



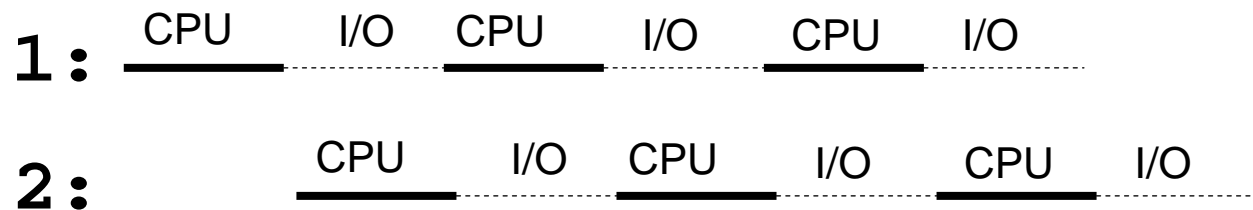
Processes and Pipes

COS 217
Prof. David August

When to Change Which Process is Running?



- When a process is stalled waiting for I/O
 - Better utilize the CPU, e.g., while waiting for disk access

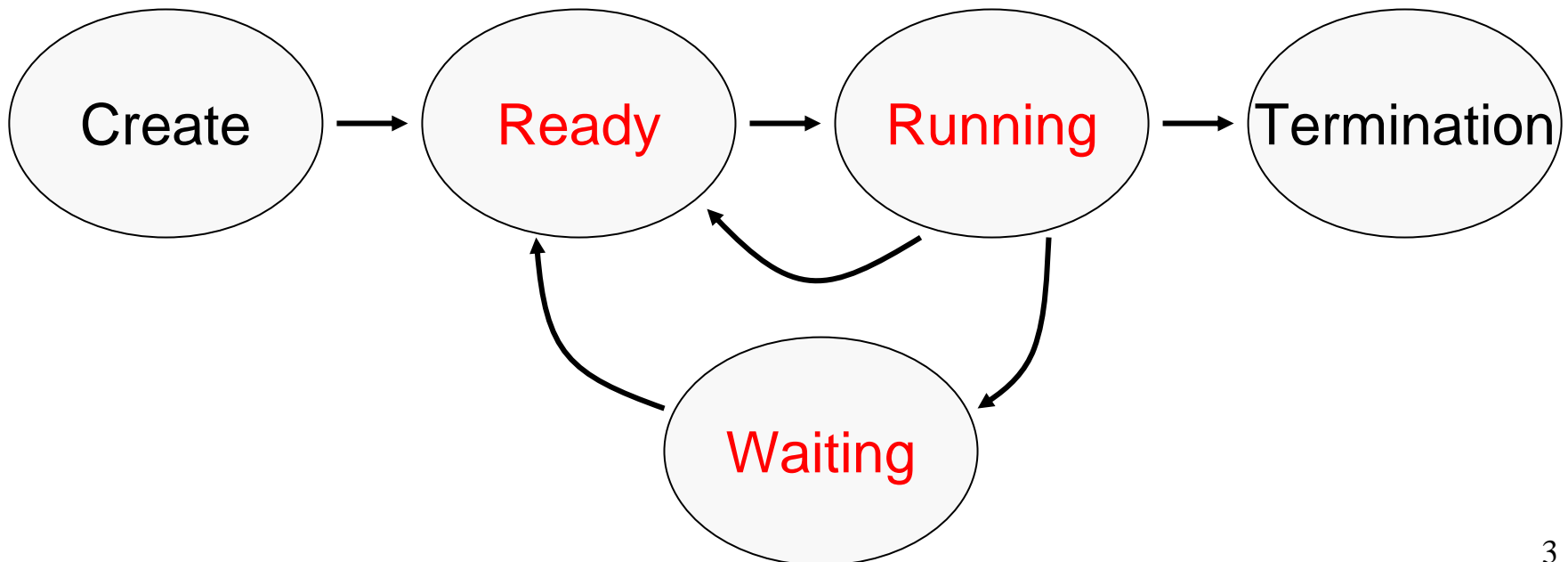


- When a process has been running for a while
 - Sharing on a fine time scale to give each process the illusion of running on its own machine
 - Trade-off efficiency for a finer granularity of fairness

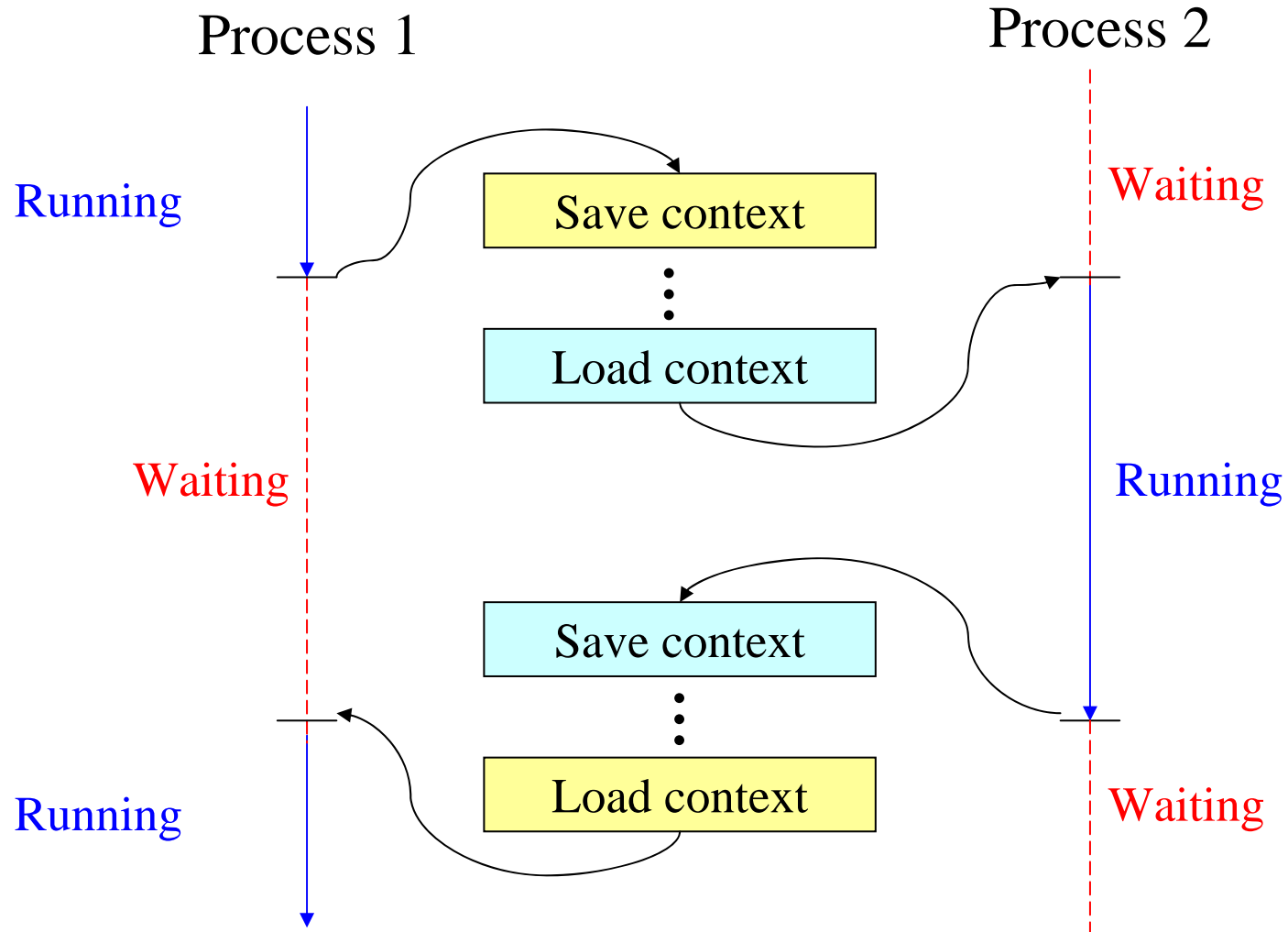


Life Cycle of a Process

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



Switching Between Processes

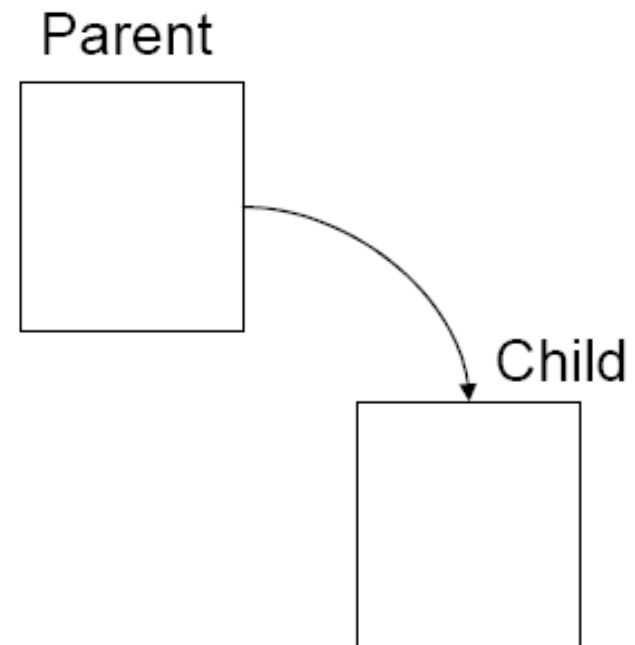




Fork

- Create a new process (system call)
 - 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



- 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);
```



Exec

- Overlay current process image with a specified image file (system call)
 - affects process memory and registers
 - has no affect on file table

- Example:

```
execlp("ls", "ls", "-l", NULL);  
fprintf(stderr, "exec failed\n");  
exit(1);
```




Fork/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");
```

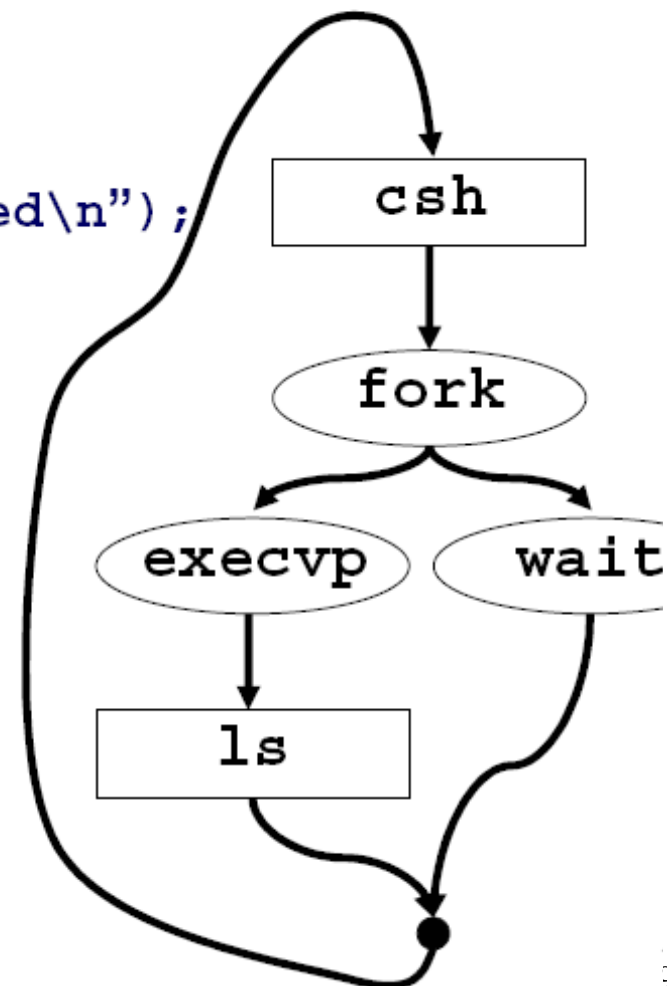
```
} else {
```

```
    /* 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

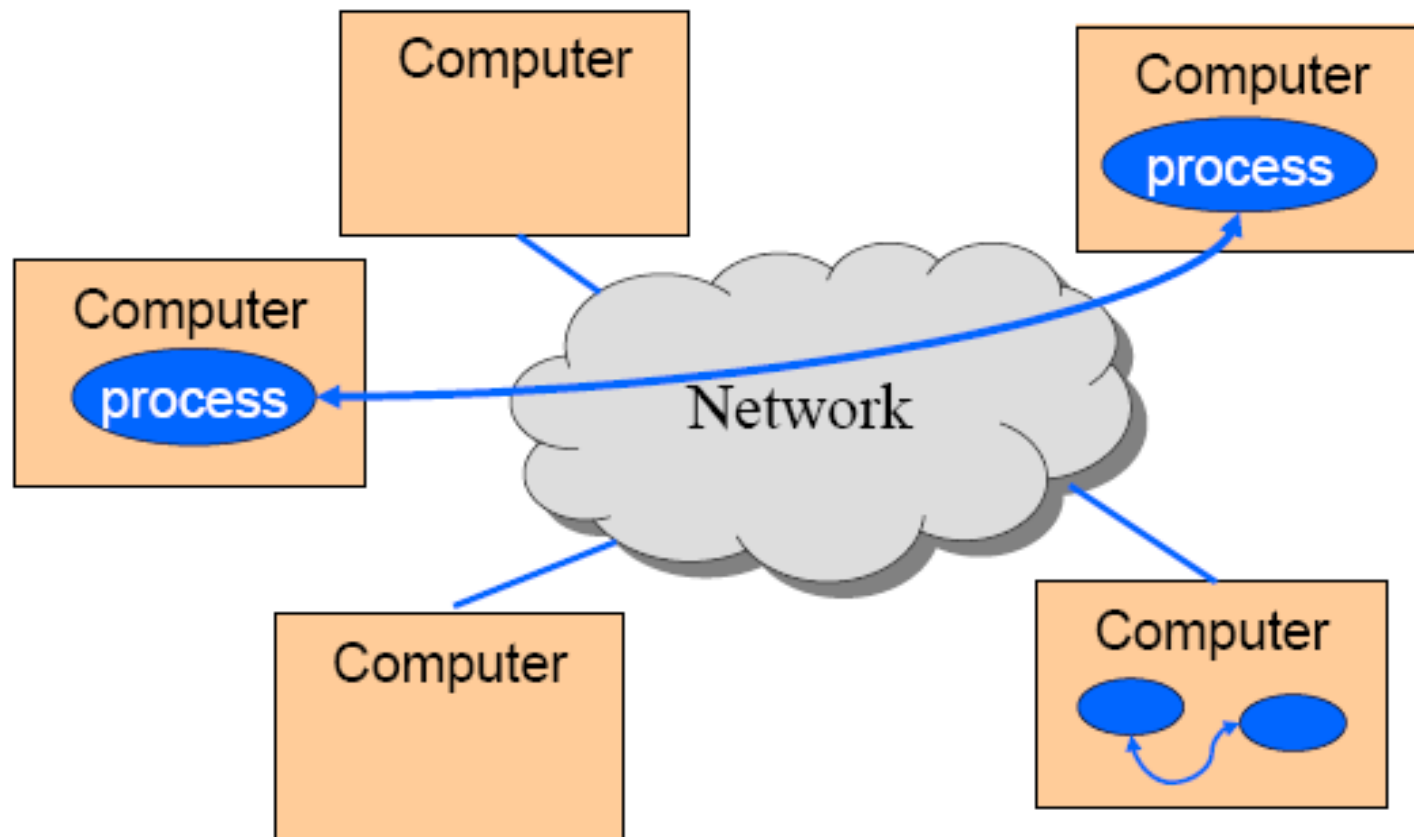
```
int system(const char *cmd);
```

- Example:

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

Networks

- Mechanism by which two processes exchange information and coordinate activities





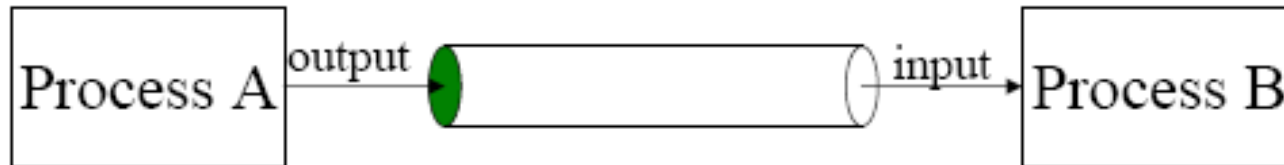
Interprocess Communication

- Pipes
 - Processes must be on same machine
 - One process spawns the other
 - Used mostly for filters
- Sockets
 - Processes can be on any machine
 - Processes can be created independently
 - Used for clients/servers, distributed systems, etc.



Pipes

- Provides an interprocess communication channel

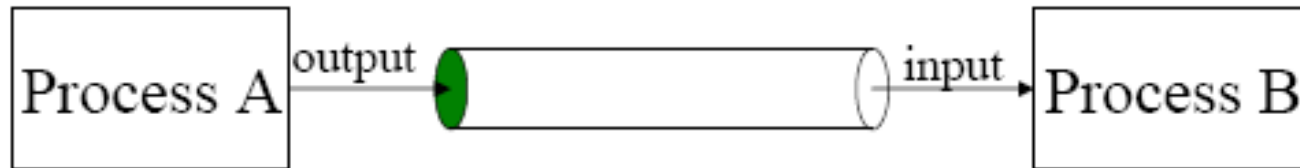


- A filter is a process that reads from `stdin` and writes to `stdout`





Creating a Pipe



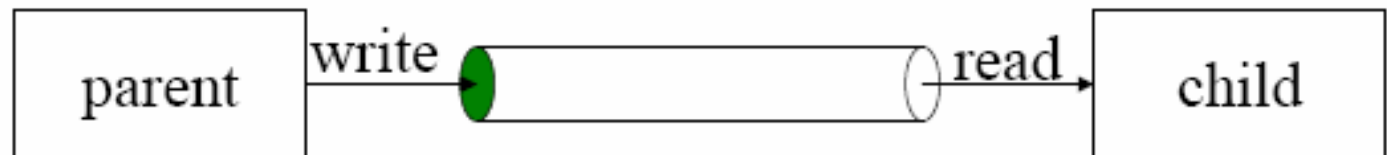
- Pipe is a communication channel abstraction
 - Process A can write to one end using “write” system call
 - Process B can read from the other end using “read” system call
- System call

```
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);
}
```





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);
```

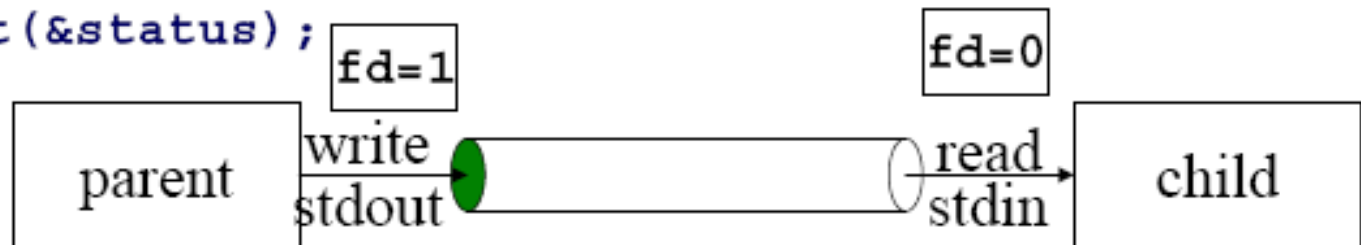
```
dup2( fd, 0 );
```

```
close( fd );
```



Pipes and Stdio

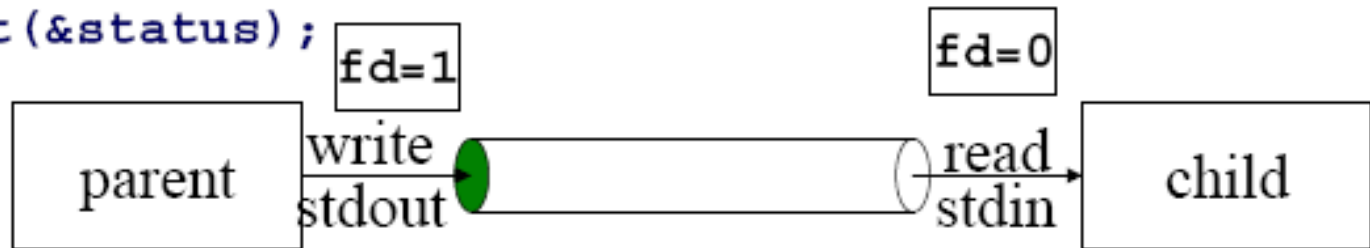
```
int pid, p[2];
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    dup2(p[0], 0);
    close(p[0]);
    ... read from stdin ...
}
else {
    close(p[0]);
    dup2(p[1], 1);
    close(p[1]);
    ... write to stdout ...
    wait(&status);
}
```





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);
}
```





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!