Shell and signals

Mythili Vutukuru

CSE, IIT Bombay

Recap: system calls for process management

- `fork()` creates a new child process
  - All processes are created by forking from a parent
  - OS starts `init` process after boot up, which forks other processes
  - The `init` process is ancestor of all processes
- `exec()` makes a process execute a given executable
- `exit()` terminates a process
- `wait()` causes a parent to block until child terminates

Shell / Terminal

- After bootup, the `init` process is first process created
- The `init` process spawns a shell like `bash`
- All future processes are created by forking from existing processes like `init` or shell
- Shell reads user command, forks a child, execs the command executable, waits for it to finish, and reads next command
- Common commands like `ls, echo, cat` are all readily available executables that are simply exec-ed by the shell

Example shell code

- How does the shell run a user command?
- Read input from user
- Shell process forks a child process
- Child process runs exec with “echo” program executable as argument, calls exit when done
- Parent shell calls wait, blocks till child terminates, reaps it, goes back for next input

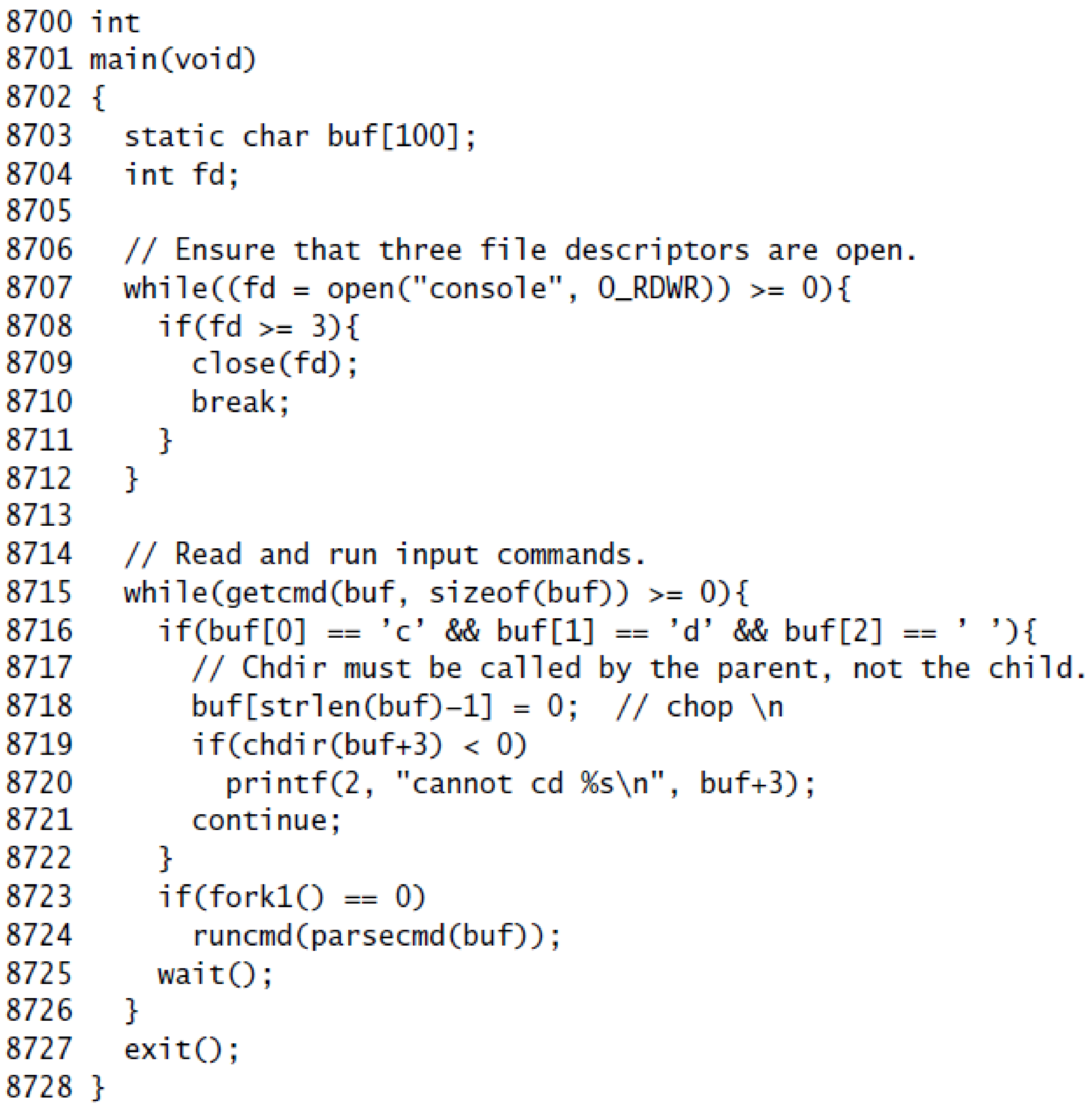
```
$echo hello
hello
$
```

```
do forever {
    input(command)

    int ret = fork()

    if(ret == 0) {
        exec(command)
    }
    else {
        wait()
    }
}
```

More on shell and commands

- Some commands already exist as programs written by OS developers and compiled into executables
  - Shell runs such command by simply calling exec in child process
- Some commands are implemented directly in shell code itself
- Think: why doesn't shell exec command directly? Why fork a child?
  - Do we want the shell program code to be rewritten fully?
- For "cd" command, "chdir" system call used to change directory of parent process itself, no child process is forked. Why?
  - Every process has a current working directory
  - Do we want to change directory of some child process or shell itself?

xv6: shell



```
int
main(void)
{
  static char buf[100];
  int fd;

  // Ensure that three file descriptors are open.
  while((fd = open("console", O_RDWR)) >= 0){
    if(fd >= 3){
      close(fd);
      break;
    }
  }

  // Read and run input commands.
  while(getcmd(buf, sizeof(buf)) >= 0){
    if(buf[0] == ’c’ && buf[1] == ’d’ && buf[2] == ’ ’){
      // Chdir must be called by the parent, not the child.
      buf[strlen(buf)-1] = 0;  // chop \n
      if(chdir(buf+3) < 0)
        printf(2, "cannot cd %s\n", buf+3);
      continue;
    }
    if(fork1() == 0)
      runcmd(parsecmd(buf));
    wait();
  }
  exit();
}
```

Foreground and background execution

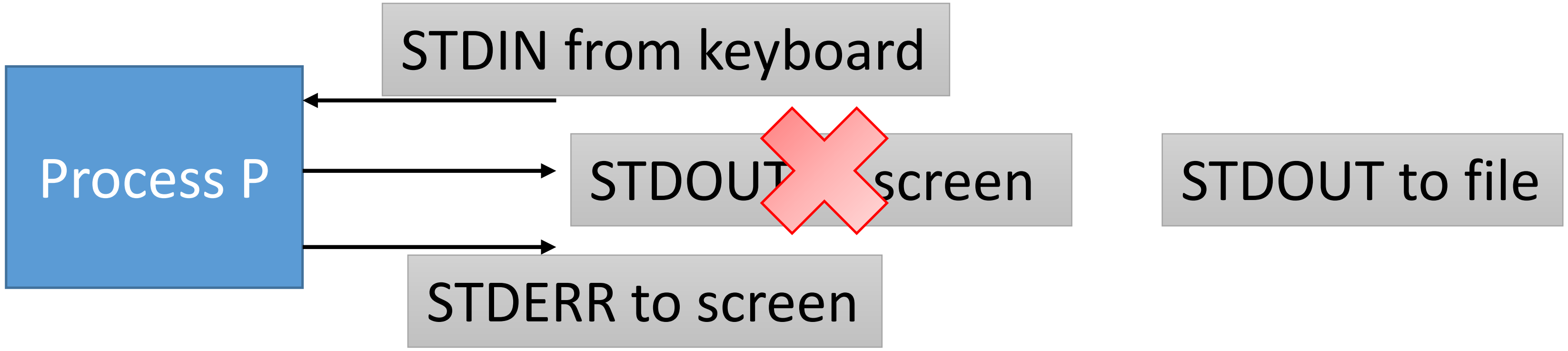
```
$sleep 10 &
$
```

- By default, user command runs in foreground, shell cannot accept next command until previous one finishes
- Background execution: when we type command followed by &
  - Shell starts child to run command, but does not wait for command to finish
- Background processes reaped at a later time by shell
  - When? Periodically? When next input is typed?
  - How? There is a way to invoke wait where parent is not blocked even if child has not exited (explore it on your own)
- It is also possible to run multiple commands in the foreground
  - One after the other serially (next command starts after previous finishes)
  - Or, all start at same time in parallel
  - Explore how such things can be done in the standard Linux shell

```
$ls > foo.txt
$
```

I/O redirection

- Every process has some I/O channels (“files”) open, which can be accessed by file descriptors
  - STDIN, STDOUT, STDERR open by default for all processes
- Parent shell can manipulate these file descriptors of child before exec in order to do things like I/O redirection
- E.g., output redirection is done by closing the default STDOUT and opening a regular file in its place



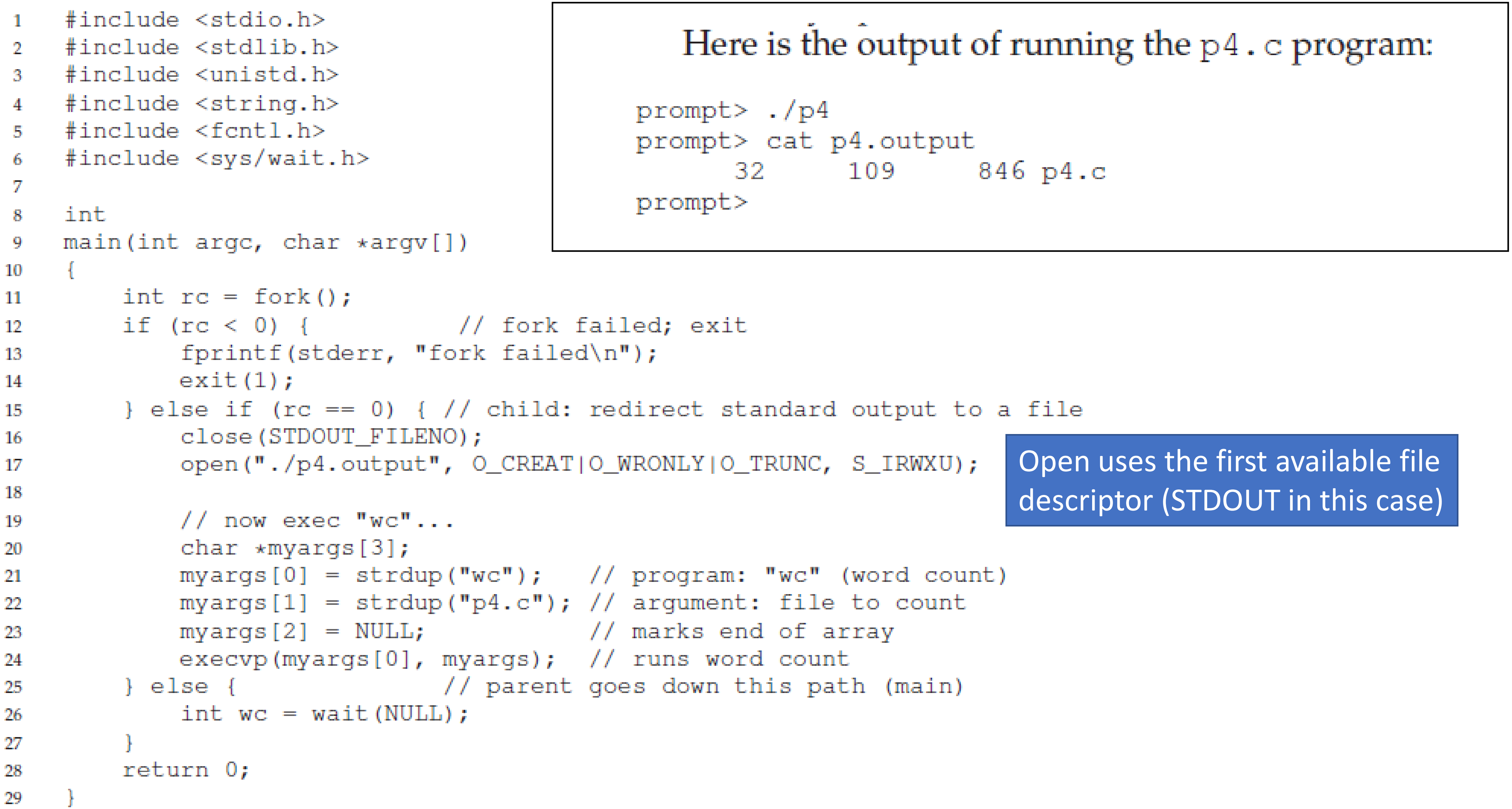
STDIN from keyboard

Process P

STDOUT screen

STDOUT to file

STDERR to screen



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child: redirect standard output to a file
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

        // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p4.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
    } else {              // parent goes down this path (main)
        int wc = wait(NULL);
    }
    return 0;
}
```

Here is the output of running the p4.c program:

```
prompt> ./p4
prompt> cat p4.output
       32     109     846 p4.c
prompt>
```

Open uses the first available file descriptor (STDOUT in this case)

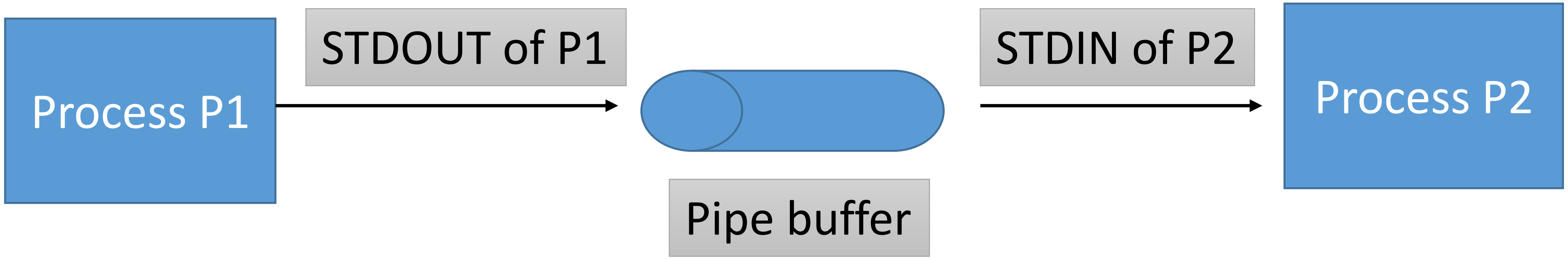
Figure 5.4: **All Of The Above With Redirection (p4.c)**

Image credit: OSTEP

Pipes

```
$cat foo.txt | grep something
```

- How does shell run commands with pipes (output of one command given as input to another command)?
- Shell uses a temporary “buffer” (memory) in the OS, called a “pipe”
- Shell creates child processes for commands, creates pipe buffer in OS, redirect stdout of one process to stdin of another process via the pipe



Signals

- Signal: a way to send notifications to processes
- Standard signals available in operating systems, each corresponding to a specific event, and with a specific signal number
  - Signal SIGINT sent to process by typing Ctrl+C, SIGSTP for Ctrl+D
  - Signal SIGCHLD sent to parent process when child terminates
  - SIGTERM and SIGKILL to terminate/kill processes
- System call kill can be used to send a signal from one process to other
  - Kill system call can send all signals, not just SIGKILL
  - Some restrictions on who can send to whom for isolation and security
- Signals can also be generated by OS for a process, e.g., when it handles interrupt due to Ctrl+C keyboard event
- Kill command to send signals, e.g., “kill -9 <pid>” sends SIGKILL (#9)

Process groups

- When we type Ctrl+C on keyboard, which processes get the signal?
- Processes are organized into process groups, every process belongs by default to process group of its parent
- When signal is sent to a process, it is delivered to all processes in its process group by default
- Example: when we hit Ctrl+C on keyboard, signal sent to all processes in the foreground process group
- System call setpgid can be used to change process group of signals, to control signal distribution

Signal handling

- Signals to a process are queued up by OS and delivered when process is running
- Default behavior defined by OS for a process receiving a signal
  - Ignore some signals (e.g., SIGCHLD)
  - Terminate when some signals are received (e.g., SIGINT)
- User processes can define their own signal handler functions to be executed when a signal is received
  - Override default behavior defined for a signal
  - Some signals (e.g., SIGKILL) cannot be overridden
- Process jumps to signal handler, executes it, resumes if still alive

Examples: sending and catching signals

- Parent sends SIGKILL to child using kill system call
- Child runs in infinite loop until killed by parent

```
int pid = fork()
if(pid == 0) {
  while(1); //infinite loop
  //terminates on SIGKILL
}
//parent
kill(pid, SIGKILL)
```

- Default SIGINT hander overridden
- Process prints message before terminating on SIGINT

```
void sigint_handler(int sig) {
  print “caught signal”
  exit()
}
int main() {
  signal(SIGINT, sigint_handler)
  …
}
```

Summary

Read a command, fork, exec, and wait
That's all there is to the life of the shell
Day after day, command prompt after prompt,
Like the monotony of Tom Hanks in “The Terminal”

The only excitement is in the minor skirmishes
When signals are thrown via the system call "kill"
And processes defend using signal handler shields
Or perish by bowing to the operating system's will