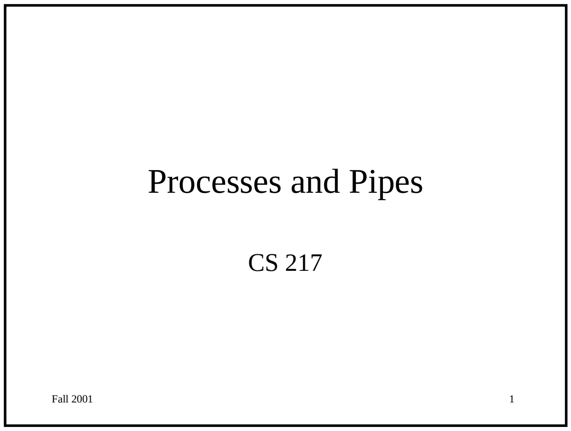
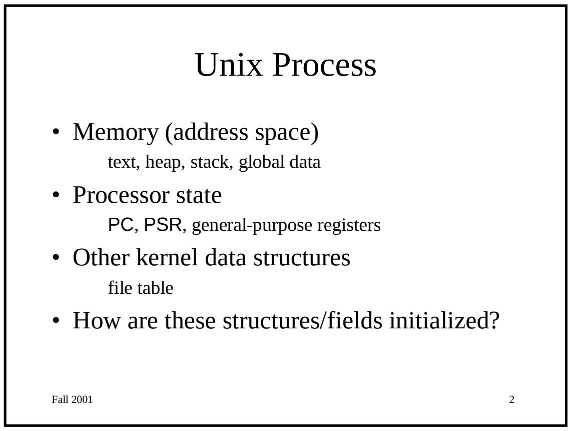


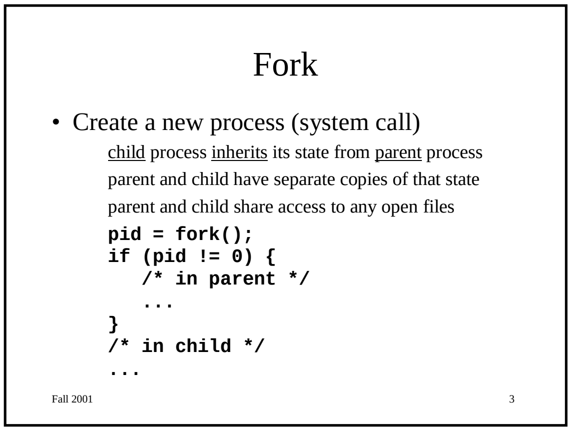
Processes and Pipes

CS 217



Unix Process

- Memory (address space)
  - text, heap, stack, global data
- Processor state
  - PC, PSR, general-purpose registers
- Other kernel data structures
  - file table
- How are these structures/fields initialized?



Fork

- Create a new process (system call)
  - child process inherits its 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 */
   ...
}
/* in child */
...
```

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);
```

Exec (cont)

- Many variations of **`exec`**

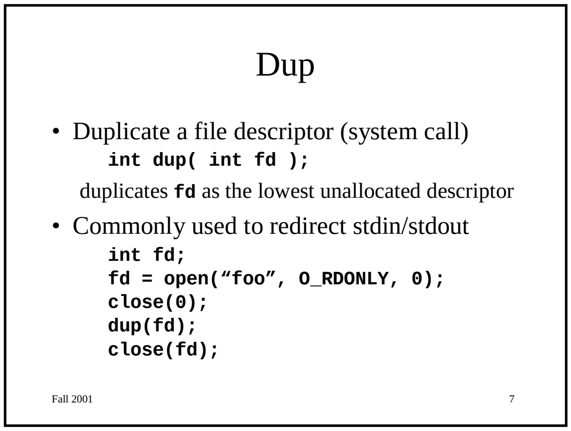
```
int execlp(const char *file,
           const char *arg, ...)
int execl(const char *path,
           const char *arg, ...)
int execv(const char *path,
           char * const argv[])
int execle(const char *path,
           const char *arg, ...,
           char * const envp[])
```

Also **`execve`** and **`execvp`**

Fork/Exec

- Commonly used together by the shell

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



Dup

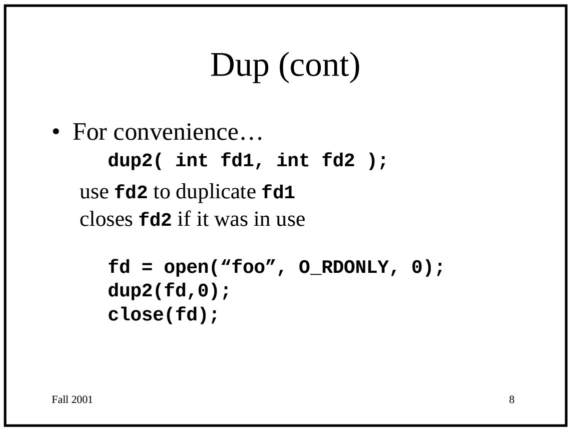
- Duplicate a file descriptor (system call)

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

duplicates **fd** as the lowest unallocated descriptor

- Commonly used to redirect stdin/stdout

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



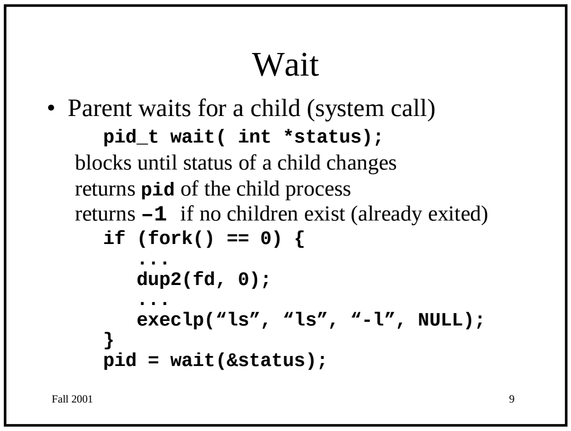
Dup (cont)

- For convenience…

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

use **fd2** to duplicate **fd1**
closes **fd2** if it was in use

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



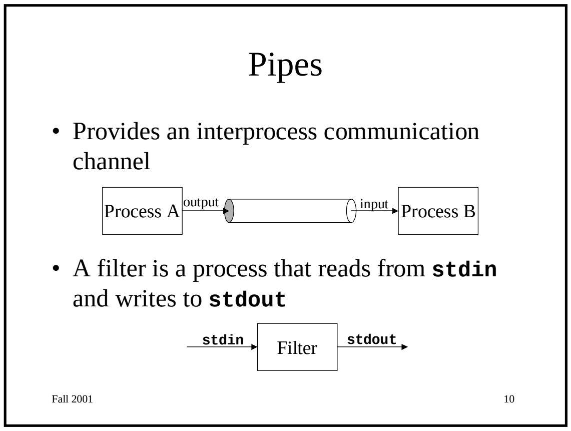
Wait

- Parent waits for a child (system call)

```
pid_t wait( int *status);
```

blocks until status of a child changes
returns **pid** of the child process
returns **–1** if no children exist (already exited)

```
if (fork() == 0) {
   ...
   dup2(fd, 0);
   ...
   execlp(“ls”, “ls”, “-l”, NULL);
}
pid = wait(&status);
```



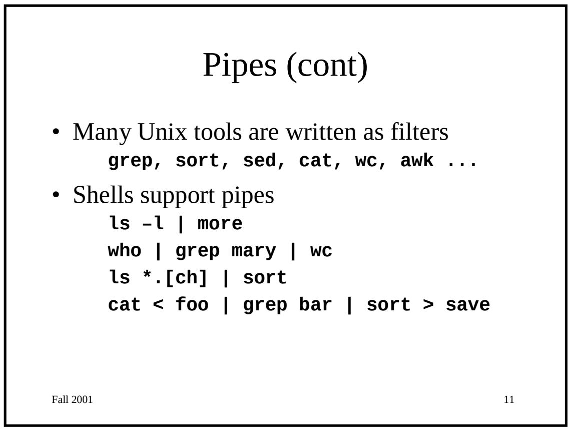
Pipes

- Provides an interprocess communication channel

Process A
output
input
Process B

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

stdin
Filter
stdout



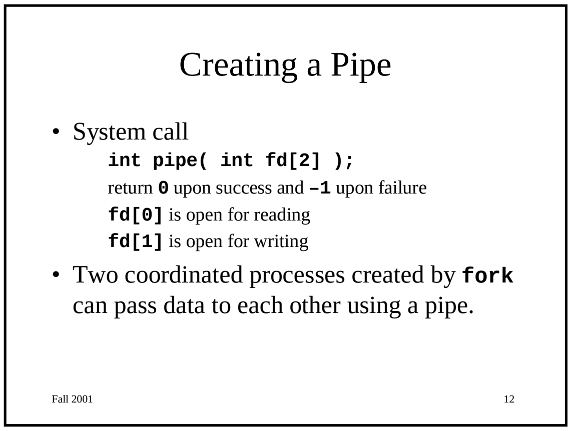
Pipes (cont)

- Many Unix tools are written as filters

```
grep, sort, sed, cat, wc, awk ...
```

- Shells support pipes

```
ls –l | more
who | grep mary | wc
ls *.[ch] | sort
cat < foo | grep bar | sort > save
```



Creating a Pipe

- System call

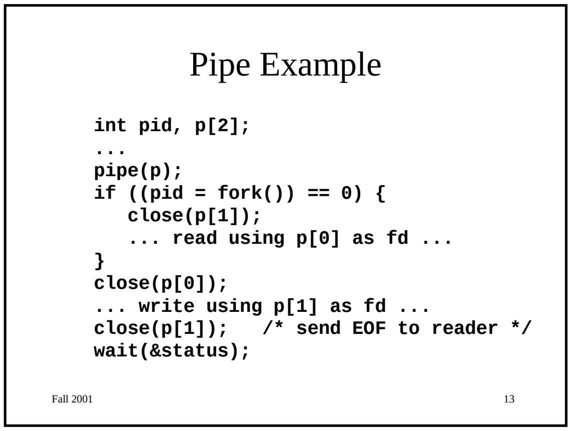
  `int pipe( int fd[2] );`

  return **`0`** upon success and **`–1`** upon failure

  `fd[0]` is open for reading

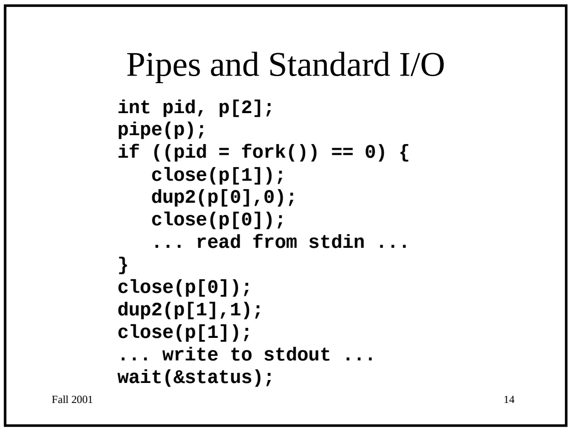
  `fd[1]` is open for writing

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



Pipe Example

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



Pipes and Standard I/O

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