L2: kkkkkkkkkk
L3: bbbbbbbbbb
L4: lllllllll
```


Sections 8.2-8.5, Bryant and O'Hallaron

More On Process Control

Tools for working with processes

- **`ps`**: UNIX command to see the current list of processes
  - several different options (**`-e`**, **`-f`**, **`-p`**, **`-u`**, etc.)
- **`strace`**: Linux tool to print trace of all system calls a program and its children perform (precedes rest of command line)
- **`pgrep`** *name*: prints pids of processes with *name* in their command lines
- **`pkill`** *name*: sends a signal to all processes with *name* in the command lines
- **`kill`** *pid-list*: sends a signal to all specified processes
- **`top`**: display current info about system and processes
- Ctrl+Z, **`bg`**, **`fg`**, and **`&`**: process backgrounding
- **`/proc`**: a (virtual) part of the filesystem on Linux that has all sorts of info on kernel data structures related to processes, and other OS related system info

Signals

- A message to a process to notify it of an event
- Kernel notifies processes of many events:
  - **`SIGSEGV`**: segmentation violation (aka segfault)
  - **`SIGFPE`**: floating point exception
  - **`SIGCHLD`**: child process stopped/terminated
- Users can trigger sending of signals:
  - **`SIGINT`**: interrupt (Ctrl-C)
  - **`SIGQUIT`**: quit and dump core (Ctrl+\)
  - **`SIGTSTP`**: terminal stop, (Ctrl+Z)

Signals, cont.

- Processes can also send signals to each other (via the kernel)
- Processes have default actions when receiving signals (sometimes ignore, sometimes quit)
- There are mechanisms for processes to block signals, but two (**`SIGKILL`** and **`SIGSTOP`**) can't be blocked
- Signals are sent with the **`kill`** UNIX command, or in a program using the **`kill()`** function
  - they do not always send the signal **`SIGKILL`**!