

Process Management

Goals of this Lecture

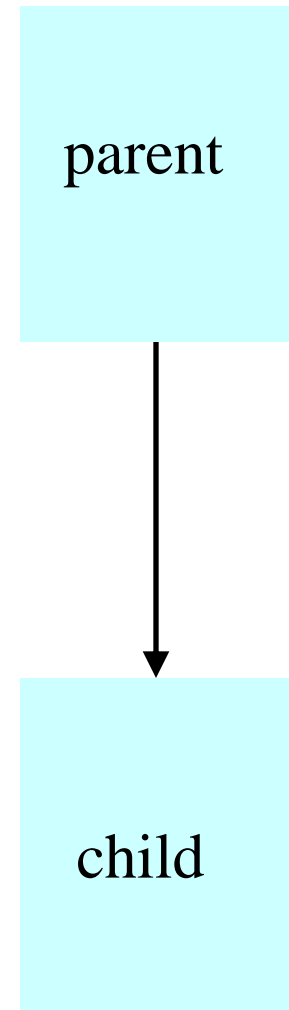
- Help you learn about:
 - Creating new processes
 - Programmatically redirecting stdin, stdout, and stderr
 - (Appendix) communication between processes via pipes
- Why?
 - Creating new processes and programmatic redirection are fundamental tasks of a Unix **shell** (see Assignment 6)
 - A power programmer knows about Unix shells, and thus about creating new processes and programmatic redirection

Why Create a New Process?

- Run a new program
 - E.g., shell executing a program entered at command line
 - Or, even running an entire pipeline of commands
 - Such as `wc -l * | sort | uniq -c | sort -nr`
- Run a new thread of control for the same program
 - E.g., a Web server handling a new Web request
 - While continuing to allow more requests to arrive
 - Essentially time sharing the computer
- Underlying mechanism
 - A process executes `fork()` to create a child process
 - (Optionally) child process does `exec()` of a new program

Creating a New Process

- Cloning an existing process
 - Parent process creates a new child process
 - The two processes then run concurrently
- Child process inherits state from parent
 - Identical (but separate) copy of virtual address space
 - Copy of the parent's open file descriptors
 - Parent and child share access to open files
- Child then runs independently
 - Executing independently, including invoking a new program
 - Reading and writing its own address space



Fork System-Level Function

- `fork()` is called once
 - But returns twice, once in each process
- Telling which process is which
 - Parent: `fork()` returns the child's process ID
 - Child: `fork()` returns 0

```
pid = fork();
if (pid != 0) {
    /* in parent */
    ...
} else {
    /* in child */
    ...
}
```

Fork and Process State

- Inherited
 - User and group IDs
 - Signal handling settings
 - Stdio
 - File pointers
 - Root directory
 - File mode creation mask
 - Resource limits
 - Controlling terminal
 - All machine register states
 - Control register(s)
 - ...
- Separate in child
 - Process ID
 - Address space (memory)
 - File descriptors
 - Parent process ID
 - Pending signals
 - Time signal reset times
 - ...

Example: What Output?

```
int main(void)
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid != 0) {
        printf("parent: x = %d\n", --x);
        exit(0);
    } else {
        printf("child: x = %d\n", ++x);
        exit(0);
    }
}
```

Executing a New Program

- `fork()` copies the state of the parent process
 - Child continues running the parent program
 - ... with a copy of the process memory and registers
- Need a way to invoke a new program
 - In the context of the newly-created child process
- Example
 - `program` (red text) Contains command-line arguments (to become "argv[]" of `ls`) (purple text)
 - `NULL-terminated array` (purple text)

```
execvp("ls", argv);  
fprintf(stderr, "exec failed\n");  
exit(EXIT_FAILURE);
```

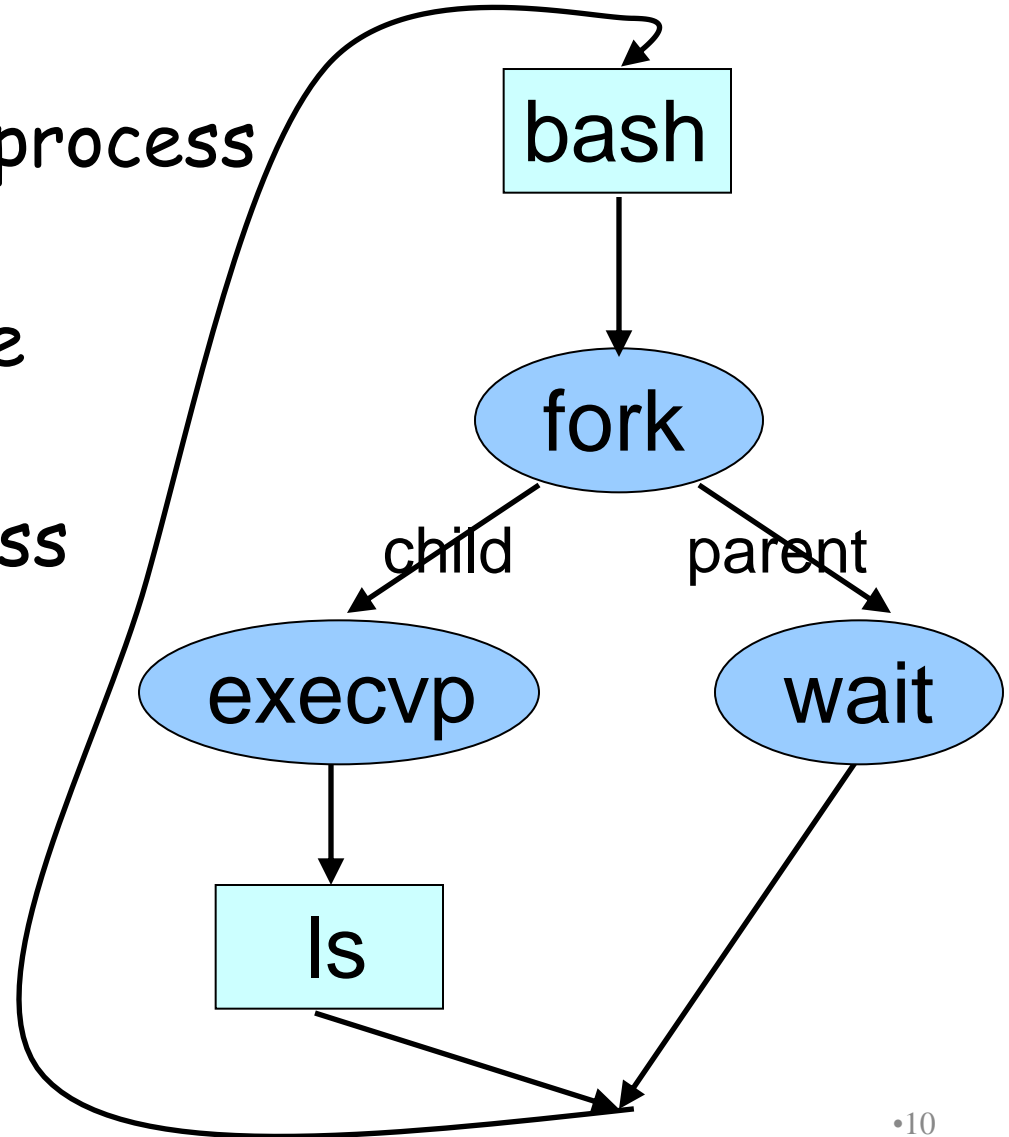

Waiting for the Child to Finish

- Parent should wait for children to finish
 - Example: a shell waiting for operations to complete
- Waiting for a child to terminate: `wait()`
 - Blocks until some child terminates
 - Returns the process ID of the child process
 - Or returns -1 if no children exist (i.e., already exited)
- Waiting for specific child to terminate: `waitpid()`
 - Blocks till a child with particular process ID terminates

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Example: A Simple Shell

- Shell is the parent process
 - E.g., bash
- Parses command line
 - E.g., "ls -l"
- Invokes child process
 - `fork()`, `execvp()`
- Waits for child
 - `wait()`



Simple Shell Code

```
Parse command line  
Assign values to somepgm, someargv  
pid = fork();  
if (pid == 0) {  
    /* in child */  
    execvp(somepgm, someargv);  
    fprintf(stderr, "exec failed\n");  
    exit(EXIT_FAILURE);  
}  
/* in parent */  
pid = wait(&status);  
Repeat the previous
```

Simple Shell Trace (1)

Parent Process

```
Parse command line  
Assign values to somepgm, someargv  
pid = fork();  
if (pid == 0) {  
    /* in child */  
    execvp(somepgm, someargv);  
    fprintf(stderr, "exec failed\n");  
    exit(EXIT_FAILURE);  
}  
/* in parent */  
pid = wait(&status);  
Repeat the previous
```

Parent reads and parses command line

Parent assigns values to **somepgm** and **someargv**

Simple Shell Trace (2)

Parent Process

```
Parse command line  
Assign values to somepgm, someargv  
pid = fork();  
if (pid == 0) {  
    /* in child */  
    execvp(somepgm, someargv);  
    fprintf(stderr, "exec failed\n");  
    exit(EXIT_FAILURE);  
}  
/* in parent */  
pid = wait(&status);  
Repeat the previous
```

executing
concurrently

Child Process

```
Parse command line  
Assign values to somefile, someargv  
pid = fork();  
if (pid == 0) {  
    /* in child */  
    execvp(somepgm, someargv);  
    fprintf(stderr, "exec failed\n");  
    exit(EXIT_FAILURE);  
}  
/* in parent */  
pid = wait(&status);  
Repeat the previous
```

fork() creates child process

Which process gets the CPU first? Let's assume the parent...

Simple Shell Trace (3)

Parent Process

child's pid

Child Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

executing
concurrently

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

In parent, pid != 0; parent waits; OS gives CPU to child

Simple Shell Trace (4)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

Child Process

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
    /* in child */
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
Repeat the previous
```

0
executing
concurrently

In child, pid == 0; child calls `execvp()`

Simple Shell Trace (5)

Parent Process

Child Process

```
Parse command line  
Assign values to somepgm, someargv  
pid = fork();  
if (pid == 0) {  
    /* in child */  
    execvp(somepgm, someargv);  
    fprintf(stderr, "exec failed\n");  
    exit(EXIT_FAILURE);  
}  
/* in parent */  
pid = wait(&status);  
Repeat the previous
```

executing
concurrently

somepgm

In child, somepgm overwrites shell program;
main() is called with someargv as argv parameter

Simple Shell Trace (6)

Parent Process

```
Parse command line  
Assign values to somepgm, someargv  
pid = fork();  
if (pid == 0) {  
    /* in child */  
    execvp(somepgm, someargv);  
    fprintf(stderr, "exec failed\n");  
    exit(EXIT_FAILURE);  
}  
/* in parent */  
pid = wait(&status);  
Repeat the previous
```

executing
concurrently

Child Process

~~somepgm~~

Somepgm executes in child, and eventually exits

Simple Shell Trace (7)

Parent Process

```
Parse command line  
Assign values to somepgm, someargv  
pid = fork();  
if (pid == 0) {  
    /* in child */  
    execvp(somepgm, someargv);  
    fprintf(stderr, "exec failed\n");  
    exit(EXIT_FAILURE);  
}  
/* in parent */  
pid = wait(&status);  
Repeat the previous
```

Parent returns from `wait()` and proceeds

Combined Fork/Exec/Wait

- Common combination of operations
 - `fork()` to create a new child process
 - `exec()` to invoke new program in child process
 - `wait()` in the parent process for the child to complete
- Single call that combines all three
 - `int system(const char *cmd);`
- Example

```
int main(void) {  
    system("echo Hello world");  
    return 0;  
}
```

Fork and Virtual Memory

- Incidentally...
- Question:
 - `fork()` duplicates an entire process (text, bss, data, rodata, stack, heap sections)
 - Isn't that *very* inefficient???!??!
- Answer:
 - Using virtual memory, not really!
 - Upon `fork()`, OS creates virtual pages for child process
 - Each child virtual page points to real page (in memory or on disk) of parent
 - OS duplicates real pages incrementally, and only if/when "write" occurs

Redirection

- Unix allows programmatic redirection of stdin, stdout, or stderr
- How?
 - Use `open()`, `creat()`, and `close()` system calls
 - Described in **I/O Management** lecture
 - Use `dup()` system call...

```
int dup(int oldfd);
```

- Create a copy of the file descriptor `oldfd`. After a successful return from `dup()` or `dup2()`, the old and new file descriptors may be used interchangeably. They refer to the same open file description and thus share file offset and file status flags. Uses the lowest-numbered unused descriptor for the new descriptor. Return the new descriptor, or -1 if an error occurred.

Redirection Example

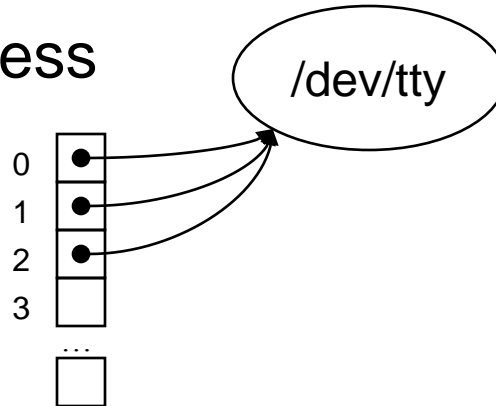
How does shell implement “somepgm > somefile”?

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Redirection Example Trace (1)

Parent Process

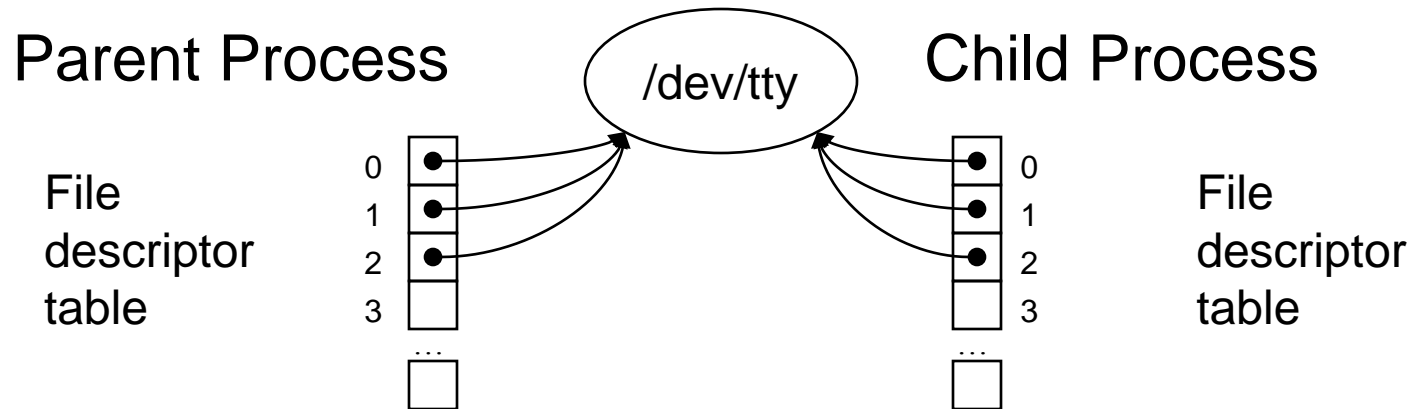
File
descriptor
table



```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Parent has file descriptor table; first three point to “terminal”

Redirection Example Trace (2)

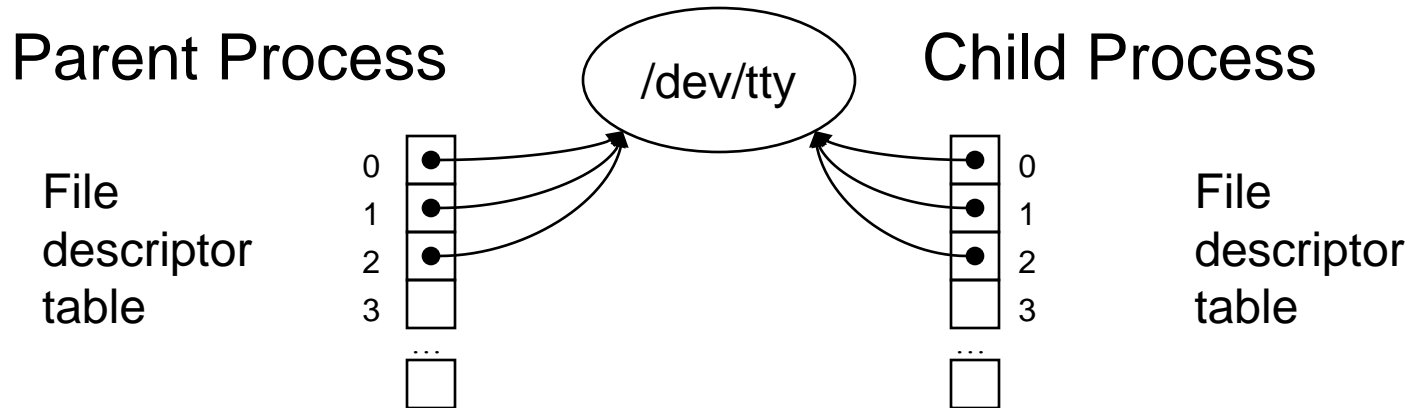


```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Parent forks child; child has identical file descriptor table 24

Redirection Example Trace (3)

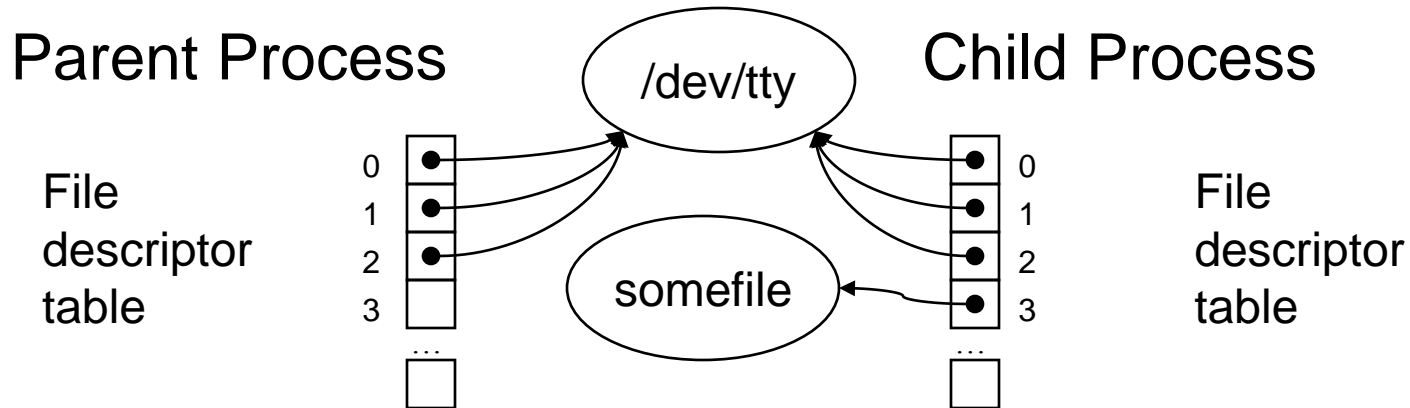


```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Let's say parent gets CPU first; parent waits

Redirection Example Trace (4)



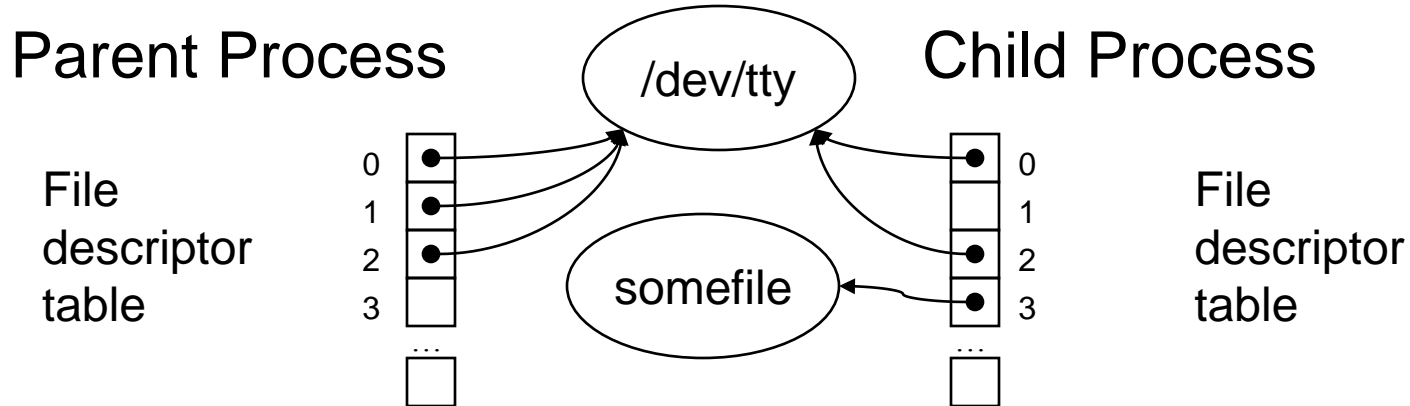
```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

3

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child gets CPU; child creates somefile

Redirection Example Trace (5)



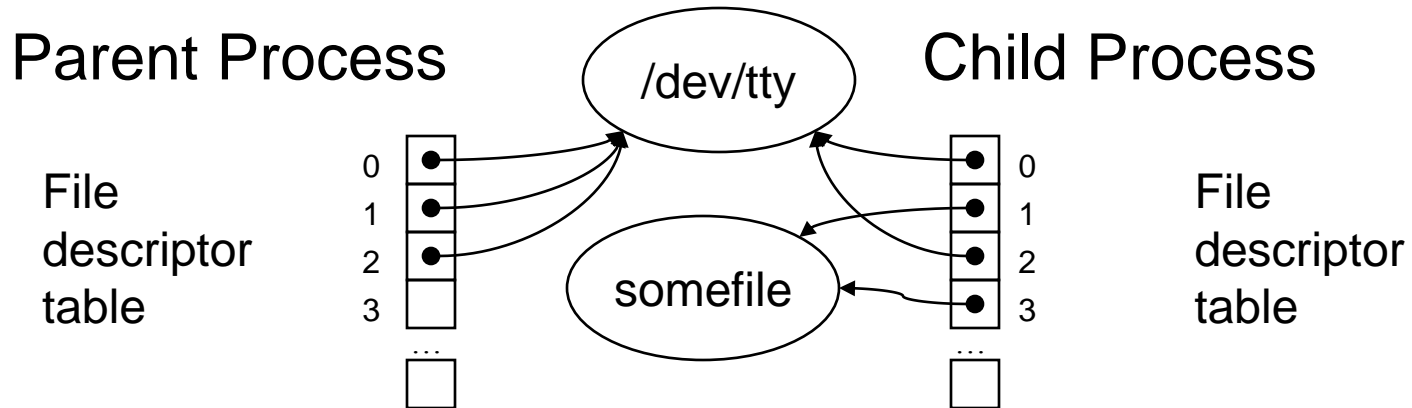
```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

3

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child closes file descriptor 1 (stdout)

Redirection Example Trace (6)



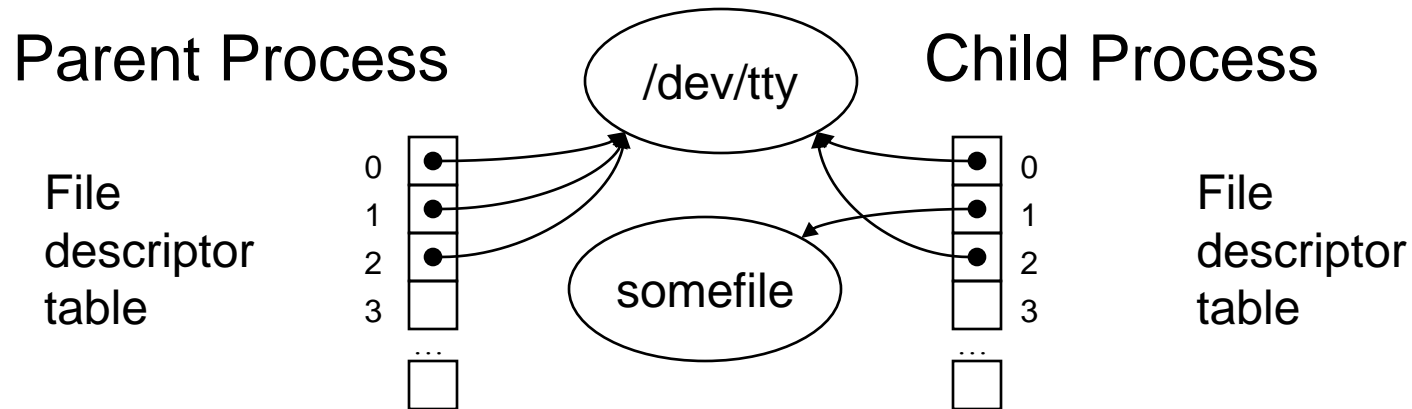
```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

3

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child duplicates file descriptor 3 into first unused spot

Redirection Example Trace (7)



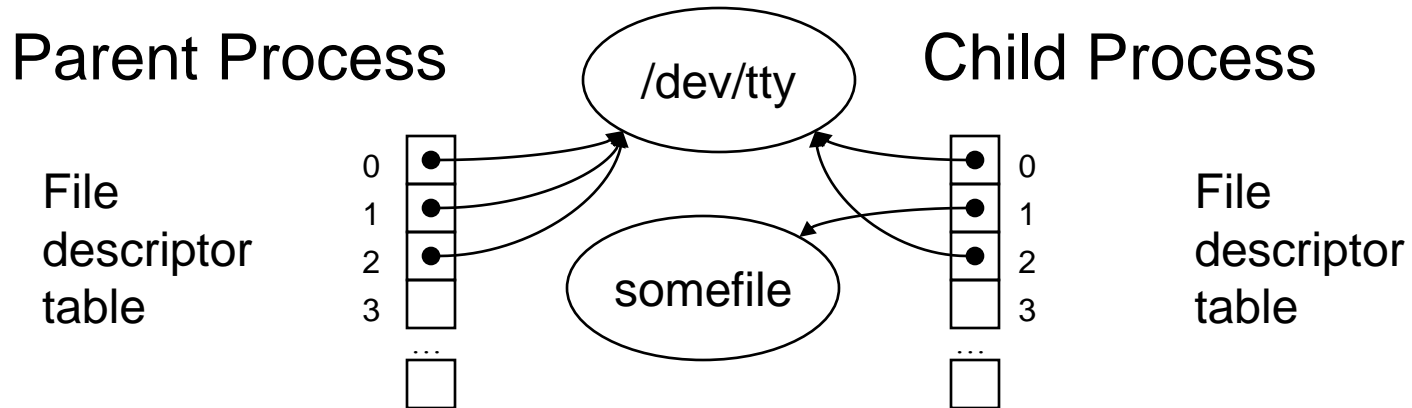
```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

3

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child closes file descriptor 3

Redirection Example Trace (8)



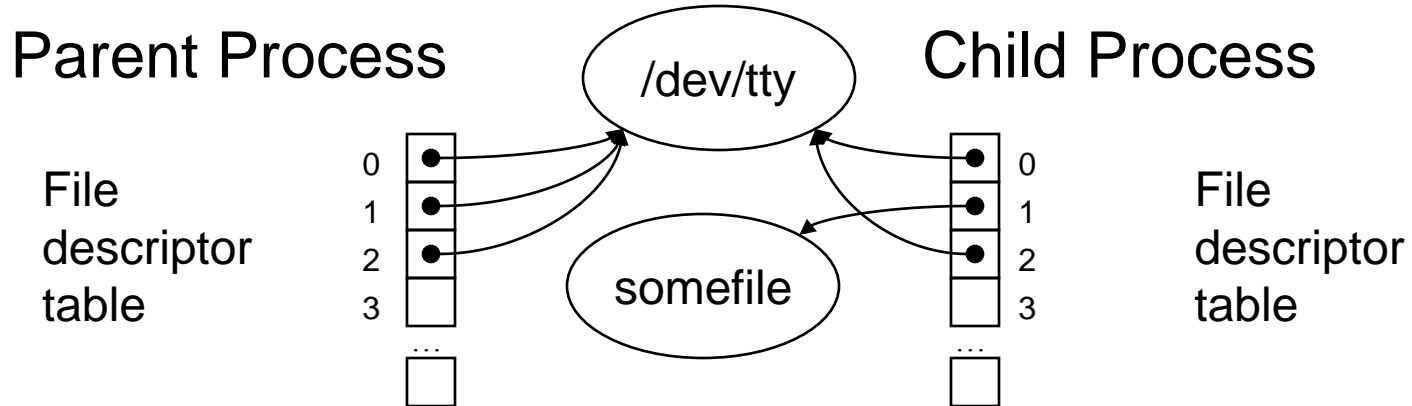
```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child calls execvp()

3

```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepgm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Redirection Example Trace (9)



```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somepfm, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

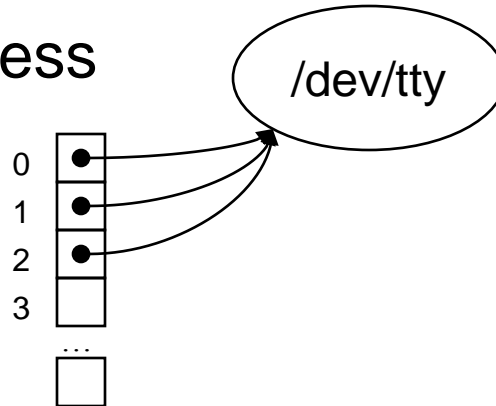
```
somepgm
```

Somepgm executes with stdout redirected to somefile

Redirection Example Trace (10)

Parent Process

File
descriptor
table



```
pid = fork();
if (pid == 0) {
    /* in child */
    fd = creat("somefile", 0640);
    close(1);
    dup(fd);
    close(fd);
    execvp(somefile, someargv);
    fprintf(stderr, "exec failed\n");
    exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Somepgm exits; parent returns from `wait()` and proceeds

The Beginnings of a Unix Shell

- A shell is mostly a big loop
 - Parse command line from stdin
 - Expand wildcards ('*')
 - Interpret redirections ('<', and '>')
 - `fork()`, `dup()`, `exec()`, and `wait()`, as necessary
- Start from the code in earlier slides
 - And edit till it becomes a Unix shell
 - This is the heart of the last programming assignment

Summary

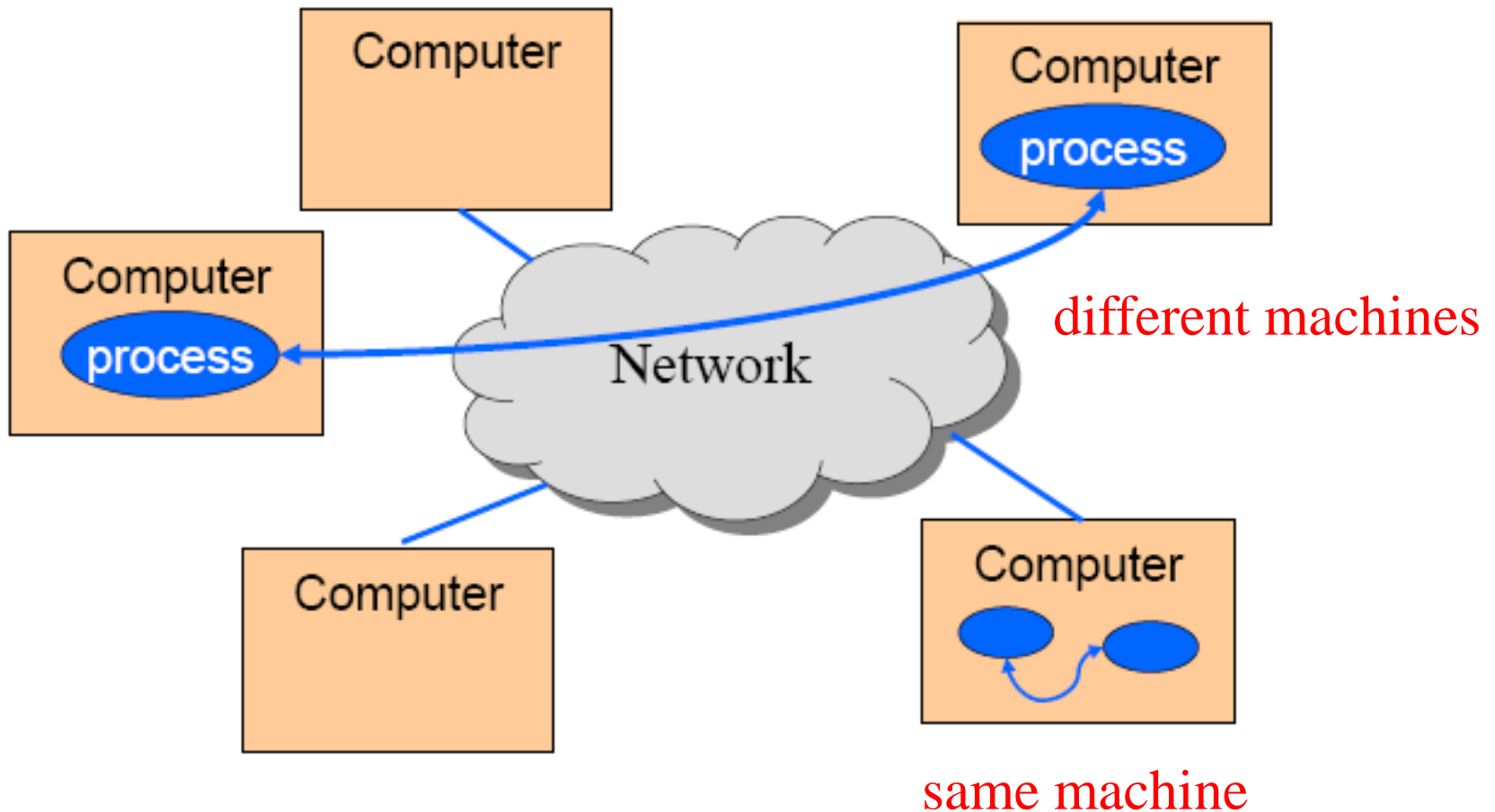
- System-level functions for creating processes
 - `fork()`: process creates a new child process
 - `wait()`: parent waits for child process to complete
 - `exec()`: child starts running a new program
 - `system()`: combines `fork`, `wait`, and `exec` all in one
- System-level functions for redirection
 - `open()` / `creat()`: to open a file descriptor
 - `close()`: to close a file descriptor
 - `dup()`: to duplicate a file descriptor

Appendix

Inter-Process Communication (IPC)

IPC

- Mechanism by which two processes exchange information and coordinate activities

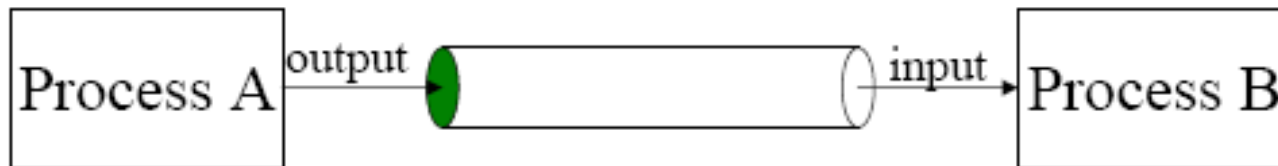


IPC Mechanisms

- Pipes
 - Processes on the same machine
 - Allows parent process to communicate with child process
 - Allows two “sibling” processes to communicate
 - Used mostly for a pipeline of filters
 - Sockets
 - Processes on any machines
 - Processes created independently
 - Used for client/server communication (e.g., Web)
- Both provide abstraction of an “ordered stream of bytes”

Pipes

- Provides an interprocess communication channel



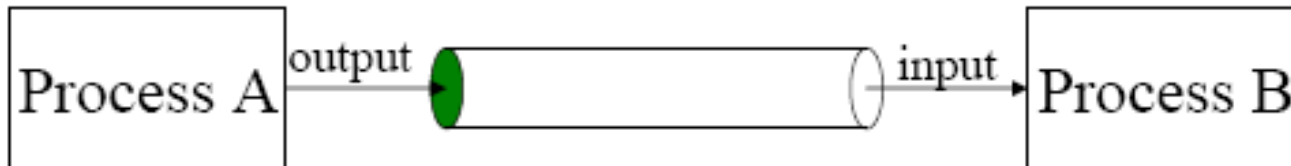
- A filter is a process that reads from `stdin` and writes to `stdout`



Example Use of Pipes

- Compute a histogram of content types in my e-mail
 - Many e-mail messages, consisting of many lines
 - Lines like "Content-Type: image/jpeg" indicate the type
- Pipeline of Unix commands
 - Identifying content type: `grep -i Content-Type *`
 - Extracting just the type: `cut -d" " -f2`
 - Sorting the list of types: `sort`
 - Counting the unique types: `uniq -c`
 - Sorting the counts: `sort -nr`

Creating a Pipe



- Pipe is a communication channel abstraction
 - Process A can write to one end using “write” system call
 - Process B can read from the other end using “read” system call
- System call

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

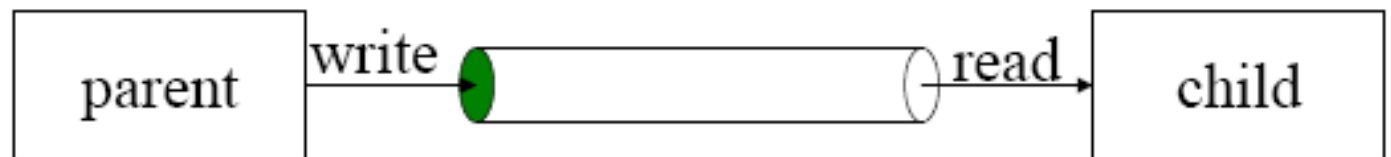
return 0 upon success -1 upon failure
fd[0] is open for reading
fd[1] is open for writing
- Two coordinated processes created by `fork` can pass data to each other using a pipe.

Pipe Example

```
int pid, p[2];
...
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    ... read using p[0] as fd until EOF ...
}
else {
    close(p[0]);
    ... write using p[1] as fd ...
    close(p[1]); /* sends EOF to reader */
    wait(&status);
}
}
```

child

parent

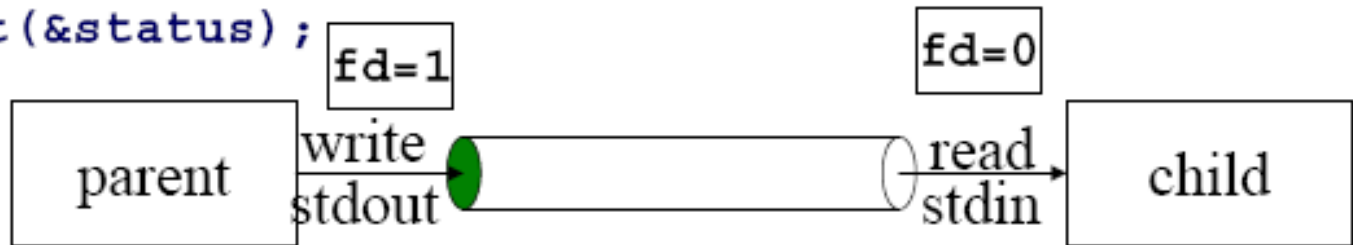


Pipes and Stdio

```
int pid, p[2];
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    dup2(p[0], 0);
    close(p[0]);
    ... read from stdin ...
}
else {
    close(p[0]);
    dup2(p[1], 1);
    close(p[1]);
    ... write to stdout ...
    wait(&status);
}
```

child makes stdin (0)
the read side of the pipe

parent makes stdout (1)
the write side of the pipe



Pipes and Exec

```
int pid, p[2];
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    dup2(p[0], 0);
    close(p[0]);
    execl(...);
}
else {
    close(p[0]);
    dup2(p[1], 1);
    close(p[1]);
    ... write to stdout ...
    wait(&status);
}
```

child process

invokes a new program

