Processes and Pipes

Shell Assignment

Review and Discuss

- Creating a new process
  - 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
- Communication between processes
  - Pipe between two processes
  - Redirecting stdin and stdout
- Initial background for the shell assignment
  - Software that provides interface for the user
  - Primarily used to launch other programs

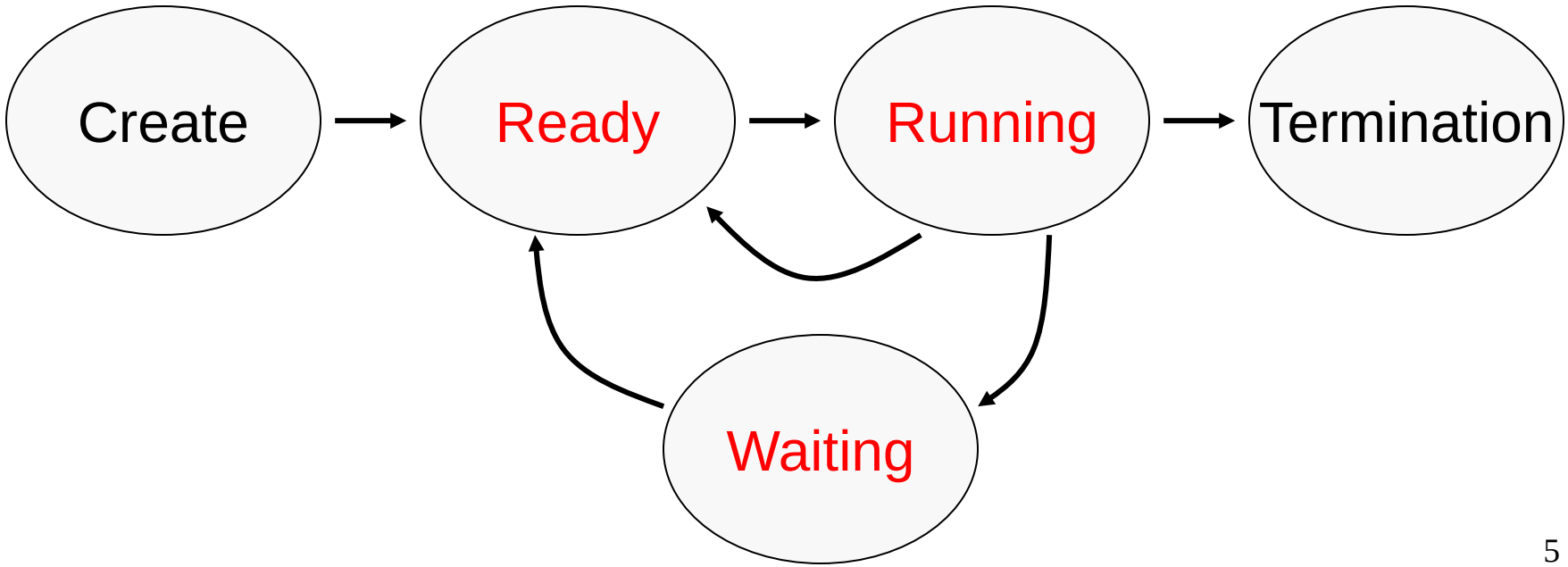
Creating a New Process

Program vs. Process

- Program
  - Executable code
  - No dynamic state
- Process
  - An instance of a program in execution
  - With its own control flow (illusion of a processor)
  - … & private address space (illusion of memory)
  - State including code, data, stack, registers, instruction pointer, open file descriptors, …
  - Either running, waiting, or ready…
- Can run multiple instances of the same program
  - Each as its own process, with its own process ID

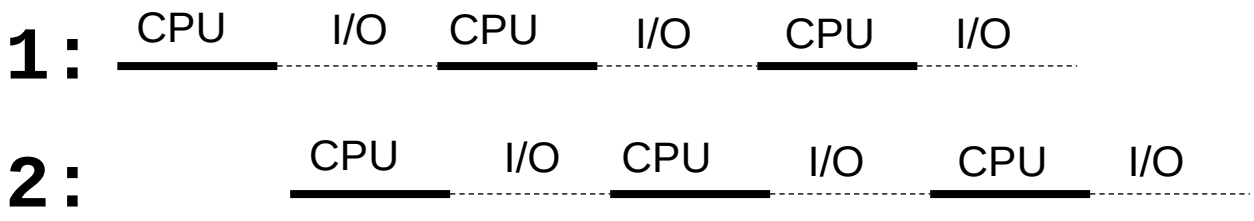
Life Cycle of a Process

- Running: instructions are being executed
- Waiting: waiting for some event (e.g., I/O finish)
- Ready: ready to be assigned to a processor



Many Processes Running “Concurrently”

- Multiple processes sharing the CPU



- Processor switches context between the two
  - When process blocks waiting for operation to complete
  - When process finishing using its share of the CPU
- But, how do multiple processes start running
  - How are they invoked in the first place?

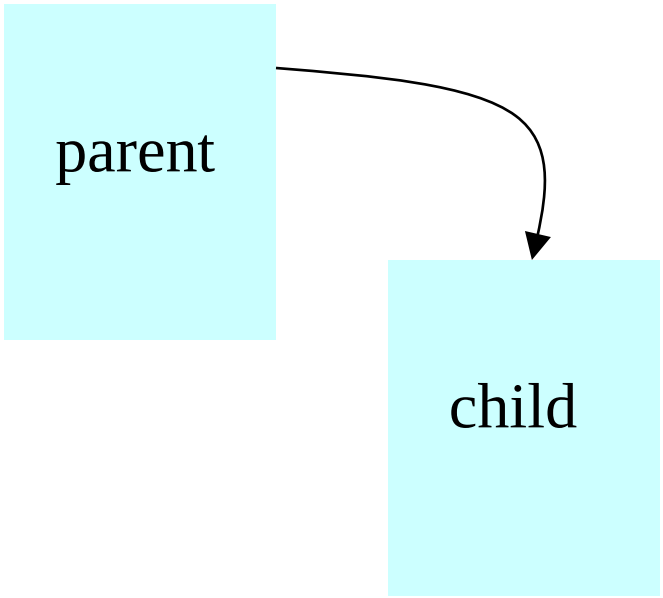
Why Start a New Process?

- 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 runs "fork" to create a child process
  - (Optionally) child process does "exec" of a new program

Fork System Call

- Create a new process
  - Child process inherits state from parent process
  - Parent and child have separate copies of that state
  - Parent and child share access to any open files

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



Fork System Call

- 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 a 0

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

Example: What Output?

```
int main()
{
    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);
    }
}
```

Fork

- Inherited:
  - user and group IDs
  - signal handling settings
  - stdio
  - file pointers
  - current working directory
  - 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
  - timer signal reset times
  - . . .

Wait

- Parent waits for a child (system call)
  - blocks until a child terminates
  - returns pid of the child process
  - returns –1 if no children exists (already exited)
  - status

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

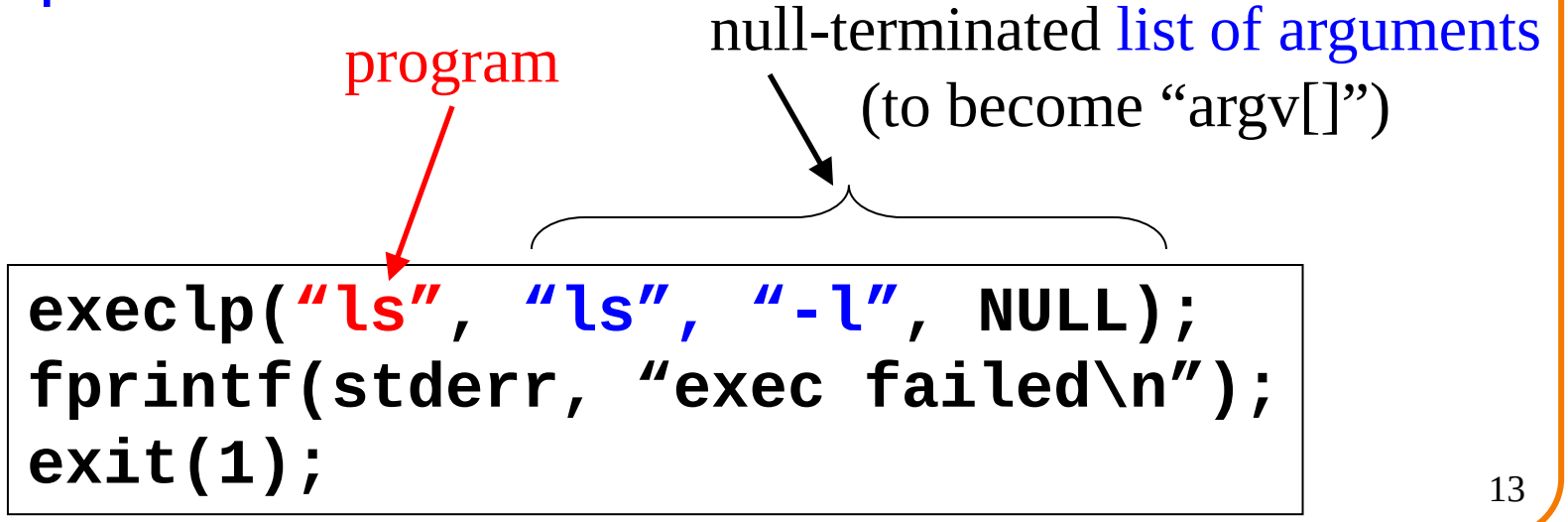
- Parent waits for a specific child to terminate

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

Executing a New Program

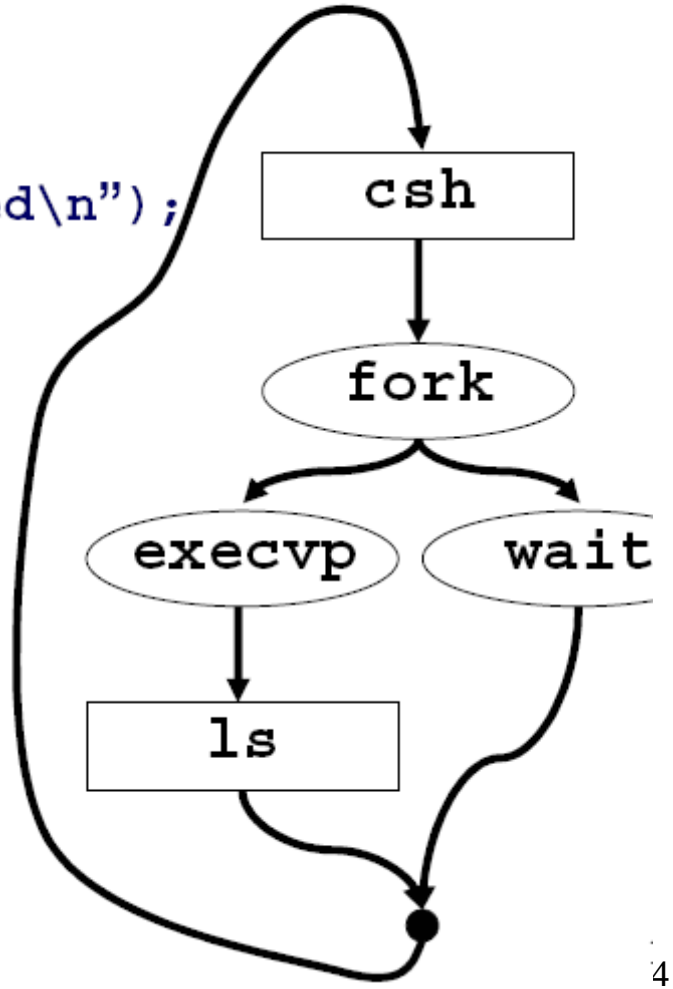
- Fork copies the state of the parent process
  - Child continues running the parent program
  - … with a copy of the process memory and registers
- Need a way to invoke a new program
  - In the context of the newly-created child process
- Example



Combining Fork() and Exec()

- Commonly used together by the shell

```
... parse command line ...
pid = fork()
if (pid == -1)
    fprintf(stderr, "fork failed\n");
else if (pid == 0) {
    /* in child */
    execvp(file, argv);
    fprintf(stderr,
            "exec failed\n");
} else {
    /* in parent */
    pid = wait(&status);
}
... return to top of loop ...
```



System

- Convenient way to invoke fork/exec/wait
  - Forks new process
  - Execs command
  - Waits until it is complete

```
int system(const char *cmd);
```

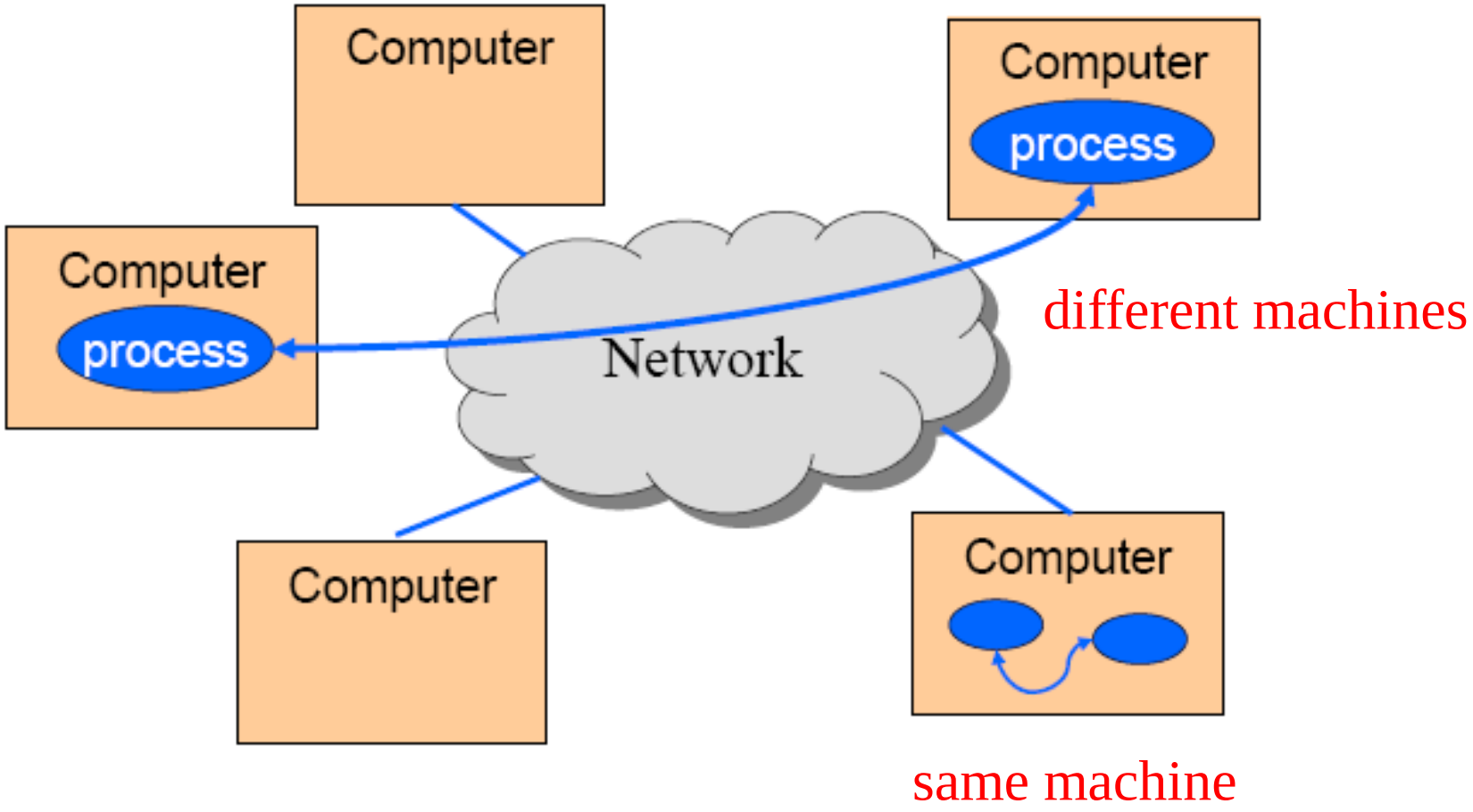
- Example:

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

Communication Between Processes

Communication Between Processes

- Mechanism by which two processes exchange information and coordinate activities

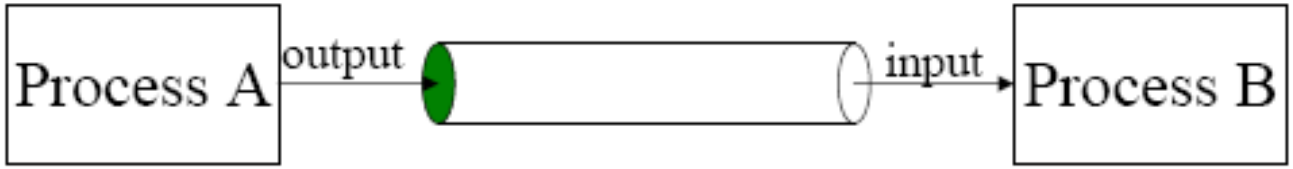


Interprocess Communication

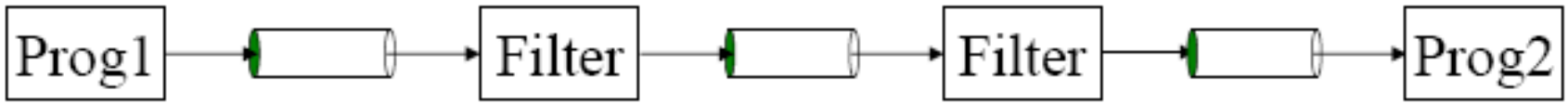
- Pipes
  - Processes on the same machine
  - One process spawns the other
  - Used mostly for a pipeline of filters
- Sockets
  - Processes on any machines
  - Processes created independently
  - Used for client/server communication (e.g., Web)

Pipes

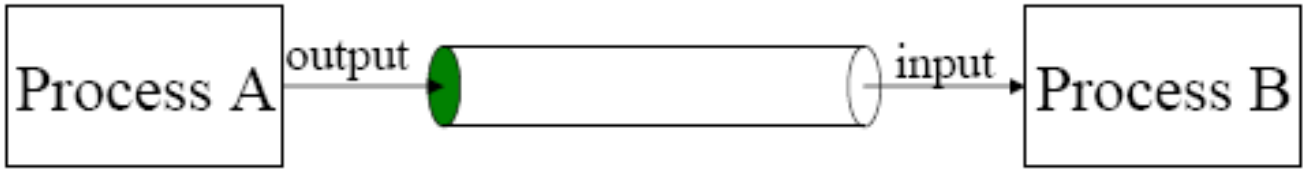
- Provides an interprocess communication channel



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



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
```

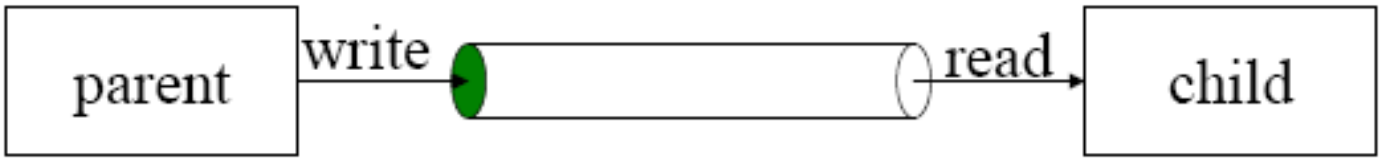
- Two coordinated processes created by `fork` can pass data to each other using a pipe.

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



Dup

- Duplicate a file descriptor (system call)

```
int dup( int fd );
```

duplicates `fd` as the lowest unallocated descriptor

- Commonly used to implement redirection of stdin/stdout
- Example: redirect stdin to "foo"


```
int fd;
fd = open("foo", O_RDONLY, 0);
close(0);
dup(fd);
close(fd);
```

Dup2

- For convenience…

```
dup2( int fd1, int fd2 );
```

use `fd2` (`new`) to duplicate `fd1` (`old`)
closes `fd2` if it was in use

- Example: redirect stdin to “foo”

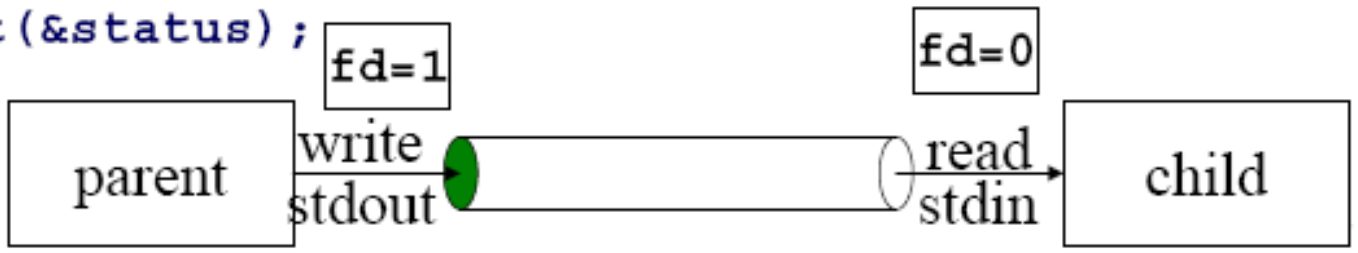
```
fd = open(“foo”, O_RDONLY, 0);
dup2(fd,0);
close(fd);
```

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

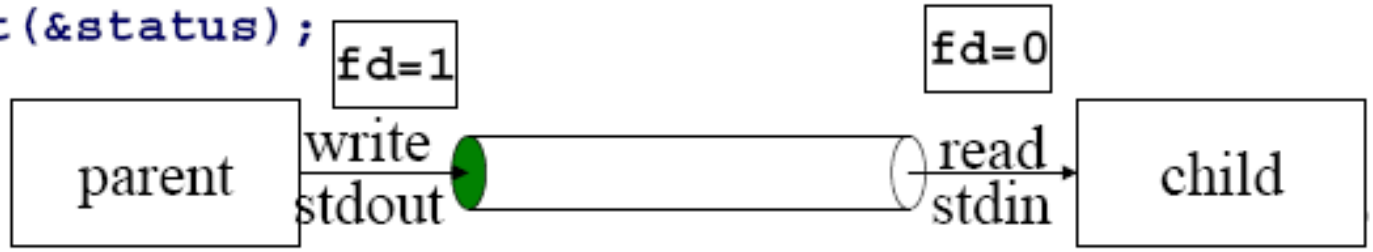
parent



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



A Unix Shell!

- Loop
  - Read command line from stdin
  - Expand wildcards
  - Interpret redirections < > |
  - pipe (as necessary), fork, dup, exec, wait
- Start from code on previous slides, edit it until it's a Unix shell!

Conclusion

- System calls
  - An interface to the operating system
  - To perform operations on behalf of a user process
- System calls 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 calls for inter-process communication
  - Pipe: create a pipe with a write end and a read end
  - Open/close: to open or close a file
  - Dup2: to duplicate a file descriptor