CS143A
Principles on Operating Systems
Discussion 02:
OS Interfaces

Instructor: Prof. Anton Burtsev
TA: Saehanseul Yi (Hans)
Oct 16, 2020 Noon
About me

• Link for all office hours/discussion: https://uci.zoom.us/j/93369206818

• Teaching staff office hours:
  
  Hari: *Mon* 12:00 PST
  *Zhaofeng Li: Tue 12:00 PST*
  Deep: *Wed 9:00 AM PST*
  **Hans: *Thu 12:00 PST***
  Se-Min Lim: *Fri 9:00 PST*
Motivating example: redirection

- Best example for explaining pipe(), fork(), and exec()
- Program output -> stdout (default: screen)
- | (pipe operator): send outputs to somewhere else

```bash
$ ls
a.out b.out asdfasdf
$
$ ls | grep asdf
asdfasdf
$```
System calls are the interface of the OS.
But what is shell?

- Normal process
  - Kernel starts it for each user that logs in into the system
  - In xv6 shell is created after the kernel boots
- Shell interacts with the kernel through system calls
  - E.g., starts other processes
System calls, interface for...

- **Processes**
  - Creating, exiting, waiting, terminating
- **Memory**
  - Allocation, deallocation
- **Files and folders**
  - Opening, reading, writing, closing
- **Inter-process communication**
  - Pipes

Child process (left)

Child process (right)

parent process (shell)
Wait... stdin? stdout?
(standard input, standard output)
Wait... stdin? stdout?
(standard input, standard output)

```bash
$ ls | grep asdf
asdfasdf
$
```

**Diagram:**
- STDIN (keyboard) flows through process symbols:
  - `ls` (input) -> `grep` (output) -> terminal
  - `ls` (input) -> terminal
  - `grep` (input) -> terminal

[Diagram showing standard input (STDIN) flowing through processes and standard output (STDOUT) arrows]

[saehansy@uci.edu]
Wait... stdin? stdout?
(standard input, standard output)

$ ls | grep asdf
asdfasdf
$

$
Wait... stdin? stdout?

(standard input, standard output)

$ ls | grep asdf
asdfasdf
$

- stdin(0), stdout(1), and stderr(2) are file descriptors (just an integer in user-program)
Wait... stdin? stdout?
(standard input, standard output)

$ ls | grep asdf
asfasdf
$

- stdin(0), stdout(1), and stderr(2) are file descriptors (i.e. just an integer in user-program)
- Each program has its own descriptor table
Wait... stdin? stdout?

(standard input, standard output)

• stdin(0), stdout(1), and stderr(2) are file descriptors (i.e., just an integer in user-program)
• Each program has its own descriptor table
• How to modify process’ file descriptors?
Wait... stdin? stdout?
(standard input, standard output)

- stdin(0), stdout(1), and stderr(2) are file descriptors (*just an integer* in user-program)
- Each program has its own descriptor table
- How to modify process’ file descriptors?
  - close, dup(or open)
Wait... stdin? stdout?
(standard input, standard output)

• stdin(0), stdout(1), and stderr(2) are file descriptors (just an integer in user-program)
• Each program has its own descriptor table
• How to modify process’ file descriptors?
  • close, dup(or open)
• What we need to do:
close appropriate descriptors for each process and set the appropriate descriptor by copying
Wait... stdin? stdout?
(standard input, standard output)

• stdin(0), stdout(1), and stderr(2) are file descriptors (*just an integer* in user-program)
• Each program has its own descriptor table
• How to modify process’ file descriptors?
  • close, dup(or open)
• What we need to do: close appropriate descriptors for each process and set the appropriate descriptor by copying

POSIX.1-2001

pipe() creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. filedes[0] is for reading, filedes[1] is for writing.

pipe is uni-directional
pipe() and fork()

case PIPE:
    pcmd = (struct pipecmd*) cmd;
    if(pipe(p) < 0)
        panic("pipe");
---------------------
Point A---------------------
if(fork1() == 0){
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
---------------------
Point B---------------------
    runcmd(pcmd->left);
}
if(fork1() == 0){
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->right);
}
close(p[0]);
close(p[1]);
---------------------
Point C---------------------
wait();
wait();
break;

int pipe(int pipefd[2]);
Create a pipe & assign each end to pipefd

pid_t fork(void);
Copy the current process (parent)
Returns the PID of the child (parent) or 0 (child)
pipe() and fork()

---------------------Point 0---------------------
case PIPE:
   pcmd = (struct pipecmd*)cmd;
   if(pipe(p) < 0)
      panic("pipe");
---------------------Point A---------------------
   if(fork1() == 0){
      close(1);
      dup(p[1]);
      close(p[0]);
      close(p[1]);
   }
---------------------Point B---------------------
   runcmd(pcmd->left);

※ Throughout the example, stderr is always connected to the screen. Omitted for simplicity as well as p[0] and p[1] to the parent process
pipe() and fork()

case PIPE:
  pcmd = (struct pipecmd*)cmd;
  if(pipe(p) < 0)
    panic("pipe");

if(fork1() == 0){
  close(1);
  dup(p[1]);
  close(p[0]);
  close(p[1]);
}

runcmd(pcmd->left);
pipe() and fork()

---------------------Point 0---------------------
case PIPE:
    pcmd = (struct pipecmd*)cmd;
    if(pipe(p) < 0)
        panic("pipe");
---------------------Point A---------------------
if(fork1() == 0){
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
---------------------Point B---------------------
runcmd(pcmd->left);
}
pipe() and fork()

case PIPE:
    pcmd = (struct pipecmd*)cmd;
    if(pipe(p) < 0)
        panic("pipe");

if(fork1() == 0){
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd-left);
}

fork() copies the descriptors too!

--- Point A ---

--- Point B ---

Executed by child process
pipe() and fork()

---Point 0---
case PIPE:
    pcmd = (struct pipecmd*)cmd;
    if(pipe(p) < 0)
        panic("pipe");
---Point A---
    if(fork1() == 0){
        close(1);
        dup(p[1]);
        close(p[0]);
        close(p[1]);
    }
    Executed by child process
---Point B---
runcmd(pcmd->left);

fork() copies the descriptors too!
pipe() and fork()

--- Point 0 ---

```c
case PIPE:
    pcmd = (struct pipecmd*)cmd;
    if(pipe(p) < 0)
        panic("pipe");
```

--- Point A ---

```c
if(fork1() == 0){
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
}
```

--- Point B ---

```c
runcmd(pcmd->left);
```
pipe() and fork()

--- Point 0 ---

```c
case PIPE:
    pcmd = (struct pipecmd*)cmd;
    if(pipe(p) < 0)
        panic("pipe");
```

--- Point A ---

```c
if(fork1() == 0){
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
}
```

--- Point B ---

```c
runcmd(pcmd->left);
```
pipe() and fork()

case PIPE:
    pcmd = (struct pipecmd*) cmd;
    if (pipe(p) < 0)
        panic("pipe");

if (fork1() == 0){
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
}

fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

--- Point A ---
--- Point B ---

Executed by child process

saehansy@uci.edu
**pipe() and fork()**

```
Point B

```runcmd(pcmd>left);
}
if(fork1() == 0){
close(0);
dup(p[0]);
close(p[0]);
close(p[1]);
runcmd(pcmd->right);
}
close(p[0]);
close(p[1]);
```

**Point C**

```wait();
wait();
break;```

---

**fork() copies the descriptors too!**
**dup()’s destination is the lowest & unused file descriptor!**

<table>
<thead>
<tr>
<th>PARENT PROCESS</th>
<th>CHILD PROCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdin 0</td>
<td>stdin 0</td>
</tr>
<tr>
<td>stdout 1</td>
<td>stdout 1</td>
</tr>
<tr>
<td>stderr 2</td>
<td>stderr 2</td>
</tr>
<tr>
<td>p[0] 3</td>
<td>p[0] 3</td>
</tr>
</tbody>
</table>

Executed by child process
pipe() and fork()

---------------------Point B---------------------

runcmd(pcmd>left);

}  
if(fork1() == 0){
  close(0);
  dup(p[0]);  
  close(p[0]);
  close(p[1]);
  runcmd(pcmd->right);
}  
close(p[0]);
close(p[1]);

---------------------Point C---------------------

wait();
wait();
break;

fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!
pipe() and fork()

```c
if (fork1() == 0) {
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd -> right);
}
```

fork() copies the descriptors too! dup()'s destination is the lowest & unused file descriptor!

```c
close(p[0]);
close(p[1]);
```

```
wait();
wait();
break;
```

Point B

Point C
problem: pipe() and fork()

---Point B---
runcmd(pcmd>left);

} if(fork1() == 0){
close(0);
dup(p[0]);
close(p[0]);
close(p[1]);
runcmd(pcmd->right);
}
close(p[0]);
close(p[1]);

---Point C---
wait();
wait();
break;

fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

dup()'s destination is the lowest & unused file descriptor!
pipe() and fork()

---------------------Point B---------------------
  runcmd(pcmd>left);
}
if(fork1() == 0){
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd->right);
}
close(p[0]);
close(p[1]);
---------------------Point C---------------------
wait();
wait();
break;

fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!
pipe() and fork()

---------------------Point B---------------------

runcmd(pcmd>left);

}  
if(fork1() == 0){
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd->right);
}

close(p[0]);
close(p[1]);
---------------------Point C---------------------

wait();
wait();
break;

fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

dup()'s destination is the lowest & unused file descriptor!
pipe() and fork()

----------Point B----------

runcmd(pcmd>left);

if(fork1() == 0){
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->right);
}

close(p[0]);
close(p[1]);

----------Point C----------

wait();
wait();
break;

fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

dup()'s destination is the lowest & unused file descriptor!

Parent waits child processes
pipe() and fork() and exec()

if(fork1() == 0){
    ...
    runcmd(pcmd->right);
}

runcmd() contains exec functions

$ ls | grep asdf
asdfasdf
$

int execvp(const char *file, char *const argv[]);
replaces the current process image with a new process image.
pipe() and fork() and exec()

```
if (fork1() == 0) {
  ...
  runcmd(pcmd->right);
}
```

$runcmd()$ contains exec functions

```
$ ls | grep asdf
asdfasdf
$
```

```
int execvp(const char *file, char *const argv[]);
replaces the current process image with a new process image.
```
pipe() and fork() and exec()

if(fork1() == 0){
  ...
  runcmd(pcmd->right);
}

*runcmd() contains exec functions*

```bash
$ ls | grep asdf
asdfasdf
$
```

```c
int execvp(const char *file, char *const argv[]);
```

replaces the current process image with a new process image.
pipe() and fork() and exec()

if(fork1() == 0){
    ...
    runcmd(pcmd->left);
}

runcmd() contains exec functions

```c
int execvp(const char *file, char *const argv[]);
```
replaces the current process image with a new process image.

```sh
$ ls | grep asdf
asdfasdf
$