CSci 4061
Introduction to Operating Systems

Input/Output: Low-level cont'd

File descriptor magic

Chapter 4

File Descriptor Inheritance

FDs and FD table are inherited by children at `fork()` time!

```
int fd, fd1, fd2, pid;
fd = open (“my_file”, O_RDONLY, 0);
pid = fork ();
if (pid != 0) {
        read (fd, …);
        fd1 = open (“foo”, …);
}
else {
        read (fd, …); // the same fd
        fd2 = open (“bar”, …);
        …
}
```

Important for pipelines (later):

shell> `cat foo.bar | grep jon`

Re-direction: another key CS concept!

- What is the default input/output location?

```
shell> cat foo.bar
shell> cat foo.bar > baz.out
```

What is this? Output redirection

How does this work?

Let's build up to this

1. duplicating file descriptors
2. fd preservation

1. Duplicating File Descriptors

Sometimes you need to change what an `fd` points to (why? redirection)

```
#include <unistd.h>
int dup2 (int fd1, int fd2);
```

Closes `fd2` (if open) -> frees up `fd2`

Makes `fd2` now point to what `fd1` points to

Output Re-direction

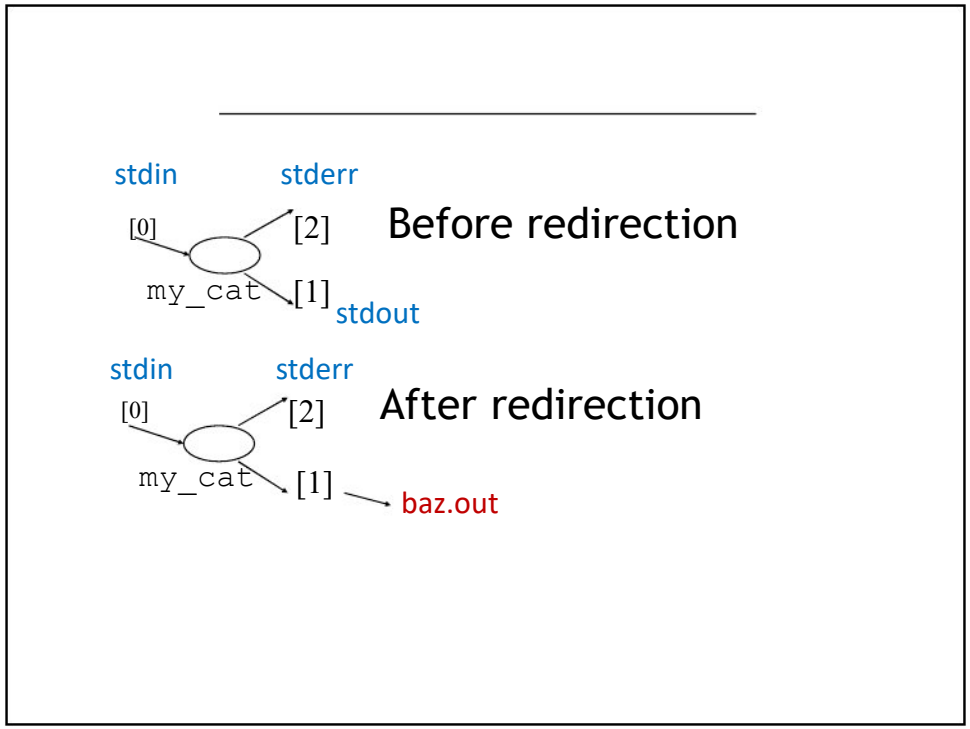
shell> `cat foo.bar > baz.out`

What is happening? … writes to stdout are redirected to `baz.out` … but how?

Output redirection (`my_cat.c`)

```
fd = open ("baz.out", O_CREAT|O_WRONLY, …);
dup2 (fd, 1); // 1 now refers to fd
close (fd);   // might as well …
write (1, …); // writes to "standard out" => baz.out
```

Output Re-direction (cont'd)



Output Re-direction (cont'd)

Output redirection (my_cat.c)

```
fd = open ("baz.out", O_CREAT|O_WRONLY, …);
dup2 (fd, 1); //close stdout and 1 refer to fd
close (fd);    // not needed but good style
write (1, …); // writes to "standard out" => baz.out
```

"lost" stdout - how can we re-open it?

```
fd = open ("/dev/tty", O_WRONLY);
```

Input Re-Direction

- How would we handle input redirection?

```
shell> wc -c
Hi there jon
^D
shell>13
```

Suppose I want to use a file instead

```
shell> wc -c < mydata.txt
```

Input Re-Direction (cont'd)

my_wc.c:

```
fd = open ("mydata.txt", O_RDONLY, …);
dup2 (fd, 0);
close (fd); // not needed but good style


// reads from "standard in" directed to mydata.txt
read (0, …);
```

Re-Direction

- Re-directing within a single process not so useful, let's return to the shell
  - If I want to read/write to a file, just do it!

- How does the shell actually do it for us?

shell> `cat foo.bar > baz.out`

// `cat` delivers output to stdout and we don't
// want to **modify** the code of `cat` or `ls` or ….

Step 2: fd preservation

fd Preservation

```
int fd, pid;
char fd_str[2];

pid = fork ();
if (pid != 0) {
    read (fd, …);
}
else {
    fd = open (“source_file”, O_RDONLY, 0);
    read (fd, ….);
    sprintf (fd_str, “%d”, fd);
    // fd’s preserved through exec!
    execl (“./foo (char*)”foo”, (char*)fd_str);
        …
}
```

foo.c

```
int main (int argc, char *argv[]) {
    int fd;
    fd = atoi (argv[2];
    read (fd, … ); // will read from “source_file”
…
}
```

I/O Redirection by the Shell

cat foo > bar

wc < baz

- Preservation of fd's
- Go back to `dup2`

Inside `cat`

…

```
write (1, …);
```

…..

By default `cat` writes to stdout
This is true of virtually all shell commands/programs
(read from stdin, write to stdout)
How do we get `cat` to write somewhere else?

The Shell: dup + preservation

shell> `myprog source > output`

inside "myprog"

`write (1, …);`

```
int pid;
pid = fork ();
if (pid != 0) {   // shell parent
        …
        wait (NULL);
}
else {// shell child
        // redirect name of file provided to the shell '>' case
        int o_fd;
        char *s_file; // extract from argv -- source
        char *o_file; // extract from argv -- output
        o_fd = open (o_file, O_WRONLY);
        dup2 (o_fd, 1);
        execl ("myprog …", (char*)&s_file, NULL);
}
```