CMPSCI 230
Computer Systems Principles

More on Processes
Objectives

• To learn how to control child processes
• To learn about other process-related functions
• To learn how to load and execute programs
• To learn about signals
• To learn about pipes
Create & Terminate Processes

We use the fork() function to create a new process.

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```
Create & Terminate Processes

What happens here?

```c
void main()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

01_zombie.c

Zombie!
01_zombie.c

Linux

student@osboxes:code$ ./01_zombie
Running Parent, PID = 1176
Terminating Child, PID = 1177
^Z
[1]+  Stopped                  ./01_zombie
student@osboxes:code$ ps
       PID  TTY          TIME   CMD
 1136 pts/0    00:00:00   bash
 1176 pts/0    00:00:03 01_zombie
 1177 pts/0    00:00:00 01_zombie <defunct>
 1179 pts/0    00:00:00   ps
student@osboxes:code$
ridgway@swift:code$ ./01_zombie
Running Parent, PID = 25768
Terminating Child, PID = 25769
^Z
[1]+  Stopped                 ./01_zombie
ridgway@swift:code$ ps
   PID TTY           TIME CMD
  444 ttys000    0:00.92   -bash
  25768 ttys000    0:02.44 ./01_zombie
25769 ttys000    0:00.00 (01_zombie)
ridgway@swift:code$
Create & Terminate Processes

What about this one?

```c
void main()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    } else { /* Infinite loop */
        printf("Terminating Parent, PID = %d\n", getpid());
        exit(0);
    }
}
```

02_zombie.c

Zombie?
02_zombie.c

```
ridgway@swift:code$ ./02_zombie
Terminating Parent, PID = 25820
Running Child, PID = 25821
```
```
ridgway@swift:code$ ps -f
   UID  PID  PPID   C STIME      TTY           TIME   CMD
 501  444   443   0  5:47PM ttys000  0:00.95 -bash
 501 25821     1   0 10:31AM ttys000  0:02.05 ./02_zombie
```

NOTE!
How do we control processes?

Need a way to get rid of *kill* the zombies!
How do we control processes?

Need a way to get rid of (kill) the zombies!

This is called reaping!
Create & Terminate Processes

So, how do we “reap” a child process programmatically?

Zombie?
Create & Terminate Processes

So, how do we “reap” a child process programmatically?

wait()  
waitpid()
Create & Terminate Processes

int wait(int *child_status)

```c
void main()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0) { // Child
            exit(100+i);
        }
    }  
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status)) {
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        } else {
            printf("Child %d terminated abnormally\n", wpid);
        }
    }  
}
```
Create & Terminate Processes

int wait(int *child_status)

void main()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0) { // Child
            exit(100+i);
        }
    }
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status)) {
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        } else {
            printf("Child %d terminated abnormally\n", wpid);
        }
    }
}
Create & Terminate Processes

```
int wait(int *child_status)
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0) { // Child
            exit(100+i);
        }
    }
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status)) {
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        } else {
            printf("Child %d terminated abnormally\n", wpid);
        }
    }
}
```
Create & Terminate Processes

```c
int waitpid(pid, &status, options) {
  // ... implementation ...
}
```

```c
void main() {
  pid_t pid[N];
  int i;
  int child_status;
  for (i = 0; i < N; i++) {
    if ((pid[i] = fork()) == 0) { // Child
      exit(100+i);
    }
  }
  for (i = N-1; i >= 0; i--) {
    pid_t wpid = waitpid(pid[i], &child_status, 0);
    if (WIFEXITED(child_status)) {
      printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
    } else {
      printf("Child %d terminated abnormally\n", wpid);
    }
  }
}
```

wait for specific child to terminate
Create & Terminate Processes

\[
\text{int waitpid}(-1, \ &\text{status}, \ 0) \\
\text{is the same as...} \\
\text{int wait}(&\text{status})
\]
void main()
{
  if (fork() == 0) {
    printf("a");
  }
  else {
    printf("b");
    waitpid(-1, NULL, 0);
  }
  printf("c");
  exit(0);
}

What is the output of the program on the left?

A. acbc
B. bcac
C. abcc
D. bacc
E. A or C or D
void main()
{
    if (fork() == 0) {
        printf("a");
    }
    else {
        printf("b");
        waitpid(-1, NULL, 0);
    }
    printf("c");
    exit(0);
}

What is the output of the program on the left?

A. acbc
B. bcac
C. abcc
D. bacc
E. A or C or D
Objectives

• To learn how to control child processes
• To learn about sleep and wait
• To learn how to load and execute programs
• To learn about signals
• To learn about pipes
Putting a process to sleep: zzzzz...

unsigned int sleep(unsigned int secs);

#include <unistd.h>
void main()
{
    int amt = sleep(5);
}

Returns 0 if time has elapsed.
Returns non-0 if a signal woke up the process.
Putting a process to sleep: zzzzz...

```c
#include <unistd.h>
void main()
{
    int amt = pause();
}
```

int pause(void);

Returns the amount of time the process was paused.
Objectives

• To learn how to control child processes
• To learn about `sleep` and `wait`
• To learn how to load and execute programs
• To learn about signals
• To learn about pipes
Loading and Running Programs

What does a shell program do?
Loading and Running Programs

What does a shell program do?

A shell program is a special kind of program whose primary responsibility is to run other programs.
Loading and Running Programs

```c
int execve(const char* filename,
            const char* argv[],
            const char* envp[]);
```
Loading and Running Programs

```c
int execve(const char* filename,
            const char* argv[],
            const char* envp[]);
```

We know what these are...
Loading and Running Programs

```c
int execve(const char* filename, const char* argv[], const char* envp[]);
```

We know what these are...

But, what is this?
Loading and Running Programs

```c
int execve(const char* filename,
            const char* argv[],
            const char* envp[]);
```

Every program runs in an “environment”. The environment is customized using environment variables: PATH, EDITOR, ...

```
[elnux1:~] printenv
XDG_SESSION_ID=7763
HOSTNAME=elnux1.cs.umass.edu
TERM=xterm-256color
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=128.119.40.196 58373 22
SSH_TTY=/dev/pts/10
USER=richards
```

```
envp
```

```
envp[0]  "USER=richards"
envp[1]  "SHELL=/bin/bash"
...
envp[n-1]
NULL    "EDITOR=emacs"
```
int execve(const char* filename,
            const char* argv[],
            const char* envp[]);

It turns out that the more general form for main is:

int main(int argc,
          const char* argv[],
          const char* envp[]);
Loading and Running Programs

Where are argv and envp in memory when a process executes?
Loading and Running Programs

Where are argv and envp in memory when a process executes?

Stack frame for main

Bottom of stack

Top of stack

Null-terminated environment variable strings
Null-terminated command-line argument strings
unused space
envp[n] == NULL
envp[n-1]
...
envp[0]
argv[argc] == NULL
argv[argc-1]
...
argv[0]
(Dynamic Linker Variables)
envp
d argv
argc
Stack frame for main
## Loading and Running Programs

### Where are `argv` and `envp` in memory when a process executes?

- `argv` and `envp` are in memory when a process executes. `argv` is the array of command-line arguments, and `envp` is the array of environment variables.

### So, how do we access the environment variables?

Access to the environment variables is done through the `envp` array. Each element in `envp` is a string of the form `key=value`, where `key` is the environment variable name and `value` is the value associated with that environment variable.

### Null-terminated strings

- `argv[argc-1] == NULL`
- `argv[0]` is the first argument
- `argv[0]` is the name of the program
- `argv[0]` is not part of the array
- `argv` includes dynamic linker variables

### Null-terminated environment variable strings

- `envp[n-1] == NULL`
- `envp[0]` is the first environment variable

### Stack frame for main

- `argc` and `argv` are part of the stack frame for the main function.

---

<table>
<thead>
<tr>
<th>Stack frame for main</th>
<th>(Dynamic Linker Variables)</th>
<th>Null-terminated command-line argument strings</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>argc</code></td>
<td><code>argv</code></td>
<td>Unused space</td>
</tr>
<tr>
<td><code>argv</code></td>
<td><code>envp</code></td>
<td>Null-terminated environment variable strings</td>
</tr>
<tr>
<td><code>envp</code></td>
<td><code>argv[argc-1]</code></td>
<td></td>
</tr>
<tr>
<td><code>argv[argc]</code></td>
<td><code>argv[0]</code></td>
<td></td>
</tr>
<tr>
<td><code>argv[0]</code></td>
<td><code>argv[argc-1]</code></td>
<td></td>
</tr>
<tr>
<td><code>argv[0]</code></td>
<td><code>argv[0]</code></td>
<td></td>
</tr>
</tbody>
</table>

---

**Stack frame for main**

- Bottom of stack
- Top of stack
Loading and Running Programs

```c
int* getenv(const char* name);

Returns pointer to value if there is an entry of the form “name=value”, and NULL if there is not

int setenv(const char* name, const char* newval, int overwrite);

Returns 0 on success, -1 on error.

void unsetenv(const char* name);

Returns nothing.
```

03_setenv.c
What does a shell program do?

A shell program is a special kind of program whose primary responsibility is to run other programs.
Loading and Running Programs

What does a shell program do?

fork + execve

A shell program is a special kind of program whose primary responsibility is to run other programs.
Objectives

• To learn how to control child processes
• To learn about `sleep` and `wait`
• To learn how to load and execute programs
• To learn about signals
• To learn about pipes
Signals

How do we communicate to processes?

We send them messages called signals

A signal is an event of some type that has occurred in the system.

It allows the OS to communicate to a process AND user processes to communicate to each other.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Action</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>1</td>
<td>Term</td>
<td>Hangup detected on controlling terminal or death of controlling process</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Term</td>
<td>Interrupt from keyboard</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>3</td>
<td>Core</td>
<td>Quit from keyboard</td>
</tr>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>Core</td>
<td>Illegal Instruction</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>Core</td>
<td>Abort signal from abort(3)</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>Core</td>
<td>Floating point exception</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Term</td>
<td>Kill signal</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>11</td>
<td>Core</td>
<td>Invalid memory reference</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>14</td>
<td>Term</td>
<td>Broken pipe: write to pipe with no readers</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>15</td>
<td>Term</td>
<td>Timer signal from alarm(2)</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>15</td>
<td>Term</td>
<td>Termination signal</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>30,10,16</td>
<td>Term</td>
<td>User-defined signal 1</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>31,12,17</td>
<td>Term</td>
<td>User-defined signal 2</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>20,17,18</td>
<td>Ign</td>
<td>Child stopped or terminated</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>19,18,25</td>
<td>Cont</td>
<td>Continue if stopped</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>17,19,23</td>
<td>Stop</td>
<td>Stop process</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>18,20,24</td>
<td>Stop</td>
<td>Stop typed at tty</td>
</tr>
<tr>
<td>SIGTIN</td>
<td>21,21,26</td>
<td>Stop</td>
<td>TTY input for background process</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>22,22,27</td>
<td>Stop</td>
<td>TTY output for background process</td>
</tr>
</tbody>
</table>
Signals

How do we communicate to processes?

We send them messages called \textit{signals}

A \textit{signal} is an \textit{event} of some type that has occurred in the system.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Action</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Term</td>
<td>Interrupt from keyboard</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Core</td>
<td>Illegal instruction</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Core</td>
<td>Kill signal</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>19,25</td>
<td>Cont</td>
<td>Continue if stopped</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>17,23</td>
<td>Stop</td>
<td>Stop process</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>1</td>
<td>Stop</td>
<td>Stop typed at tty</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>18</td>
<td>Stop</td>
<td>Stop by input for background process</td>
</tr>
<tr>
<td>SIGIO</td>
<td></td>
<td>Stop</td>
<td></td>
</tr>
<tr>
<td>SIGURG</td>
<td></td>
<td>Stop</td>
<td></td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>30,16</td>
<td>Term</td>
<td>User-defined signal 1</td>
</tr>
<tr>
<td>SIGTRAP</td>
<td>4</td>
<td>Core</td>
<td>Trap signal</td>
</tr>
<tr>
<td>SIGSYS</td>
<td>15</td>
<td>Term</td>
<td>Broken pipe. Write to pipe with no readers</td>
</tr>
<tr>
<td>SIGPWR</td>
<td></td>
<td>Term</td>
<td>Power signal</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>2</td>
<td>Term</td>
<td>Alarm signal</td>
</tr>
<tr>
<td>SIGPROF</td>
<td>3</td>
<td>Term</td>
<td>Prof signal</td>
</tr>
<tr>
<td>SIGSEV</td>
<td>21</td>
<td>Stop</td>
<td>System signal</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>15</td>
<td>Term</td>
<td>Terminating signal</td>
</tr>
<tr>
<td>SIGSTK</td>
<td>18</td>
<td>Stop</td>
<td>Stack trace</td>
</tr>
<tr>
<td>SIGWIN</td>
<td></td>
<td>Stop</td>
<td></td>
</tr>
<tr>
<td>SIGWIN</td>
<td></td>
<td>Stop</td>
<td></td>
</tr>
</tbody>
</table>

If a process tries to divide by 0: OS sends it a SIGFPE signal

If a process executes an illegal instruction: OS sends it a SIGILL signal

If a process makes illegal memory reference: OS sends it a SIGSEGV signal

It allows the OS to communicate to a process AND user processes to communicate to each other.
Signals

How do we communicate to processes?

We send them messages called *signals*

A *signal* is an *event* of some type that has occurred in the system.

If you type `ctrl-c` during process exec
OS sends it a SIGINT signal

A process can kill another process:
P1 sends P2 a SIGKILL signal

When a child terminates:
OS sends the parent a SIGCHLD signal

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Action</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Term</td>
<td>Interrupt from keyboard</td>
</tr>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>Core</td>
<td>Illegal instruction</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>Core</td>
<td>Abort signal from abort(3)</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>Core</td>
<td>Floating point exception</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Term</td>
<td>Kill signal</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>11</td>
<td>Term</td>
<td>Bus error</td>
</tr>
<tr>
<td>SIGTRAN</td>
<td>12</td>
<td>Term</td>
<td>Termination request from kernel</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>20,17,18</td>
<td>Ign</td>
<td>Child stopped or terminated</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>30,10,16</td>
<td>Term</td>
<td>User-defined signal 1</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>19,18,25</td>
<td>Ign</td>
<td>Continue if stopped</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>17,19,23</td>
<td>Stop</td>
<td>Stop process</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>18,20,24</td>
<td>Stop</td>
<td>Stop typed at tty</td>
</tr>
<tr>
<td>SIGTTK</td>
<td>17,22,29</td>
<td>Stop</td>
<td>Stop typed at tty of foreground process</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>12,22,29</td>
<td>Stop</td>
<td>Stop typed at tty of background process</td>
</tr>
</tbody>
</table>
Signals

How do we communicate to processes?

The transfer of a signal to a destination occurs in 2 steps

1. Sending a Signal
2. Receiving a Signal
Signals

How do we communicate to processes?

Sending a Signal

The kernel sends a signal to a destination process by updating some state in the context of that process.

This occurs when the kernel detects a system event (div by 0) or when a process invokes the kill system call to explicitly request the kernel to send a signal to the destination process.
Signals

How do we communicate to processes?

Receiving a Signal

A destination process receives a signal when it is forced by the kernel to react in some way.

The process can either ignore the signal, terminate, or catch the signal by executing user-level functions called signal handlers.
Signals

How do we communicate to processes?

• **A signal** is *pending* if sent but not yet received
  – There can be at most 1 pending signal for a type
  – **Important**: signals are *not queued*
    • If a process has a pending signal of type $k$, then subsequent signals of type $k$ that are sent to that process are discarded.

• **A process can block** the receipt of certain signals
  – Blocked signals can be delivered, but will not be received until the signal is *unblocked*. 
Signals

So, how are they implemented?
Signals

So, how are they implemented?

Each process structure contains a **pending** and **blocked** bit vector. Each entry corresponds to a specific *signal*. 
Signals

So, how are they implemented?

A process can decide to **block** a signal by setting the corresponding bit in the *blocked* bit vector.
Signals

So, how are they implemented?

Not all signals can be blocked:
SIGKILL can’t be blocked or handled by the process.
Signals

So, how are they implemented?

Imagine the processor encounters an illegal instruction during the execution of Process 1
Signals

So, how are they implemented?

The kernel will set the corresponding bit in the Pending bit vector.
Signals

So, how are they implemented?

Just before Process 1 resumes execution the kernel will check to see if the process has any pending signals that are not blocked.
Signals

So, how are they implemented?

If it does it will check if the process has defined a signal handler for the signal or execute default behavior.
Signals

So, how are they implemented?

The default behavior for SIGILL is to *terminate* the process.
Signals

So, how are they implemented?

A process can send another process a signal using the `kill` system call.

```c
#include <signal.h>
int kill(pid_t pid, int sig);
```
Signals
So, how are they implemented?

A process can send another process a signal using the **kill** system call

```c
#include <signal.h>
int kill(pid_t pid, int sig);
```
Signals

So, how are they implemented?

#include <signal.h>
int kill(pid_t pid, int sig);

Again, the kernel will check for pending signals, check for a handler or execute default behavior.
Signals

So, how are they implemented?

Again, the kernel will check for pending signals, check for a handler or execute default behavior.

```c
#include <signal.h>
int kill(pid_t pid, int sig);
```
Signals

We can send signals from the keyboard

ctrl-c
Typing ctrl-c from the keyboard sends a SIGINT signal to the shell.

The shell catches the signal and then sends a SIGINT to every process in the foreground process group.

ctrl-z
Typing ctrl-z from the keyboard sends a SIGTSTP signal to the shell.

The shell catches the signal and then sends a SIGTSTP to every process in the foreground process group.
Signals

Ok, so how do we send signals programmatically?

```c
#include <signal.h>
int kill(pid_t pid, int sig);
```

05_kill_prog.c

06_handler.c
Signals

And how do we catch a signal?

```c
#include <unistd.h>
unsigned int alarm(unsigned int secs);
```

Generates a SIGALRM signal to the calling process after `secs` seconds.

Need a handler to “catch” the signal and do something interesting.

07_alarm.c     08_sigint.c
Objectives

• To learn how to control child processes
• To learn about `sleep` and `wait`
• To learn how to load and execute programs
• To learn about signals
• To learn about pipes
How else can processes communicate?

• **Different Address Space**
  – Great for protection
  – Hard for communication
  – Can’t share data structures

• **Is there another way to communicate?**
  – Yes, through files...
  – But, we need to understand how files work at a lower level...
Low-Level Files

• Two system calls
  – int open(char *path, int flags)
  – int close(int fd)

• At this level we work with file descriptors

• What is a file descriptor?
  – An integer that is used to reference an open file inside the operating system
  – Processes share open files!
How the Unix Kernel Represents Open Files

- Two descriptors referencing two distinct open disk files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file

Descriptortable
[one table per process]

Open file table
[shared by all processes]

v-node table
[shared by all processes]

File A (terminal)
File B (disk)

Info in stat struct
File Sharing

- Two distinct descriptors sharing the same disk file through two distinct open file table entries

  - E.g., Calling `open` twice with the same `filename` argument

Descriptortable
[one table per process]

Open file table
[shared by all processes]

v-node table
[shared by all processes]

stdin  fd 0
stdout fd 1
stderr fd 2
fd 3
fd 4

File A (disk)

File pos
refcnt=1

File B (disk)

File pos
refcnt=1

File access
File size
File type
How Processes Share Files: Fork()

- A child process inherits its parent’s open files

**Before** fork() call:

- **Descriptor table**
  - [one table per process]

- **Open file table**
  - [shared by all processes]

- **v-node table**
  - [shared by all processes]

```
stdin  fd 0
stdout fd 1
stderr fd 2
  fd 3
  fd 4

File A (terminal)
- File pos
- refcnt=1
  : File access
  : File size
  : File type

File B (disk)
- File pos
- refcnt=1
  : File access
  : File size
  : File type

v-node table
```

- Before fork():
  - fd 0 (stdin)
  - fd 1 (stdout)
  - fd 2 (stderr)
  - fd 3 (fd 3)
  - fd 4 (fd 4)
How Processes Share Files: Fork()

- A child process inherits its parent’s open files
- *After* `fork()`:
  - Child’s table same as parent’s, and +1 to each refcnt

**Diagram:**

- **Descriptor table** (one table per process)
- **Open file table** (shared by all processes)
- **v-node table** (shared by all processes)

**Parent**
- fd 0
- fd 1
- fd 2
- fd 3
- fd 4

**Child**
- fd 0
- fd 1
- fd 2
- fd 3
- fd 4

**File A (terminal)**
- File pos
- refcnt=2

**File B (disk)**
- File pos
- refcnt=2

**Nodes**
- File access
- File size
- File type
Pipes

• From the command line:

   $ cat hello.txt | program

• We use the pipe operator | to communicate between two running processes. So, how do we do this between processes?
pipe system call

- `int pipe(int pipefd[2])`

The `pipe` system call will create a “pipe” between two running processes.

The pipe system call will open two “files”:
- `pipefd[0]` for reading
- `pipefd[1]` for writing
Pipes in Action

Process 1 writes to pipefd[1]

Process 0 reads from pipefd[0]

09_pipes.c